INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

Coprocessor for Decoding Multi-Valued Pseudo-Random Sequence Patterns

by

ADRIAN MOICA

A thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the
requirements for the degree of
Master of Applied Sciences
in Computer Engineering

OTTAWA-CARLETON INSTITUTE FOR ELECTRICAL AND COMPUTER ENGINEERING

School of Information Technology and Engineering
Faculty of Engineering

University of Ottawa, Ontario, CANADA

September 27, 2000

This work is dedicated to my mum and dad,

who are far away,

but always in my thoughts.

ABSTRACT

This thesis discusses generalized Pseudo-Random Multi-Valued Sequence (PRMVS) encoding and corresponding PRMVS-to-natural code conversion algorithms. A hardware architecture implementing in real-time this generalized code conversion algorithm is implemented as a programmable Pseudo-Random DECoder (PRDEC) coprocessor. This coprocessor is intended to work in microprocessor-based embedded systems where it is seen as an 8-bit memory mapped device by the system microprocessor. The coprocessor architecture is implemented and simulated using VHDL.

ACKNOWLEDGEMENTS

I would like to express my deep gratitude and appreciation to my supervisor Dr. Emil M. Petriu for the support, guidance and encouragement he always was ready to give me during the whole period that took me to complete this work.

My biggest thanks go to my wife Anca whose love and encouragement made it possible for me to do this work and also to my two active young children Christina and Alexander who had to patiently wait for it to be completed so we can get back to the business of living.

I also thank AiT Corporation for allowing me the use of their computers and CAD/CAE tools to verify the VHDL models in this work.

LIST OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CPLD	Complex Programmable Logic Devices
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FSR	Feedback Shift Register
GF	Galois Field
LFSR	Linear Feedback Shift Register
PRBS	Pseudo-Random Binary Sequence
PRDEC	Pseudo-Random DECoder
PRMVA	Pseudo-Random Multi-Valued Array
PRMVS	Pseudo-Random Multi-Valued Sequence
PRS	Pseudo-Random Sequence
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register Transfer Logic
SOC	System On Chip
SRAM	Static Random Access Memory
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
μP	Microprocessor

Chapter 1. INTRODUCTION

A very important requirement for the fields of position control, robotics, and machine vision applications is the possibility to know the spatial position with the least delay possible. Absolute position measurement is more convenient than its incremental type alternative as it allows for an instant position recovery, and also has the notable advantage of not allowing errors to accumulate as it may happen in the case of incremental position recovery [Pe98].

Absolute position recovery paradigm using pseudo-random encoding can be stated as follows: "Given a pseudo-random encoded object-surface, whose grid node coordinates are defined in a 3-D reference frame, and given a perspective image showing only a part/window of the pseudo-random array, recover the 3-D position and orientation of the observer's frame relative to the object reference frame." [Pe96]

1.1 Pseudo-Random Encoding and Applications to Absolute Position Recovery

The traditional approach for the absolute position encoding requires that each quantization interval of a scale be marked with a distinct *n*-bit code. The easiest way to encode the absolute position is by using binary symbols, as shown in Figure 1-1, where the two symbols of the alphabet are represented by the colors *white* and respectively *black*.

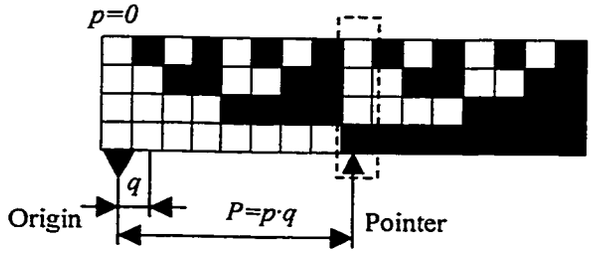


Figure 1-1. Absolute position encoding using a 4-bit code

The position P of the pointer with respect to the origin of this scale is estimated by reading the specific code $\{x(k)\in\{0,1\} \ / \ k=1,\ldots,4\}$ written on the quantization interval currently facing the pointer:

$$P = p \cdot q = (\sum_{k=1}^{4} x(k) \cdot 2^{k-1}) \cdot q \quad / \quad x(k) \in \{0,1\}$$

Equation 1-1

where q is the quantization step and the two symbols *white* and *black* of the alphabet are replaced by their numeric equivalent *0* and respectively *1*.

It can be seen that the number of code tracks on the scale increases proportionally with the desired measuring resolution, prohibiting the use of this straightforward absolute encoding method for the high resolution applications required by today's industrial practice.

A non-traditional encoding method is to use a pseudo-random sequence to encode the position, method that represents a more efficient alternative to absolute position measurement applications. This method based on the properties of pseudo-random sequences has the notable advantage of requiring only one track that contains one symbol of code per quantization interval. This advantage makes absolute position measurement more accessible for very high-resolution applications [Ara84, Pe98].

The pseudo-random sequence is generated by a special type of shift register having an appropriate modulo feedback [Pe88-2]. The absolute position recovery is based on the pseudo-random sequence window property that says that if a window of prescribed size is slid over the encoded track then each of the possible symbol nonzero combinations is seen through the window exactly once.

Figure 1-2 illustrates an example of absolute position encoding and recovery using a pseudo-random binary sequence. The two symbols of the binary alphabet are represented by the colors *white* and respectively *black*. In this example, a 5-bit shift register having a

certain modulo-two feedback equation generates a 31-term Pseudo-Random Binary Sequence (PRBS). It is easy to verify in Figure 1-2 that any code seen through the 5-bit sliding window is unique.

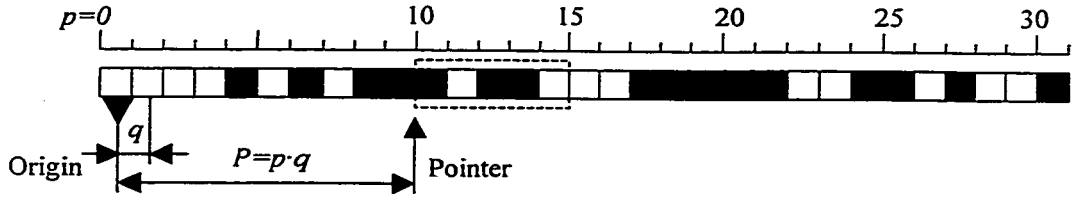


Figure 1-2. Absolute position encoding using a PRBS

The position P of the pointer with respect to the origin of this scale is measured by reading the code $\{ x(k) \in \{0,1\} / k=1,...,5 \}$ inside the sliding window (where the two symbols *white* and *black* of the alphabet have been replaced by their numeric equivalent *0* and respectively *1*) followed by a direct conversion to the more convenient natural code.

A more compact absolute position encoding can be obtained by using Pseudo-Random Multi-Valued Sequences (PRMVSs) where sequence elements are entries taken from an alphabet with more than two symbols. Figure 1-3 shows an example of absolute position encoding and recovery using a PRMVS based on a 3-symbol alphabet. The three symbols of the alphabet are represented by the colors *white, gray,* and respectively *black.* It is easy to verify in Figure 1-3 that any code seen through the 3-bit sliding window is unique.

Figure 1-3. Absolute position encoding using a PRMVS

The position P of the pointer with respect to the origin of this scale is estimated, as in the previous case, by reading the specific code $\{ x(k) \in \{0,1,2\} / k=1,...,3 \}$ inside the sliding

window (where the three symbols *white*, *gray*, and *black* of the alphabet have been replaced by their numeric equivalent *0*, *1* and respectively *2*) followed by a direct conversion to the more convenient natural code.

Compared to the traditional approach for the absolute position encoding based on the straightforward encoding method the resulting number of code tracks on the scale at the same resolution decreases proportionally with the size of the alphabet used. More, at the same resolution the width of the sliding window decreases proportionally with the size of the alphabet used.

1.2 Thesis Motivation

The advantage of using pseudo-random encoding for absolute position recovery, as opposed to the straightforward encoding method, relies on the fact that it requires only one track that contains one symbol of code per quantization interval.

The essential disadvantage of the pseudo-random encoding is the pseudo-randomness of the code read for each position. In order to be useful for any practical application this pseudo-random code must be converted to the natural code.

Because of the complexity of PRMVS most of the applications involving this type of encoding use software programs to implement pseudo-random-to-natural code conversion algorithms. Such an approach is obviously time consuming although the performance could be improved to a certain degree by using a higher performance computer.

The motivation for this work was the need for a real-time PRMVS-to-natural decoding solution. Due to the complexity of the decoding algorithm, real-time performance could be reached only by a hardware implementation.

Field Programmable Gate Arrays (FPGAs) technology has been chosen for this implementation as it allows to efficiently implement (design and prototyping costs [Sm97]) the very complex hardware structure needed by high resolution PRMVS-to-natural decoders.

1.3 Thesis Organization and Contributions

Chapter 2 gives an overview of the finite field and pseudo-random sequence theory. Discussed are both finite fields of order prime number and finite fields of order power of prime number, as well as practical methods for construction of these finite fields. With this background, the methods used to generate Pseudo-Random Sequences (PRSs) in each type of field are then presented.

Chapter 2 concludes with a review of PRS properties that can be used for pseudo-random-to-natural code conversion. Three types of decoding algorithms are analyzed in detail and compared to each other in terms of performances. The best performing algorithm among them is chosen for a physical hardware implementation as described in Chapter 3.

Chapter 3 presents the architecture of the Pseudo-Random DECoder (PRDEC) coprocessor. The architectural details include the external connectivity and control register mapping, hierarchy approach, and design optimization as a trade-off to improve the rapport of cost versus performance. The design follows a top-down approach where each of the coprocessor's high level modules is analyzed in depth from functionality point of view. These high level modules are represented by the Input/Output Unit that connects the PRDEC Core to an external microprocessor, the Control Unit that manages all the data flow and control signals inside the chip, the PRS Generator Unit that is used to generate internally the PRS according to the arithmetic of a specific finite field, the Milestone Database Unit that is a highly organized memory space used for storing/retrieving of certain PRS windows called milestones, and the Decoder Unit that

compares the current PRS window from the PRS generator unit against the milestones stored in the milestone database unit.

A brief introduction to the VHDL is given next, followed by a brief description of the content of the VHDL libraries used by the PRDEC coprocessor implementation. Then the physical implementation of each of the coprocessor's high level modules is discussed (where each module is analyzed from data flow and control signals point of view), and then simulated extensively. At the end of this section the module used to exercise the PRDEC coprocessor is described as a driver which emulates the behavior of a microprocessor interface. This driver is actually used in the PRDEC coprocessor testbench simulation to obtain the experimental results.

Chapter 3 concludes with presenting a performance evaluation of the PRDEC coprocessor in terms of resource utilization and processing speed. Some hints about improving the performances of the current PRDEC coprocessor architecture are also given at the end of this section.

Chapter 4 presents the conclusion of this thesis where a brief overview of specialized versus generic PRDEC coprocessor implementation is given. Future research issues are identified and discussed. These research areas are recommended to be explored in order to improve the architecture and performances of the PRDEC coprocessor design.

The contributions of this thesis are:

1. Generalization of the pseudo-random encoding and code conversion algorithm. Most of the work presented in literature was done for pseudo-random binary encoding, where the encoding symbols are elements of the binary finite field *GF(2)* or finite fields which are powers of *GF(2)*. The generalization of these concepts applicable to any finite field $GF(p^m)$, where p represents any prime and m represents a positive integer, is done by using a practical how-to approach that explains how the finite

fields are built and how to generate PRSs based on the arithmetic defined on those fields.

2. Design of a specialized hardware processor for real-time implementation of the generalized code conversion algorithm over a finite field $GF(p^m)$. This processor offers a practical pseudo-random-to-natural decoding solution for the high resolution encoding meaningful for industrial applications.

Chapter 2. PSEUDO-RANDOM ENCODING

This chapter presents the pseudo-random sequences, how they are generated and some of their useful properties along with their decoding mechanism.

Pseudo-Random Sequences (PRSs) are usually used in a large range of applications such as position measurement, communications, encryption, and system identification. They have been applied successfully in ingenious applications such as error-correcting codes, cryptography, interplanetary satellite picture transmission, precision measurements in physiology and general relativity, and even concert hall acoustics [Sch97].

2.1 Pseudo-Random Sequences

PRSs, also called pseudo-noise (PN) sequences, maximal-length shift-register sequences, or *m*-sequences, are finite-length sequences generated by a Linear Feedback Shift Register (LFSR). Depending of the application there are various types of Linear Feedback Shift Registers [MaS76].

Because pseudo-random encoding theory is based on the finite fields theory, a short review of the finite fields theory is given first.

The finite fields are of two types: finite fields of order prime number denoted *GF(p)* and finite fields of order power of prime number denoted *GF(q)*. From now on, unless otherwise stated, p will represent a prime number, and q will represent a prime power.

2.1.1 Definition of the Finite Fields

By definition, a field, denoted $(F, +, \cdot)$, is a set of elements F endowed with two operations "+" and "·", called addition and multiplication, satisfying the following properties:

1) The group $(F, +)$ is abelian under addition:
   a. Addition in F is commutative:

      $\forall x,y \in F$: $\quad x + y = y + x$

   b. Addition in F is associative:

      $\forall x,y,z \in F$: $\quad x + (y + z) = (x + y) + z$

   c. There exists an identity element in F for addition:

      $\exists 0 \in F$ such that $\forall x \in F$: $\quad x + 0 = 0 + x = x$

   d. For each element in F there exists a symmetrical element for addition:

      $\forall x \in F$, $\exists (-x) \in F$ such that $x + (-x) = (-x) + x = 0$

2) The group $(F-\{0\}, \cdot)$ is abelian under multiplication:
   a. Multiplication in F is commutative:

      $\forall x,y \in F$: $\quad x \cdot y = y \cdot x$

   b. Multiplication in F is associative:

      $\forall x,y,z \in F$: $\quad x \cdot (y \cdot z) = (x \cdot y) \cdot z$

   c. There exists an identity element in F for multiplication:

      $\exists 1 \in F$ such that $\forall x \in F$: $\quad x \cdot 1 = 1 \cdot x = x$

   d. For each element in F there exists a symmetrical element for multiplication:

      $\forall x \in F$, $\exists (x^{-1}) \in F$ such that $x \cdot (x^{-1}) = (x^{-1}) \cdot x = 0$

3) Distributive laws between the two operations:

a. $\forall x,y,z \in F$: $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$

b. $\forall x,y,z \in F$: $(x + y) \cdot z = (x \cdot z) + (y \cdot z)$

The symbol *0* is the identity element of the additive group *(F, +)* and the symbol *1* is the identity element of the multiplicative group *(F, ·)*.

Examples of fields are the rational numbers **Q**, the real numbers **R**, and the complex numbers **C**; and because each of these fields contains an infinite number of elements they are called infinite fields.

As opposed to infinite fields, a field that contains a finite number of elements is called a finite field, where the number of elements in the field is called the *order* of the field. Finite fields are also called Galois fields, in honor of Evariste Galois who discovered them; and usually a finite field of size q is denoted by *GF(q)*.

2.1.2 GF(p) – Finite Fields of Order Prime Number

The finite field whose order is a prime number represents the simplest class of finite fields. They are known in literature as Galois fields of order p and are denoted by *GF(p)*.

A finite field *GF(p)* of order p, where p is a prime number, has:

- p elements *{0, 1, 2, ..., p-1}* represented by the integers modulo p;
- the addition and multiplication carried out modulo p.

The following examples illustrate a number of finite fields whose order is a prime number. For the sake of space, the examples are limited to the finite fields *GF(2)*, *GF(3)*, and *GF(5)*.

Finite field GF(2)

The finite field *GF(2)* is the field of the well known pseudo-random binary sequences. The addition and multiplication tables carried out modulo *2* over the 2-element finite field *GF(2) = {0, 1}* are shown in Figure 2-1, where:

$$0 + 0 = 0 = 0 \text{ modulo } 2 \qquad 0 \cdot 0 = 0 = 0 \text{ modulo } 2$$
$$0 + 1 = 1 + 0 = 1 = 1 \text{ modulo } 2 \qquad 0 \cdot 1 = 1 \cdot 0 = 0 = 0 \text{ modulo } 2$$
$$1 + 1 = 2 = 0 \text{ modulo } 2 \qquad 1 \cdot 1 = 1 = 1 \text{ modulo } 2$$

+	0	1
0	0	1
1	1	0

·	0	1
0	0	0
1	0	1

Figure 2-1. Addition and multiplication tables for GF(2)

Finite field GF(3)

The addition and multiplication tables carried out modulo *3* over the 3-element finite field *GF(3) = {0, 1, 2}* are shown in Figure 2-2, where:

$$0 + 0 = 0 = 0 \text{ modulo } 3 \qquad 0 \cdot 0 = 0 = 0 \text{ modulo } 3$$
$$0 + 1 = 1 + 0 = 1 = 1 \text{ modulo } 3 \qquad 0 \cdot 1 = 1 \cdot 0 = 0 = 0 \text{ modulo } 3$$
$$0 + 2 = 2 + 0 = 2 = 2 \text{ modulo } 3 \qquad 0 \cdot 2 = 2 \cdot 0 = 0 = 0 \text{ modulo } 3$$
$$1 + 1 = 2 = 2 \text{ modulo } 3 \qquad 1 \cdot 1 = 1 = 1 \text{ modulo } 3$$
$$1 + 2 = 2 + 1 = 3 = 0 \text{ modulo } 3 \qquad 1 \cdot 2 = 2 \cdot 1 = 2 = 2 \text{ modulo } 3$$
$$2 + 2 = 4 = 1 \text{ modulo } 3 \qquad 2 \cdot 2 = 4 = 1 \text{ modulo } 3$$

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

·	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

Figure 2-2. Addition and multiplication tables for GF(3)

Finite field GF(5)

The addition and multiplication tables carried out modulo *5* over the 5-element finite field *GF(5) = {0, 1, 2, 3, 4}* are shown in Figure 2-3, where:

$0 + 0 = 0 = 0$ *modulo 5*
$0 + 1 = 1 + 0 = 1 = 1$ *modulo 5*
$0 + 2 = 2 + 0 = 2 = 2$ *modulo 5*
$0 + 3 = 3 + 0 = 3 = 3$ *modulo 5*
$0 + 4 = 4 + 0 = 4 = 4$ *modulo 5*
$1 + 1 = 2 = 2$ *modulo 5*
$1 + 2 = 2 + 1 = 3 = 3$ *modulo 5*
$1 + 3 = 3 + 1 = 4 = 4$ *modulo 5*
$1 + 4 = 4 + 1 = 5 = 0$ *modulo 5*
$2 + 2 = 4 = 4$ *modulo 5*
$2 + 3 = 3 + 2 = 5 = 0$ *modulo 5*
$2 + 4 = 4 + 2 = 6 = 1$ *modulo 5*
$3 + 3 = 6 = 1$ *modulo 5*
$3 + 4 = 4 + 3 = 7 = 2$ *modulo 5*
$4 + 4 = 8 = 3$ *modulo 5*

$0 \cdot 0 = 0 = 0$ *modulo 5*
$0 \cdot 1 = 1 \cdot 0 = 0 = 0$ *modulo 5*
$0 \cdot 2 = 2 \cdot 0 = 0 = 0$ *modulo 5*
$0 \cdot 3 = 3 \cdot 0 = 0 = 0$ *modulo 5*
$0 \cdot 4 = 4 \cdot 0 = 0 = 0$ *modulo 5*
$1 \cdot 1 = 1 = 1$ *modulo 5*
$1 \cdot 2 = 2 \cdot 1 = 2 = 2$ *modulo 5*
$1 \cdot 3 = 3 \cdot 1 = 3 = 3$ *modulo 5*
$1 \cdot 4 = 4 \cdot 1 = 4 = 4$ *modulo 5*
$2 \cdot 2 = 4 = 4$ *modulo 5*
$2 \cdot 3 = 3 \cdot 2 = 6 = 1$ *modulo 5*
$2 \cdot 4 = 4 \cdot 2 = 8 = 3$ *modulo 5*
$3 \cdot 3 = 9 = 4$ *modulo 5*
$3 \cdot 4 = 4 \cdot 3 = 12 = 2$ *modulo 5*
$4 \cdot 4 = 16 = 1$ *modulo 5*

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

·	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Figure 2-3. Addition and multiplication tables for GF(5)

Primitive elements of finite fields

Every finite field contains at least one primitive element from which all the others elements of the field, except the element *0* (that is the identity element for addition), can be obtained by multiplication with itself. This observation implies that the elements of the field, except the element *0*, form a cyclic group under multiplication [MaS77].

A minimal polynomial over *GF(p)* is, by definition, the lowest degree monic polynomial (it has the leading coefficient 1) with coefficients from *GF(p)*. This definition will be used later, in the next subchapter, in defining the arithmetic over finite fields of order power of prime number.

Example

The elements *2* and *3*, belonging to *GF(5) = {0, 1, 2, 3, 4}*, are primitive elements for the cyclic group *{1, 2, 3, 4}* under multiplication modulo *5* [Tr93]:

$2^1 = 2 = 2$ *modulo* 5	$3^1 = 3 = 3$ *modulo* 5
$2^2 = 4 = 4$ *modulo* 5	$3^2 = 9 = 4$ *modulo* 5
$2^3 = 8 = 3$ *modulo* 5	$3^3 = 27 = 2$ *modulo* 5
$2^4 = 16 = 1$ *modulo* 5	$3^4 = 81 = 1$ *modulo* 5
$2^5 = 32 = 2$ *modulo* $5 = 2^1$	$3^5 = 243 = 3$ *modulo* $5 = 3^1$
$2^6 = 64 = 4$ *modulo* $5 = 2^2$	$3^6 = 729 = 4$ *modulo* $5 = 3^2$
$2^7 = 128 = 3$ *modulo* $5 = 2^3$	$3^6 = 2187 = 2$ *modulo* $5 = 3^3$
etc.	etc.

2.1.3 GF(p^m) – Finite Fields of Order Power of Prime Number

The finite fields whose order is a power of a prime number represents the most general class of finite fields. They are known in literature as polynomial fields or Galois fields of order $q=p^m$ and are denoted by $GF(p^m)$. It is proven that finite fields of order p^m exists for every prime numbers p and positive integers m [Rot90], that these fields are unique [Rom92], and that they have exactly p^m elements [Rot90]. Any finite field $GF(p^m)$ whose order is a power of a prime number contains a subfield $GF(p)$ whose order is that prime.

Finite fields $GF(p)$ of order prime number are defined by taking the integers modulo p with the operation of addition and multiplication (also done modulo p). However finite fields $GF(p^m)$ of order power of prime number can not be defined in this straightforward way because the integers modulo $q=p^m$ do not form a field. In this case the nonzero elements are expressed as the first $q-1$ powers of some primitive element t of the field $GF(q)$ [GrT74]. In order to do arithmetic in such a finite field, whose order is power of prime, there is necessary to have a primitive element and its minimal polynomial [Rom92]. It should be noted that these minimal polynomial are not unique.

A polynomial field $GF(p^m)$ of order $q=p^m$, where p is a prime number and m is any positive integer, has:

- q elements $\{0, 1, 2, ..., q-1\}$ denoted as integers modulo q that represent the set of all polynomials in x of degree at most $m-1$ and coefficients from $GF(p)$;
- the addition carried out modulo p over the polynomial representation;
- the multiplication performed over the polynomial representation by multiplying modulo p and taking the remainder when dividing by a primitive polynomial $h(x)$.

Example

The field $GF(2^3)$ has $q = p^m = 2^3 = 8$ elements that are polynomials whose coefficients represent all possible three-element combinations of the elements of its prime order subfield $GF(2) = \{0, 1\}$.

The elements of the field $GF(2^3)$ in polynomial representation are: 0, 1, x, $x+1$, x^2, x^2+1, x^2+x, and x^2+x+1.

The mechanism used to obtain this polynomial representation is summarized in the following table:

3-element combination	Polynomial equivalent	Polynomial representation
000	$0\cdot x^2+0\cdot x^1+0\cdot x^0$	0
001	$0\cdot x^2+0\cdot x^1+1\cdot x^0$	1
010	$0\cdot x^2+1\cdot x^1+0\cdot x^0$	x
011	$0\cdot x^2+1\cdot x^1+1\cdot x^0$	$x+1$
100	$1\cdot x^2+0\cdot x^1+0\cdot x^0$	x^2
101	$1\cdot x^2+0\cdot x^1+1\cdot x^0$	x^2+1
110	$1\cdot x^2+1\cdot x^1+0\cdot x^0$	x^2+x
111	$1\cdot x^2+1\cdot x^1+1\cdot x^0$	x^2+x+1

The polynomial addition is performed modulo 2:

$$(x^2+x) + (x+1) = (x^2+2x+1) \text{ modulo } 2 = x^2+1$$

$$(x^2+x+1) + (x^2+x) = (2x^2+2x+1) \text{ modulo } 2 = 1$$

while the polynomial multiplication is performed modulo 2 and taking the remainder when dividing by a primitive polynomial. This statement implies that the polynomial multiplication in a finite of order power of prime is not unique and it depends always by the primitive polynomial.

The polynomial multiplication in $GF(2^3)$, when the primitive polynomial is $h(x) = x^3+x+1$, is done as follows:

$$(x^2+x) \cdot (x+1) = (x^3+2x^2+x) \text{(modulo 2)}$$

$$= x^3+x$$
$$= (x^3+x+1) \cdot 1 + 1$$
$$= h(x) \cdot 1 + 1$$
$$= 1$$

$(x^2+x+1) \cdot (x^2+x)$ $= (x^4+2x^3+2x^2+x)$*(modulo 2)*
$$= x^4+x$$
$$= (x^3+x+1) \cdot x + x^2+x+1$$
$$= h(x) \cdot x + x^2+x+1$$
$$= x^2+x+1$$

while for the primitive polynomial $h(x) = x^3+x^2+1$ it is done as follows:

$(x^2+x) \cdot (x+1)$ $= (x^3+2x^2+x)$*(modulo 2)*
$$= x^3+x$$
$$= (x^3+x^2+1) \cdot 1 + x^2+x+1$$
$$= h(x) \cdot 1 + x^2+x+1$$
$$= x^2+x+1$$

$(x^2+x+1) \cdot (x^2+x)$ $= (x^4+2x^3+2x^2+x)$*(modulo 2)*
$$= x^4+x$$
$$= (x^3+x^2+1) \cdot x + x^3$$
$$= h(x) \cdot x + (x^3+x^2+1) \cdot 1 + x^2+1$$
$$= h(x) \cdot 1 + x^2+1$$
$$= x^2+1$$

The minimal polynomial $h(x)$ of a primitive element of $GF(p^m)$ has degree m and it is called a primitive polynomial [MaS77]. This polynomial has coefficients from $GF(p)$ and it is irreducible over $GF(p)$. By definition a polynomial $h(x)$ with coefficients from $GF(p)$ is irreducible over $GF(p)$ if it is not the product of two polynomials of lower degree with coefficients from $GF(p)$ [MaS77].

Example

The polynomial $x^2+x+1=(x+2)^2$ is not irreducible over $GF(3)$, but x^2+x+2 and x^2+1 are [MaS77].

Calculating the remainder modulo $h(x) = x^m+h_{m-1}\cdot x^{m-1}+...+h_1\cdot x+h_0$ is the same as setting $h(x) = 0$ that is equivalent with replacing $x^m = -h_{m-1}\cdot x^{m-1}-...-h_1\cdot x-h_0$ [Rom92].

Note that in $GF(2^m)$ the above equivalency becomes

$$x^m = -h_{m-1}\cdot x^{m-1}-...-h_1\cdot x-h_0$$
$$= h_{m-1}\cdot x^{m-1}+...+h_1\cdot x+h_0$$

because $h+h=0 \Leftrightarrow h=-h$ for all $h \in GF(2)=\{0, 1\}$.

Addition over a finite field $GF(p^m)$ in polynomial representation is simply done as vector addition while the multiplication is not as simple, since it requires to reduce modulo $h(x)$ thus becoming impractical for larger fields [Rom92].

If the finite field $GF(p^m)$ is represented as powers of a primitive element t, so $GF(p^m) = GF(q) = \{0, 1, t, t^2, ..., t^{q-2}\}$, then multiplication becomes a trivial operation [Rom92] defined by the formula:

$$t^i \cdot t^j = t^{(i+j) \text{ modulo}(q-1)}$$

The two ways of representing the elements of a finite field $GF(p^m)$ can be combined in order to make the arithmetic easier using polynomial representation to perform the addition and power of a prime element to perform the multiplication.

Actually a finite field $GF(p^m)$ can have its q elements expressed in four possible ways:

- as a m-tuple of elements from $GF(p)$;
- as a polynomial of degree at most m with coefficients from $GF(p)$;
- as a power of a primitive element t that is a zero for $h(x)$;
- as a logarithm of base t.

The first two representations, as a m-tuple of elements from $GF(p)$ and as a polynomial of degree at most m with coefficients from $GF(p)$, are best suited for addition, while the last two representations, as a power of a primitive element t that is a zero for $h(x)$ and as a logarithm of base t, are best suited for multiplication [MaS77, Rom92]. Thus, the addition of two elements of a finite field $GF(p^m)$ is performed by adding the polynomials of degree at most q, or modulo q, while the multiplication of two elements of a finite field $GF(p^m)$ is the same as adding their logarithms modulo q, where $q = p^m$.

As stated in the previous subchapter, every finite field contains at least one primitive element from which all the other elements of the field, except the element 0 (that is the identity element for addition), can be obtained by multiplication of that primitive element with itself. This statement implies that the elements $\{1, t, t^2, ..., t^{q-2}\}$ belonging to a finite field $GF(q)-\{0\}$, where $GF(q) = GF(p^m)$, form a cyclic group under the operation of multiplication and that: $t^{q-1} = t^{q-2} \cdot t = 1$, $t^q = t^{q-1} \cdot t = 1 \cdot t = t$, $t^{q+1} = t^2$, etc.

For an easier representation, the primitive element t and its powers t^k can have assigned a symbol from the field of natural numbers **N**; thus, $GF(p^m) = GF(q) = \{0, 1, t, t^2, ..., t^{q-2}\}$ can be rewritten as $GF(p^m) = \{0, 1, 2, 3, ..., q-1\}$.

Following there are presented examples how to find the elements of, and construct the addition and multiplication tables over, finite fields whose order is a power of a prime number. For the sake of space, the examples will be given only for the finite fields $GF(2^2)$, $GF(2^3)$, and $GF(3^2)$.

Construction of the finite field $GF(2^2)$

For $p=2$, $m=2$, and primitive polynomial $h(x) = x^2+x+1$ construct $GF(2^2)$.

The field $GF(2^2)$ has $q = p^m = 2^2 = 4$ elements that are polynomials whose coefficients represent all possible two-element combinations of the elements of its prime order subfield $GF(2) = \{0, 1\}$.

The elements of a finite field can be written in several different ways. Thus, the elements of the finite field $GF(2^2)$ in polynomial representation are: 0, 1, x, and $x+1$; while, when representing them as a power of a primitive element t they are: 0, 1, t, and t^2.

The primitive element t is a root for the primitive polynomial $h(x) = x^2+x+1$ that implies that $h(t) = t^2+t+1 = 0$ or equivalent $t^2=t+1$ in $GF(2)$ according to the operation tables shown in Figure 2-1.

Following there is shown the equivalency between the representations of the elements of the finite field $GF(2^2)$ as polynomials and as powers of the primitive element t when the primitive polynomial is $h(x) = x^2+x+1$:

- the element 0 ⇔ the polynomial 0
- the element 1 ⇔ the polynomial 1
- the element t ⇔ the polynomial x
- the element $t^2 = t+1$ ⇔ the polynomial $x+1$, by replacing the elements t and 1 with their polynomial equivalents x and respectively 1.

Note that the elements $\{1, t, t^2\}$ belonging to $GF(2^2)-\{0\}$ form a cyclic group under multiplication and that $t^3 = t^2 \cdot t = 1$, $t^4 = t^3 \cdot t = t$, etc.

Table 2-1 presents the elements of the finite field $GF(2^2)$ in six different representations.

as a doublet	as a polynomial	as a power of t	as a logarithm	as an assigned symbol	as a natural number
00	*0*	*0*	*-∞*	*0*	*0*
01	*1*	*1*	*0*	*1*	*1*
10	*x*	*t*	*1*	*A*	*2*
11	*x+1*	t^2	*2*	*B*	*3*

Table 2-1. The elements of GF(2^2) generated by h(x) = x^2+x+1

The addition of the elements of the finite field $GF(2^2)$ is performed modulo *2* using the polynomial representation, as shown in Table 2-2.

as an assigned symbol	as a polynomial
$0+\alpha = \alpha+0 = \alpha$	Note: α represents any element from GF(2^2)
1+1 = 0	*(1)+(1) = (2)(modulo 2) = 0*
1+A = A+1 = B	*(1)+(x) = (x)+(1) = x+1*
1+B = B+1 = A	*(1)+(x+1) = (x+1)+(1) = (x+2) (modulo 2) = x*
A+A = 0	*(x)+(x) = (2x)(modulo 2) = 0*
A+B = B+A = 1	*(x)+(x+1) = (x+1)+(x) = (2x+1) (modulo 2) = 1*
B+B = 0	*(x+1)+(x+1) = (2x+2)(modulo 2) = 0*

Table 2-2. Addition operation over GF(2^2) as polynomial representation

The element multiplication in the finite field $GF(2^2)$ is performed modulo *2*, as shown in Table 2-3, using the power of the primitive element *t* representation and remembering that $t^2 = t+1$.

as an assigned symbol	as a power of the primitive element t
$0\cdot\alpha = \alpha\cdot 0 = 0$	Note: α represents any element from GF(2^2)
$1\cdot\alpha = \alpha\cdot 1 = \alpha$	Note: α represents any element from GF(2^2)
$A\cdot A = B$	$(t)\cdot(t) = t^2$
$A\cdot B = B\cdot A = 1$	$(t)\cdot(t^2) = (t^2)\cdot(t) = t^3 = 1$
$B\cdot B = A$	$(t^2)\cdot(t^2) = t^4 = t^3\cdot t = 1\cdot t = t$

Table 2-3. Multiplication operation over GF(2^2) as power of t representation

Both operations of addition and multiplication over $GF(2^2)$, described in Table 2-2 and respectively Table 2-3, are synthesized in Figure 2-4 where, for more readability, the symbols *{0, 1, A, B}* have been replaced by the natural numbers *{0, 1, 2, 3}*.

+	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

*	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

Figure 2-4. Addition and multiplication tables over GF(4)

Construction of the finite field $GF(2^3)$

For $p=2$, $m=3$, and primitive polynomial $h(x) = x^3+x+1$ construct $GF(2^3)$.

The field $GF(2^3)$ has $q = p^m = 2^3 = 8$ elements that are polynomials whose coefficients represent all possible three-element combinations of the elements of its prime order subfield $GF(2) = \{0, 1\}$.

The elements of a finite field can be written in several different ways. Thus, the elements of the finite field $GF(2^3)$ in polynomial representation are: 0, 1, x, $x+1$, x^2, x^2+1, x^2+x, and x^2+x+1; while, when representing them as a power of a primitive element t they are: 0, 1, t, t^2, t^3, t^4, t^5, and t^6.

The primitive element t is a root for $h(x) = x^3+x+1$ that implies that $h(t) = t^3+t+1 = 0$ or equivalent $t^3=t+1$ in $GF(2)$ according to the operation tables shown in Figure 2-1.

Following there is shown the equivalency between the representations of the elements of the finite field $GF(2^3)$ as polynomials and as powers of the primitive element t when the primitive polynomial is $h(x) = x^3+x+1$:

- the element 0 ⇔ the polynomial 0
- the element 1 ⇔ the polynomial 1

- the element t ⇔ the polynomial x
- the element t^2 ⇔ the polynomial x^2
- the element $t^3 = t+1$ ⇔ the polynomial $x+1$, by replacing the elements t and 1 with their polynomial equivalents x and 1.
- the element $t^4 = t^3 \cdot t$
  $= (t+1) \cdot t$
  $= t^2+t$
  ⇔ the polynomial x^2+x, by replacing the elements t^2 and t with their polynomial equivalents x^2 and respectively x.
- the element $t^5 = t^3 \cdot t^2$
  $= (t+1) \cdot t^2$
  $= t^3+t^2$
  $= t^2+t+1$
  ⇔ the polynomial x^2+x+1, by replacing the elements t^2, t, and 1 with their polynomial equivalents x^2, x, and respectively 1.
- the element $t^6 = t^3 \cdot t^3$
  $= (t+1) \cdot (t+1)$
  $= t^2+t^2$
  $= t^2+t+t+1$
  $= t^2+1$ *(modulo 2)*
  ⇔ the polynomial x^2+1, by replacing the elements t^2 and 1 with their polynomial equivalents x^2 and respectively 1.

Note that the elements $\{1, t, t^2, t^3, t^4, t^5, t^6\}$ belonging to $GF(2^3)-\{0\}$ form a cyclic group under multiplication and that $t^7 = t^6 \cdot t = 1$, $t^8 = t^7 \cdot t = t$, $t^9 = t^8 \cdot t = t^2$, etc.

Table 2-4 presents the elements of the finite field $GF(2^3)$ in six different representations.

as a triplet	as a polynomial	as a power of t	as a logarithm	as an assigned symbol	as a natural number
000	*0*	*0*	*-∞*	*0*	*0*
001	*1*	*1*	*0*	*1*	*1*
010	*x*	*t*	*1*	*A*	*2*
100	x^2	t^2	*2*	*B*	*3*
011	*x+1*	t^3	*3*	*C*	*4*
110	x^2+x	t^4	*4*	*D*	*5*
111	x^2+x+1	t^5	*5*	*E*	*6*
101	x^2+1	t^6	*6*	*F*	*7*

Table 2-4. The elements of GF(2^3) generated by h(x) = x^3+x+1

The element addition in this field is performed modulo *2* using the polynomial representation, as shown in Table 2-5.

as an assigned symbol	as a polynomial
$0+\alpha = \alpha+0 = \alpha$	Note: α represents any element from GF(2^3)
$1+1 = 0$	$(1)+(1) = (2)(modulo\ 2) = 0$
$1+A = A+1 = C$	$(1)+(x) = (x)+(1) = x+1$
$1+B = B+1 = F$	$(1)+(x^2) = (x^2)+(1) = x^2+1$
$1+C = C+1 = A$	$(1)+(x+1) = (x+1)+(1) = (x+2)(modulo\ 2) = x$
$1+D = D+1 = E$	$(1)+(x^2+x) = (x^2+x)+(1) = x^2+x+1$
$1+E = E+1 = D$	$(1)+(x^2+x+1) = (x^2+x+1)+(1) = (x^2+x+2)(modulo\ 2) = x^2+x$
$1+F = F+1 = B$	$(1)+(x^2+1) = (x^2+1)+(1) = (x^2+2)(modulo\ 2) = x^2$
$A+A = 0$	$(x)+(x) = (2x)(modulo\ 2) = 0$
$A+B = B+A = D$	$(x)+(x^2) = (x^2)+(x) = x^2+x$
$A+C = C+A = 1$	$(x)+(x+1) = (x+1)+(x) = (2x+1)(modulo\ 2) = 1$
$A+D = D+A = B$	$(x)+(x^2+x) = (x^2+x)+(x) = (x^2+2x)(modulo\ 2) = x^2$
$A+E = E+A = F$	$(x)+(x^2+x+1) = (x^2+x+1)+(x) = (x^2+2x+1)(modulo\ 2) = x^2+1$
$A+F = F+A = E$	$(x)+(x^2+1) = (x^2+1)+(x) = x^2+x+1$
$B+B = 0$	$(x^2)+(x^2) = (2x^2)(modulo\ 2) = 0$
$B+C = C+B = E$	$(x^2)+(x+1) = (x+1)+(x^2) = x^2+x+1$
$B+D = D+B = 2$	$(x^2)+(x^2+x) = (x^2+x)+(x^2) = (2x^2+x)(modulo\ 2) = x$
$B+E = E+B = C$	$(x^2)+(x^2+x+1) = (x^2+x+1)+(x^2) = (2x^2+x+1)(modulo\ 2) = x+1$
$B+F = F+B = E$	$(x^2)+(x^2+1) = (x^2+1)+(x^2) = (2x^2+1)(modulo\ 2) = 1$
$C+C = 0$	$(x+1)+(x+1) = (2x+2)(modulo\ 2) = 0$
$C+D = D+C = F$	$(x+1)+(x^2+x) = (x^2+x)+(x+1) = (x^2+2x+1)(modulo\ 2) = x^2+1$
$C+E = E+C = B$	$(x+1)+(x^2+x+1) = (x^2+x+1)+(x+1) = (x^2+2x+2)(modulo\ 2) = x^2$
$C+F = F+C = E$	$(x+1)+(x^2+1) = (x^2+1)+(x+1) = (x^2+x+2)(modulo\ 2) = x^2+x$
$D+D = 0$	$(x^2+x)+(x^2+x) = (2x^2+2x)(modulo\ 2) = 0$
$D+E = E+D = 1$	$(x^2+x)+(x^2+x+1) = (x^2+x+1)+(x^2+x) = (2x^2+2x+1)(modulo\ 2) = 1$
$D+F = F+D = C$	$(x^2+x)+(x^2+1) = (x^2+1)+(x^2+x) = (2x^2+x+1)(modulo\ 2) = x+1$
$E+E = 0$	$(x^2+x+1)+(x^2+x+1) = (2x^2+2x+2)(modulo\ 2) = 0$
$D+F = F+D = 2$	$(x^2+x+1)+(x^2+1) = (x^2+1)+(x^2+x+1) = (2x^2+x+2)(modulo\ 2) = x$
$F+F = 0$	$(x^2+1)+(x^2+1) = (2x^2+2)(modulo\ 2) = 0$

Table 2-5. Addition operation over GF(2^3) as polynomial representation

The element multiplication is performed modulo *2*, as shown in Table 2-6, using the power of the primitive element *t* representation and remembering that $t^3 = t+1$.

Both operations of addition and multiplication over $GF(2^3)$, described in Table 2-5 and respectively Table 2-6, are synthesized in Figure 2-5 where, for more readability, the

symbols *{0, 1, A, B, C, D, E, F}* have been replaced by the natural numbers *{0, 1, 2, 3, 4, 5, 6, 7}*.

as an assigned symbol	as a power of the primitive element t
$0 \cdot \alpha = \alpha \cdot 0 = 0$	Note: α represents any element from GF(2^3)
$1 \cdot \alpha = \alpha \cdot 1 = \alpha$	Note: α represents any element from GF(2^3)
$A \cdot A = B$	$(t) \cdot (t) = t^2$
$A \cdot B = B \cdot A = C$	$(t) \cdot (t^2) = (t^2) \cdot (t) = t^3$
$A \cdot C = C \cdot A = 1$	$(t) \cdot (t^3) = (t^3) \cdot (t) = t^4$
$A \cdot D = D \cdot A = 1$	$(t) \cdot (t^4) = (t^4) \cdot (t) = t^5$
$A \cdot E = E \cdot A = 1$	$(t) \cdot (t^5) = (t^5) \cdot (t) = t^6$
$A \cdot F = F \cdot A = 1$	$(t) \cdot (t^6) = (t^6) \cdot (t) = t^7 = 1$
$B \cdot B = D$	$(t^2) \cdot (t^2) = t^4$
$B \cdot C = C \cdot B = E$	$(t^2) \cdot (t^3) = (t^3) \cdot (t^2) = t^5$
$B \cdot D = D \cdot B = F$	$(t^2) \cdot (t^4) = (t^4) \cdot (t^2) = t^6$
$B \cdot E = E \cdot B = 1$	$(t^2) \cdot (t^5) = (t^5) \cdot (t^2) = t^7 = 1$
$B \cdot F = F \cdot B = A$	$(t^2) \cdot (t^6) = (t^6) \cdot (t^2) = t^8 = t^7 \cdot t = 1 \cdot t = t$
$C \cdot C = F$	$(t^3) \cdot (t^3) = t^6$
$C \cdot D = D \cdot C = 1$	$(t^3) \cdot (t^4) = (t^4) \cdot (t^3) = t^7 = 1$
$C \cdot E = E \cdot C = A$	$(t^3) \cdot (t^5) = (t^5) \cdot (t^3) = t^8 = t^7 \cdot t = 1 \cdot t = t$
$C \cdot F = F \cdot C = B$	$(t^3) \cdot (t^6) = (t^6) \cdot (t^3) = t^9 = t^8 \cdot t = t \cdot t = t^2$
$D \cdot D = A$	$(t^4) \cdot (t^4) = t^8 = t^7 \cdot t = 1 \cdot t = t$
$D \cdot E = E \cdot D = B$	$(t^4) \cdot (t^5) = (t^5) \cdot (t^4) = t^9 = t^8 \cdot t = t \cdot t = t^2$
$D \cdot F = F \cdot D = C$	$(t^4) \cdot (t^6) = (t^6) \cdot (t^4) = t^{10} = t^9 \cdot t = t^2 \cdot t = t^3$
$E \cdot E = C$	$(t^5) \cdot (t^5) = t^{10} = t^9 \cdot t = t^2 \cdot t = t^3$
$E \cdot F = F \cdot E = D$	$(t^5) \cdot (t^6) = (t^6) \cdot (t^5) = t^{11} = t^{10} \cdot t = t^3 \cdot t = t^4$
$F \cdot F = E$	$(t^6) \cdot (t^6) = t^{12} = t^{11} \cdot t = t^4 \cdot t = t^5$

Table 2-6. Multiplication operation over GF(2^3) as power of t representation

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	4	7	2	6	5	3
2	2	4	0	5	1	3	7	6
3	3	7	5	0	6	2	4	1
4	4	2	1	6	0	7	3	5
5	5	6	3	2	7	0	1	4
6	6	5	7	4	3	1	0	2
7	7	3	6	1	5	4	2	0

·	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	3	4	5	6	7	1
3	0	3	4	5	6	7	1	2
4	0	4	5	6	7	1	2	3
5	0	5	6	7	1	2	3	4
6	0	6	7	1	2	3	4	5
7	0	7	1	2	3	4	5	6

Figure 2-5. Addition and multiplication tables over GF(8)

Construction of the finite field $GF(3^2)$

For $p=3$, $m=2$, and primitive polynomial $h(x) = x^2+x+2$ construct $GF(3^2)$.

The field $GF(3^2)$ has $q = p^m = 3^2 = 9$ elements that are polynomials whose coefficients represent all possible two-element combinations of the elements of its prime order subfield $GF(3) = \{0, 1, 2\}$.

The elements of a finite field can be written in several different ways. Thus, the elements of the field $GF(3^2)$ in polynomial representation are: 0, 1, 2, x, $x+1$, $x+2$, $2x$, $2x+1$, and $2x+2$; while, when representing them as a power of a primitive element t they are: 0, 1, t, t^2, t^3, t^4, t^5, t^6, and t^7.

The primitive element t is a root for the primitive polynomial $h(x) = x^2+x+2$ that implies that $h(t) = t^2+t+2 = 0$ or equivalent $t^2 = (-t-2)(modulo\ 3) = 2t+1$ in $GF(2)$ according to the operation tables shown in Figure 2-2.

Following there is the equivalency between the representations of the elements of the finite field $GF(3^2)$ as polynomials and as powers of the primitive element t when the primitive polynomial is $h(x) = x^2+x+2$:

- the element 0 ⇔ the polynomial 0
- the element 1 ⇔ the polynomial 1
- the element t ⇔ the polynomial x
- the element $t^2 = 2t + 1$ ⇔ the polynomial $2x+1$, by replacing the elements $2t$ and 1 with their polynomial equivalents $2x$ and respectively 1.
- the element $t^3 = t^2 \cdot t = (2t+1) \cdot t = 2t^2 + t = 2(2t+1) + t = 4t + 2 + t$ ⇔ the polynomial $2x+2$, by replacing the elements $2t$ and 1 with their polynomial equivalents $2x$ and respectively 1.

$= (2t + 2)$ *(modulo 3)*

- the element $t^4 = t^3 \cdot t$ ⇔ the polynomial *2*, by replacing the element *2*
  $= (2t+2) \cdot t$ with its polynomial equivalent *2*.
  $= 2t^2 + 2t$
  $= 2(2t+1) + 2t$
  $= 4t + 2 + 2t$
  $= 2$ *(modulo 3)*
- the element $t^5 = t^4 \cdot t$ ⇔ the polynomial *2x*, by replacing the element *2t*
  $= 2t$ with its polynomial equivalent *2x*.
- the element $t^6 = t^5 \cdot t$ ⇔ the polynomial *x+2*, by replacing the elements *t*
  $= 2t \cdot t$ and *2* with their polynomial equivalents *x* and
  $= 2t^2$ respectively *2*.
  $= 2(2t+1)$
  $= 4t + 2$
  $= (t + 2)$ *(modulo 3)*
- the element $t^7 = t^6 \cdot t$ ⇔ the polynomial *x+1*, by replacing the elements *t*
  $= (t+2) \cdot t$ and *1* with their polynomial equivalents *x* and
  $= t^2 + 2t$ respectively *1*.
  $= (2t+1) + 2t$
  $= 2t + 1 + 2t$
  $= (t+1)$ *(modulo 3)*

Note that the elements $\{1, t, t^2, t^3, t^4, t^5, t^6, t^7\}$ belonging to $GF(3^2)-\{0\}$ form a cyclic group under multiplication and that $t^8 = t^7 \cdot t = 1$, $t^9 = t^8 \cdot t = t$, $t^{10} = t^9 \cdot t = t^2$, etc.

Table 2-7 presents the elements of the finite field $GF(3^2)$ in six different representations.

The element addition in this field is performed modulo *3* using the polynomial representation, as shown in Table 2-8, while the element multiplication is performed modulo *3* using the power of the primitive element *t* representation and remembering that $t^3 = t+1$, as shown in Table 2-9.

as a doublet	as a polynomial	as a power of t	as a logarithm	as an assigned symbol	as a natural number
00	*0*	*0*	*-∞*	*0*	*0*
01	*1*	*1*	*0*	*1*	*1*
10	*x*	*t*	*1*	*A*	*2*
21	*2x+1*	t^2	*2*	*B*	*3*
22	*2x+2*	t^3	*3*	*C*	*4*
02	*2*	t^4	*4*	*D*	*5*
20	*2x*	t^5	*5*	*E*	*6*
02	*x+2*	t^6	*6*	*F*	*7*
11	*x+1*	t^7	*7*	*G*	*8*

Table 2-7. The elements of GF(3^2) generated by h(x) = x^2+x+2

as assigned an symbol	as a polynomial
$0+\alpha = \alpha+0 = \alpha$	Note: α represents any element from GF(3^2)
1+1 = D	*(1)+(1) = 2*
1+A = A+1 = G	*(1)+(x) = (x)+(1) = x+1*
1+B = B+1 = C	*(1)+(2x+1) = (2x+1)+(1) = 2x+2*
1+C = C+1 = E	*(1)+(2x+2) = (2x+2)+(1) = 2x*
1+D = D+1 = 0	*(1)+(2) = (2)+(1) = (3)(modulo 3) = 0*
1+E = E+1 = B	*(1)+(2x) = (2x)+(1) = 2x+1*
1+F = F+1 = A	*(1)+(x+2) = (x+2)+(1) = (x+3)(modulo 3) = x*
1+G = G+1 = F	*(1)+(x+1) = (x+1)+(1) = x+2*
A+A = E	*(x)+(x) = 2x*
A+B = B+A = 1	*(x)+(2x+1) = (2x+1)+(x) = (3x+1)(modulo 3) = 1*
A+C = C+A = D	*(x)+(2x+2) = (2x+2)+(x) = (3x+2)(modulo 3) = 2*
A+D = D+A = F	*(x)+(2) = (2)+(x) = x+2*
A+E = E+A = 0	*(x)+(2x) = (2x)+(x) = (3x)(modulo 3) = 0*
A+F = F+A = C	*(x)+(x+2) = (x+2)+(x) = 2x+2*
A+G = G+A = B	*(x)+(x+1) = (x+1)+(x) = 2x+1*
B+B = F	*(2x+1)+(2x+1) = (4x+2)(modulo 3) = x+2*
B+C = C+B = A	*(2x+1)+(2x+2) = (2x+2)+(2x+1) = (4x+3)(modulo 3) = x*
B+D = D+B = E	*(2x+1)+(2) = (2)+(2x+1) = (2x+3)(modulo 3) = 2x*
B+E = E+B = G	*(2x+1)+(2x) = (2x)+(2x+1) = (4x+1)(modulo 3) = x+1*
B+F = F+B = 0	*(2x+1)+(x+2) = (x+2)+(2x+1) = (3x+3)(modulo 3) = 0*
B+G = G+B = D	*(2x+1)+(x+1) = (x+1)+(2x+1) = (3x+2)(modulo 3) = 2*
C+C = G	*(2x+2)+(2x+2) = (4x+4)(modulo 3) = x+1*
C+D = D+C = B	*(2x+2)+(2) = (2)+(2x+2) = (2x+4)(modulo 3) = 2x+1*
C+E = E+C = F	*(2x+2)+(2x) = (2x)+(2x+2) = (4x+2)(modulo 3) = x+2*
C+F = F+C = 1	*(2x+2)+(x+2) = (x+2)+(2x+2) = (3x+4)(modulo 3) = 1*
C+G = G+C = 0	*(2x+2)+(x+1) = (x+1)+(2x+2) = (3x+3)(modulo 3) = 0*
D+D = 1	*(2)+(2) = (4)(modulo 3) = 1*
D+E = E+D = C	*(2)+(2x) = (2x)+(2) = 2x+2*

$D+F = F+D = G$	$(2)+(x+2) = (x+2)+(2) = (x+4)(modulo\ 3) = x+1$
$D+G = G+D = A$	$(2)+(x+1) = (x+1)+(2) = (x+3)(modulo\ 3) = x$
$E+E = A$	$(2x)+(2x) = (4x)(modulo\ 3) = x$
$E+F = F+E = D$	$(2x)+(x+2) = (x+2)+(2x) = (3x+2)(modulo\ 3) = 2$
$E+G = G+E = 1$	$(2x)+(x+1) = (x+1)+(2x) = (3x+1)(modulo\ 3) = 1$
$F+F = B$	$(x+2)+(x+2) = (2x+4)(modulo\ 3) = 2x+1$
$F+G = G+F = E$	$(x+2)+(x+1) = (x+1)+(x+2) = (2x+3)(modulo\ 3) = 2x$
$G+G = C$	$(x+1)+(x+1) = 2x+2$

Table 2-8. Addition operation over GF(3^2) as polynomial representation

as an assigned symbol	as a power of the primitive element t
$0 \cdot \alpha = \alpha \cdot 0 = 0$	Note: α represents any element from GF(2^2)
$1 \cdot \alpha = \alpha \cdot 1 = \alpha$	Note: α represents any element from GF(2^2)
$A \cdot A = B$	$(t) \cdot (t) = t^2$
$A \cdot B = B \cdot A = C$	$(t) \cdot (t^2) = (t^2) \cdot (t) = t^3$
$A \cdot C = C \cdot A = D$	$(t) \cdot (t^3) = (t^3) \cdot (t) = t^4$
$A \cdot D = D \cdot A = E$	$(t) \cdot (t^4) = (t^4) \cdot (t) = t^5$
$A \cdot E = E \cdot A = F$	$(t) \cdot (t^5) = (t^5) \cdot (t) = t^6$
$A \cdot F = F \cdot A = G$	$(t) \cdot (t^6) = (t^6) \cdot (t) = t^7$
$A \cdot G = G \cdot A = 1$	$(t) \cdot (t^7) = (t^7) \cdot (t) = t^8 = 1$
$B \cdot B = D$	$(t^2) \cdot (t^2) = t^4$
$B \cdot C = C \cdot B = E$	$(t^2) \cdot (t^3) = (t^3) \cdot (t^2) = t^5$
$B \cdot D = D \cdot B = F$	$(t^2) \cdot (t^4) = (t^4) \cdot (t^2) = t^6$
$B \cdot E = E \cdot B = G$	$(t^2) \cdot (t^5) = (t^5) \cdot (t^2) = t^7$
$B \cdot F = F \cdot B = 1$	$(t^2) \cdot (t^6) = (t^6) \cdot (t^2) = t^8 = 1$
$B \cdot G = G \cdot B = A$	$(t^2) \cdot (t^7) = (t^7) \cdot (t^2) = t^9 = t^8 \cdot t = 1 \cdot t = t$
$C \cdot C = F$	$(t^3) \cdot (t^3) = t^6$
$C \cdot D = D \cdot C = G$	$(t^3) \cdot (t^4) = (t^4) \cdot (t^3) = t^7$
$C \cdot E = E \cdot C = 1$	$(t^3) \cdot (t^5) = (t^5) \cdot (t^3) = t^8 = 1$
$C \cdot F = F \cdot C = A$	$(t^3) \cdot (t^6) = (t^6) \cdot (t^3) = t^9 = t^8 \cdot t = 1 \cdot t = t$
$C \cdot G = G \cdot C = B$	$(t^3) \cdot (t^7) = (t^7) \cdot (t^3) = t^{10} = t^9 \cdot t = t \cdot t = t^2$
$D \cdot D = 1$	$(t^4) \cdot (t^4) = t^8 = 1$
$D \cdot E = E \cdot D = A$	$(t^4) \cdot (t^5) = (t^5) \cdot (t^4) = t^9 = t^8 \cdot t = 1 \cdot t = t$
$D \cdot F = F \cdot D = B$	$(t^4) \cdot (t^6) = (t^6) \cdot (t^4) = t^{10} = t^9 \cdot t = t \cdot t = t^2$
$D \cdot G = G \cdot D = C$	$(t^4) \cdot (t^7) = (t^7) \cdot (t^4) = t^{11} = t^{10} \cdot t = t^2 \cdot t = t^3$
$E \cdot E = B$	$(t^5) \cdot (t^5) = t^{10} = t^8 \cdot t^2 = 1 \cdot t^2 = t^2$
$E \cdot F = F \cdot E = C$	$(t^5) \cdot (t^6) = (t^6) \cdot (t^5) = t^{11} = t^{10} \cdot t = t^2 \cdot t = t^3$
$E \cdot G = G \cdot E = D$	$(t^5) \cdot (t^7) = (t^7) \cdot (t^5) = t^{12} = t^{11} \cdot t = t^3 \cdot t = t^4$
$F \cdot F = D$	$(t^6) \cdot (t^6) = t^{12} = t^8 \cdot t^4 = 1 \cdot t^4 = t^4$
$F \cdot G = G \cdot F = E$	$(t^6) \cdot (t^7) = (t^7) \cdot (t^6) = t^{13} = t^{12} \cdot t = t^4 \cdot t = t^5$
$G \cdot G = F$	$(t^7) \cdot (t^7) = t^{14} = t^8 \cdot t^6 = 1 \cdot t^6 = t^6$

Table 2-9. Multiplication operation over GF(3^2) as power of t representation

Both operations of addition and multiplication over $GF(2^3)$, described in Table 2-8 and respectively Table 2-9, are synthesized in Figure 2-6 where, for more readability, the symbols *{0, 1, A, B, C, D, E, F, G}* have been replaced by the natural numbers *{0, 1, 2, 3, 4, 5, 6, 7, 8}*.

+	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	1	5	3	8	7	0	4	6	2
2	2	3	6	4	1	8	0	5	7
3	3	8	4	7	5	2	1	0	6
4	4	7	1	5	8	6	3	2	0
5	5	0	8	2	6	1	7	4	3
6	6	4	0	1	3	7	2	8	5
7	7	6	5	0	2	4	8	3	1
8	8	2	7	6	0	3	5	1	4

*	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8
2	0	2	3	4	5	6	7	8	1
3	0	3	4	5	6	7	8	1	2
4	0	4	5	6	7	8	1	2	3
5	0	5	6	7	8	1	2	3	4
6	0	6	7	8	1	2	3	4	5
7	0	7	8	1	2	3	4	5	6
8	0	8	1	2	3	4	5	6	7

Figure 2-6. Addition and multiplication tables over GF(9)

2.1.4 Construction of PRS

Pseudo-random sequences (PRS) are generated by a Linear Feedback Shift Register (LFSR) whose feedback equation can be described by an irreducible polynomial *h(x)* of degree *m* with coefficients from a finite field *GF(q)*. The LFSR has a cycling behavior and outputs a fixed size symbol string formed from the elements of the finite field *GF(q)*. The initial state information of the LFSR is called the *generator seed* [Mor97].

For any given irreducible polynomial *h(x)* of degree *m* with coefficients from *GF(q)* a linear feedback shift register can be constructed as follows:

- the number of memory cell is equal to *m*, where the leftmost memory cell corresponds to the leading polynomial term and the rightmost memory cell corresponds to the least significant polynomial term;
- each memory cell contains an element of the finite field *GF(q)*;
- at each time unit the contents of the memory cells are shifted one place to the right towards the least significant polynomial term;

- the content of the leftmost memory cell is the sum of the terms corresponding to $h(x)$, where all the operations are done in $GF(q)$.

There is a direct one-to-one mapping between the terms of an irreducible polynomial $h(x)$ that implements a LFSR and the feedback equation of that LFSR.

Considering an irreducible polynomial $h(x) = x^m + h_{m-1} \cdot x^{m-1} + h_{m-2} \cdot x^{m-2} + \ldots + h_2 \cdot x^2 + h_1 \cdot x + h_0$, where $h_i \in GF(q)$ and $h_0 \neq 0$, the feedback equation of the LFSR is given by the condition $h(x) = 0$ that is the same with $x^m = -h_{m-1} \cdot x^{m-1} - h_{m-2} \cdot x^{n-2} - \ldots - h_2 \cdot x^2 - h_1 \cdot x - h_0$.
From the above condition the following recursion equation can be constructed

$$a_{k+m} = -h_{m-1} \cdot a_{k+m-1} - h_{m-2} \cdot a_{k+m-2} - \ldots - h_2 \cdot a_{k+2} - h_1 \cdot a_{k+1} - h_0 \cdot a_k$$

that represents actually the feedback equation for the LFSR.

Figure 2-7 shows the generic representation of a LFSR implementing the irreducible polynomial $h(x) = x^m + h_{m-1} \cdot x^{m-1} + h_{m-2} \cdot x^{m-2} + \ldots + h_2 \cdot x^2 + h_1 \cdot x + h_0$.

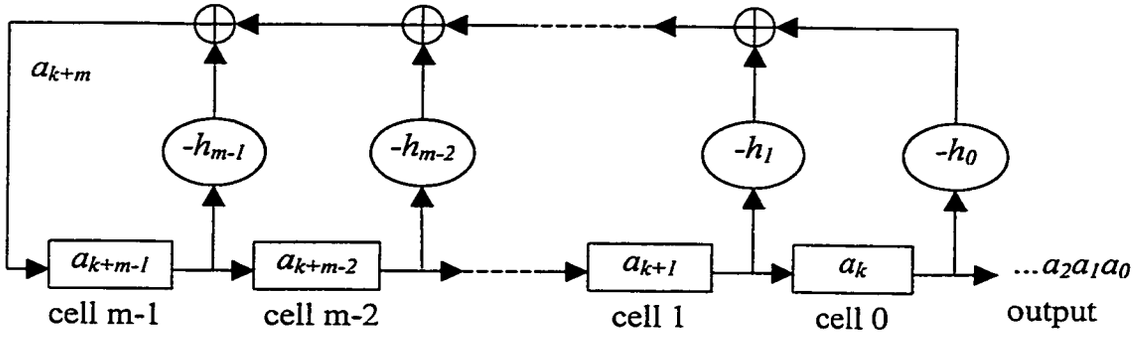


Figure 2-7. Feedback shift register for GF(q) specified by h(x)

In the binary finite field $GF(2)$ a maximal-length binary LFSR produces the maximum number of Pseudo-Random Binary Sequences (PRBSs) possible and has a pattern count equal to $2^m - 1$, where m is the number of memory cells in the LFSR and it is equal with

the degree of the irreducible polynomial $h(x)$. It produces patterns that have an approximately equal number of *1*s and *0*s and have an equal number of runs of *1*s and *0*s.

In the case of the general finite field *GF(q)*, where q is prime or power of prime, a pseudo-random sequence is generated by a m-position LFSR with a feedback path specified by a primitive polynomial of degree m with coefficients from the Galois field *GF(q)*. This sequence is called Pseudo-Random Multi-Valued Sequence (PRMVS) because it has multi-valued entries taken from the alphabet of q symbols of the *GF(q)*, as opposed to PRBS that has entries taken from the alphabet with two symbols defined on the binary space *GF(2)*. Eventually *GF(2)* is a particular case of *GF(q)* for $q=2$.

In conclusion, in order to generate Pseudo-Random Sequences (PRSs) over a finite field *GF(q)* it is necessary to have the arithmetic operations (described by the addition and multiplication tables) defined on that field along with an irreducible polynomial $h(x)$ of degree m used to implement the LFSR that will generate the PRS. The LFSR can generate a PRS of a maximum length $n = q^m-1$.

Following, there are some examples how to generate PRSs over finite fields. For the sake of space, the examples will be given only for the finite fields *GF(2)*, *GF(3)*, *GF(4)*, and *GF(5)*; which have been defined in the previous subchapters 2.1.2 respectively 2.1.3.

Generating PRS over GF(2)

Construct the PRS generated by the irreducible polynomial $h(x) = x^4+x+1$ defined over the finite field *GF(2)* whose operation tables are presented in Figure 2-1.

The feedback equation for the LFSR that will generate the PRS results from the condition

$$h(x) = 0$$

that is the same as

$$x^4 = -x-1$$
$$= x+1$$

because $x+x = 0$ *(modulo 2)* $\Leftrightarrow x = -x$ for all $x \in GF(2)$.

The 4-cell LFSR will generate a PRS of maximum length $n = 2^4-1 = 15$, has the feedback equation represented by $a_{k+4} = a_{k+1} + a_k$, and the block diagram represented in Figure 2-8.

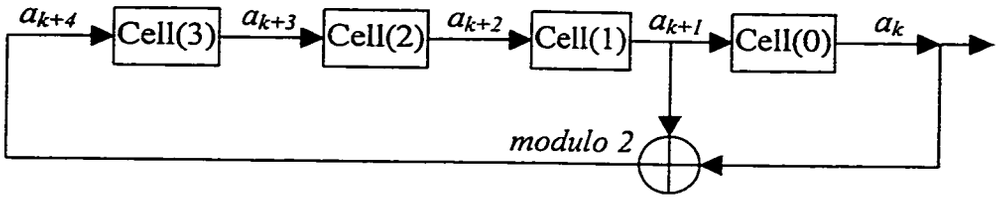


Figure 2-8. 4-cell LFSR implementing $h(x) = x^4+x+1$ on GF(2)

The states of the LFSR cells at each position k in time are shown in the Table 2-10.

Clock	Cell(3)	Cell(2)	Cell(1)	Cell(0)	Comments
0	*1*	*0*	*0*	*0*	Seed value
1	*0*	*1*	*0*	*0*	
2	*0*	*0*	*1*	*0*	
3	*1*	*0*	*0*	*1*	
4	*1*	*1*	*0*	*0*	
5	*0*	*1*	*1*	*0*	
6	*1*	*0*	*1*	*1*	
7	*0*	*1*	*0*	*1*	
8	*1*	*0*	*1*	*0*	
9	*1*	*1*	*0*	*1*	
10	*1*	*1*	*1*	*0*	
11	*1*	*1*	*1*	*1*	
12	*0*	*1*	*1*	*1*	
13	*0*	*0*	*1*	*1*	
14	*0*	*0*	*0*	*1*	
15	*1*	*0*	*0*	*0*	The cycle is repeated
16	*0*	*1*	*0*	*0*	

Table 2-10. PRS generated by $h(x) = x^4+x+1$ on GF(2)

It can be seen, from the Table 2-10, that the output of Cell(0) of the 4-bit LFSR represented in Figure 2-8 is the cyclic PRBS:

$$\ldots 000100110101111\ldots$$

of maximal length $n = 2^4\text{-}1 = 15$.

Generating PRS over GF(3)

Construct the PRS generated by the irreducible polynomial $h(x) = x^3+x+2$ defined over the finite field $GF(3)$ whose operation tables are presented in Figure 2-2.

The feedback equation for the LFSR that will generate the PRS results from the condition

$$h(x) = 0$$

that is the same as

$$x^3 = -x-2$$
$$= 2x+1$$

because $2x+x = 0$ *(modulo 3)* $\Leftrightarrow$ $2x = -x$ *and* $1+2 = 0$ *(modulo 3)* $\Leftrightarrow$ $1 = -2$ in $GF(3)$.

The 3-cell LFSR will generate a PRS of length $n = 3^3\text{-}1 = 26$, has the feedback equation represented by $a_{k+3} = 2a_{k+1} + a_k$, and the block diagram represented in Figure 2-9.

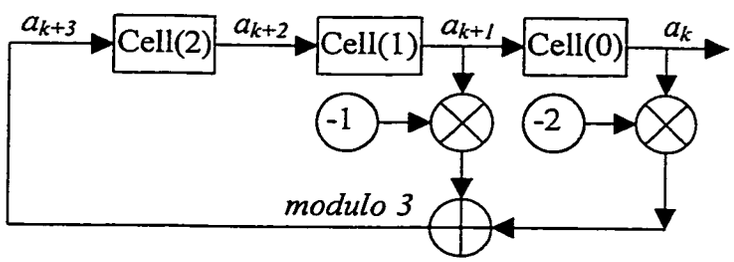


Figure 2-9. 3-cell LFSR implementing $h(x) = x^3+x+2$ on GF(3)

The states of the LFSR cells at each position k in time are shown in the Table 2-11.

It can be seen, from the Table 2-11, that the output of Cell(0) of the 3-bit LFSR represented in Figure 2-9 is the cyclic PRMVS:

$$\ldots 001012112011100202122102 22\ldots$$

of maximal length $n = 3^3-1 = 26$.

Clock	Cell(2)	Cell(1)	Cell(0)	Comments
0	*1*	*0*	***0***	Seed value
1	*0*	*1*	***0***	
2	*1*	*0*	***1***	
3	*2*	*1*	***0***	
4	*1*	*2*	***1***	
5	*1*	*1*	***2***	
6	*2*	*1*	***1***	
7	*0*	*2*	***1***	
8	*1*	*0*	***2***	
9	*1*	*1*	***0***	
10	*1*	*1*	***1***	
11	*0*	*1*	***1***	
12	*0*	*0*	***1***	
13	*2*	*0*	***0***	
14	*0*	*2*	***0***	
15	*2*	*0*	***2***	
16	*1*	*2*	***0***	
17	*2*	*1*	***2***	
18	*2*	*2*	***1***	
19	*1*	*2*	***2***	
20	*0*	*1*	***2***	
21	*2*	*0*	***1***	
22	*2*	*2*	***0***	
23	*2*	*2*	***2***	
24	*0*	*2*	***2***	
25	*0*	*0*	***2***	
26	*1*	*0*	***0***	The cycle is repeated
27	*0*	*1*	***0***	
28	*1*	*0*	***1***	

Table 2-11. PRS generated by $h(x) = x^3+x+2$ on GF(3)

Generating PRS over GF(4)

Construct the PRS generated by the irreducible polynomial $h(x) = x^3+3x^2+3x+2$ defined over the finite field *GF(4)* whose operation tables are presented in Figure 2-4.

The LFSR that will generate the PRS has 3 cells and the feedback equation of this LFSR results from the condition

$$h(x) = 0$$

that is the same as

$$x^3 = -3x^2-3x-2$$
$$= x^2+x+2$$

because $x^2+3x^2 = 0$ *(modulo 4)* $\Leftrightarrow x^2 = -3x^2$, $x+3x = 0$ *(modulo 4)* $\Leftrightarrow x = -3x$ *and* $2+2 = 0$ *(modulo 4)* $\Leftrightarrow 2 = -2$ in *GF(4)*.

The 3-cell LFSR generates a PRS of length $n = 4^3-1 = 63$, has the feedback equation represented by $a_{k+3} = a_{k+2} + a_{k+1} + 2a_k$, and the block diagram represented in Figure 2-10.

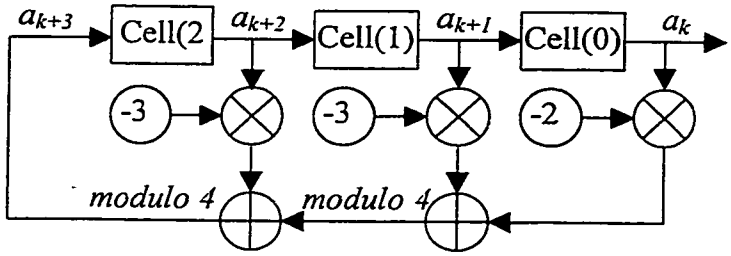


Figure 2-10. 3-cell LFSR implementing $h(x) = x^3+3x^2+x+2$ on GF(4)

The states of the LFSR cells at each position *k* in time are shown in the Table 2-12.

Clock	Cell(2)	Cell(1)	Cell(0)	Comments
0	*1*	*0*	***0***	Seed value
1	*1*	*1*	***0***	
2	*0*	*1*	***1***	
3	*3*	*0*	***1***	
4	*1*	*3*	***0***	
5	*2*	*1*	***3***	
6	*2*	*2*	***1***	
7	*2*	*2*	***2***	
8	*3*	*2*	***2***	
9	*2*	*3*	***2***	
10	*2*	*2*	***3***	

11	*1*	*2*	***2***	
12	*0*	*1*	***2***	
13	*2*	*0*	***1***	
14	*0*	*2*	***0***	
15	*2*	*0*	***2***	
16	*1*	*2*	***0***	
17	*3*	*1*	***2***	
18	*1*	*3*	***1***	
19	*0*	*1*	***3***	
20	*0*	*0*	***1***	
21	*2*	*0*	***0***	
22	*2*	*2*	***0***	
23	*0*	*2*	***2***	
24	*1*	*0*	***2***	
25	*2*	*1*	***0***	
26	*3*	*2*	***1***	
27	*3*	*3*	***2***	
28	*3*	*3*	***3***	
29	*1*	*3*	***3***	
30	*3*	*1*	***3***	
31	*3*	*3*	***1***	
32	*2*	*3*	***3***	
33	*0*	*2*	***3***	
34	*3*	*0*	***2***	
35	*0*	*3*	***0***	
36	*3*	*0*	***3***	
37	*2*	*3*	***0***	
38	*1*	*2*	***3***	
39	*2*	*1*	***2***	
40	*0*	*2*	***1***	
41	*0*	*0*	***2***	
42	*3*	*0*	***0***	
43	*3*	*3*	***0***	
44	*0*	*3*	***3***	
45	*2*	*0*	***3***	
46	*3*	*2*	***0***	
47	*1*	*3*	***2***	
48	*1*	*1*	***3***	
49	*1*	*1*	***1***	
50	*2*	*1*	***1***	
51	*1*	*2*	***1***	
52	*1*	*1*	***2***	
53	*3*	*1*	***1***	
54	*0*	*3*	***1***	
55	*1*	*0*	***3***	

56	0	1	**0**	
57	1	0	**1**	
58	3	1	**0**	
59	2	3	**1**	
60	3	2	**3**	
61	0	3	**2**	
62	0	0	**3**	
63	1	0	**0**	The cycle is repeated
64	1	1	**0**	
65	0	1	**1**	

Table 2-12. PRS generated by $h(x) = x^3+3x^2+3x+2$ on GF(4)

It can be seen, from the Table 2-12, that the output of Cell(0) of the 3-bit LFSR represented in Figure 2-10 is the cyclic PRMVS:

...00110312223221020213100220123331332030321200330231112113010132 3...

of maximal length $n = 4^3-1 = 63$.

Generating PRS over GF(5)

Construct the PRS generated by the irreducible polynomial $h(x) = x^2+2x+3$ defined over the finite field *GF(5)* whose operation tables are presented in Figure 2-3.

The LFSR that will generate the PRS has 2 cells and the feedback equation of this LFSR results from the condition

$$h(x) = 0$$

that is the same as

$$x^2 = -2x-3$$
$$= 3x+2$$

because $3x+2x = 0$ *(modulo 5)* $\Leftrightarrow 3x = -2x$ and $2+3 = 0$ *(modulo 3)* $\Leftrightarrow 2 = -3$ in *GF(5)*.

The 2-cell LFSR generates a PRS of length $n = 5^2-1 = 24$, has the feedback equation represented by $a_{k+2} = 3a_{k+1} + 2a_k$ and the block diagram represented in Figure 2-11.

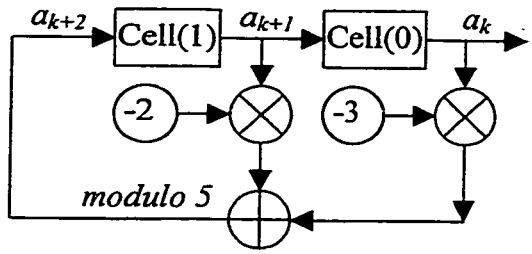


Figure 2-11. 3-cell LFSR implementing $h(x) = x^2+2x+3$ on GF(5)

The states of the LFSR cells at each position *k* in time are shown in the Table 2-13.

Clock	Cell(1)	Cell(0)	Comments
0	*1*	***0***	Seed value
1	*3*	***1***	
2	*1*	***3***	
3	*4*	***1***	
4	*4*	***4***	
5	*0*	***4***	
6	*3*	***0***	
7	*4*	***3***	
8	*3*	***4***	
9	*2*	***3***	
10	*2*	***2***	
11	*0*	***2***	
12	*4*	***0***	
13	*2*	***4***	
14	*4*	***2***	
15	*1*	***4***	
16	*1*	***1***	
17	*0*	***1***	
18	*2*	***0***	
19	*1*	***2***	
20	*2*	***1***	
21	*3*	***2***	
22	*3*	***3***	
23	*0*	***3***	
24	*1*	***0***	The cycle is repeated
25	*3*	***1***	
26	*1*	***3***	

Table 2-13. PRS generated by $h(x) = x^2+2x+3$ on GF(5)

It can be seen, from the Table 2-13, that the output of Cell(0) of the 2-bit LFSR represented in Figure 2-11 is the cyclic PRMVS:

$$\ldots 0\,1\,3\,1\,4\,4\,0\,3\,4\,3\,2\,2\,0\,4\,2\,4\,1\,1\,0\,2\,1\,2\,3\,3\ldots$$

of maximal length $n = 5^2-1 = 24$.

2.1.5 Some Useful PRS Properties

In their paper [MaS76] MacWilliams and Sloane give a comprehensive description of the properties of both pseudo-random sequences and arrays. Following there is a brief presentation of some of these properties related to the PRS and also used extensively by the PRS decoding mechanism, which is actually the purpose of this work.

Let $h(x)=\sum_{i=0}^{m} h_i \cdot x^i$ be a fixed primitive polynomial of degree m with coefficients from $GF(p)$, and let $\delta_m(q)$, where q is a prime of p, be the set consisting of the PRS obtained from $h(x)$ together with the 0 sequence.

1 – Shift Property: For any PRS $a = a_0a_1\ldots a_{q-2}$ in $\delta_m(q)$ a cyclic shift of this PRS $a_ja_{j+1}\ldots a_{q-2}a_0a_1\ldots a_{j-1}$ is also in $\delta_m(q)$.

2 – Recurrence Property: Any PRS $a \in \delta_m(q)$ satisfies the recurrence $a_{i+m} = h_{m-1}\cdot a_{i+m-1} + h_{m-2}\cdot a_{i+m-2} +\ldots + h_1\cdot a_{i+1} + h_0\cdot a_i$, where $i = 0, 1, 2, \ldots$

3 – Window Property: If a window of width m, where m is the degree of the irreducible polynomial, is slid along a PRS in $\delta_m(q)$ each of the p^m-1 non-zero m-tuples is seen exactly once.

Because this property plays a major role in developing the decoding algorithm an example is given next.

Example

The 3-bit LFSR that implements the primitive polynomial $h(x)=x^3+x+1$ of degree $m=3$ on the binary finite field $GF(2)$ generates the following PRBS $0\ 0\ 1\ 0\ 1\ 1\ 1$. A window of width $m=3$ when slid along the PRBS generated by this LFSR parses each of the $p^m-1 = 2^3-1 = 7$ non-zero triplets exactly once. To avoid difficulties at the ends imagine that the sequence is written in a circle as shown in Figure 2-12.

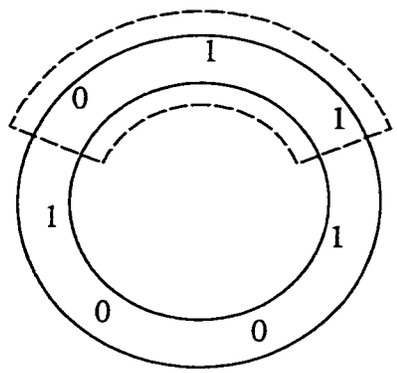


Figure 2-12. 3-bit window slid on a continuous 7-bit PRBS

4 – Pseudo-Random Property: In any PRS belonging to $\delta_m(q)$ the element 0 occurs $q^{m-1}-1$ times and every non-zero element of $GF(q)$ occurs q^m-1 times.

5 – Addition Property: The sum of two sequences in $\delta_m(q)$, done modulo q without carries, is another sequence in $\delta_m(q)$.

6 – Shift-and-Add Property: The sum between a PRS in $\delta_m(q)$ and a cyclic shift of itself is another PRS in $\delta_m(q)$.

Besides the PRS properties presented above the PRS have many others which are described in detail in the literature [MaS76].

2.2 Pseudo-Random to Natural Code Conversion Algorithm

Pseudo-random decoding is based on the PRS window property. The PRS is a generic term used to identify both the PRBS and PRMVS, whose symbols are elements from *GF(2)* and respectively *GF(q)*. It should be noted that the PRBS are a particular case of PRMVS when $q=2$.

A PRMVS has multi-valued entries taken from an alphabet of q symbols, where q is prime or power of prime. Such a sequence is generated by a m-cell LFSR whose feedback mechanism is specified by a primitive polynomial of degree m with coefficients from the Galois field *GF(q)*. Based on the PRMVS window property, which says that any q-valued contents observed through a m-position window sliding over the PRMVS is unique, it can be safely assumed that the contents of the window actually fully identifies the current position of the window. This assumption is also called in literature the Pseudo-Random Window Index Recovery.

Let a LFSR be defined by a primitive polynomial *h(x)* of degree m with coefficients from *GF(q)*, where q is prime or power of prime, that generates the PRS $a_0a_1a_2...a_{n-3}a_{n-2}$ containing $n=q^m-1$ symbols. Considering a m-size window belonging to this sequence slid over the sequence, where a_k is considered the origin of the window, the pseudo-random window index recovery can be stated as:

> *Given the elements of an m-size window belonging to a PRS generated by h(x) over GF(q) determine the k index of the origin of the window* a_k.

In other words, knowing the initial state a_0 of the LFSR, the index recovery problem requires finding the natural representation of the index k of the origin of the window with respect to a_0, as shown in Figure 2-13.

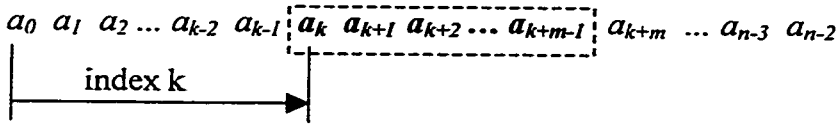
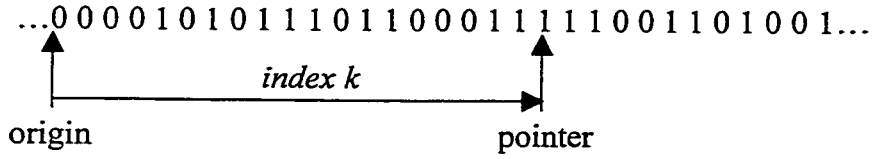


Figure 2-13. Window index recovery for a generic PRS

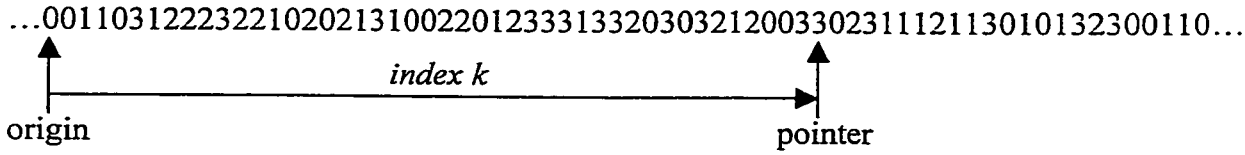
Example 1. Window index recovery for PRBS

A 5-cell LFSR that implements the irreducible polynomial $h(x) = x^5+x^3+1$ defined over the finite field *GF(2)* generates the following PRBS:



Example 2. Window index recovery for PRMVS

A 3-cell LFSR that implements the irreducible polynomial $h(x)=x^3+3x^2+3x+2$ defined over the finite field *GF(4)* generates the following PRMVS:



If the left most symbol of the sequence is considered the origin of the sequence then what is the distance k from the pointer to the origin?

In each case the solution is based on the window property and it is easy to verify that any content seen through a 5-cell window sliding over the PRBS (in Example 1) respectively a 3-cell window sliding over the PRMVS (in Example 2) is unique.

There are basically three methods, mentioned in literature, used to implement a mechanism for PRS decoding:

- a serial-type code conversion mechanism;
- a parallel-type code conversion mechanism;
- a mixed serial-parallel-type code conversion mechanism;

Following, a brief account will be given on these three methods used to implement a PRS decoding mechanism together with a solution to the above stated examples using each of the methods.

2.2.1 Serial-Type Code Conversion

The serial-type code conversion mechanism is simply based on counting the number k of feedback shifts to get back to the seed element a_0 and, it was extensively presented in [Pe87] for *GF(2)* sequences.

This concept could be generalized as follows. Knowing that the PRS is cyclic, the pseudo-random-to-natural code conversion can be obtained by counting the feedback shifts forward to get to the next seed element a_0, as illustrated in Figure 2-14, where the forward counted feedback shifts are represented by the *index j*.

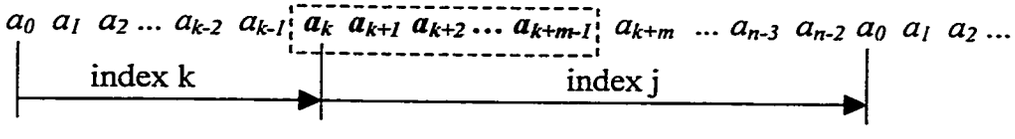


Figure 2-14. Window index recovery for a cyclic PRS

Figure 2-15 shows the block diagram of the serial-type code conversion mechanism.

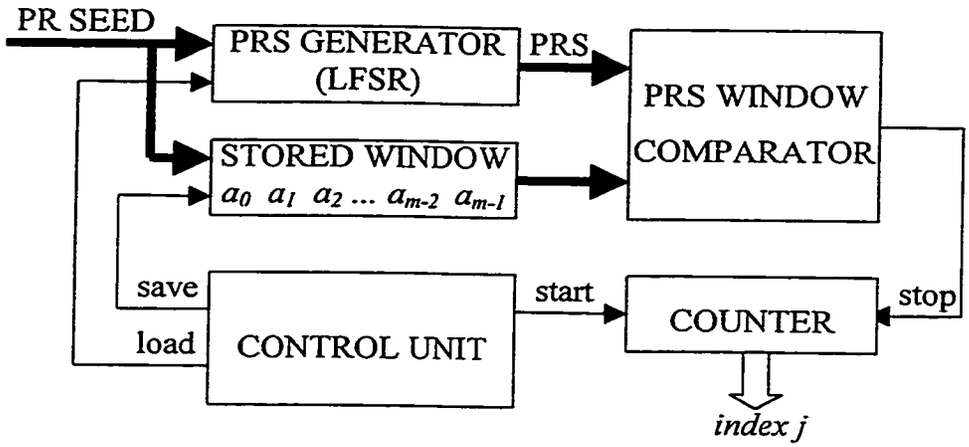


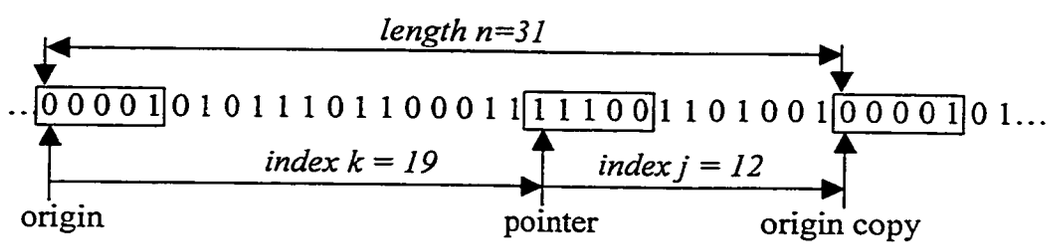
Figure 2-15. Block diagram of the serial-type code conversion

A LFSR defined by a primitive polynomial $h(x)$ of degree m with coefficients from $GF(q)$, where q is prime or power of prime, generates a PRS having $n=q^m-1$ symbols. If the forward counting index is denoted by j then the index k from the seed element can be calculated using the formula:

$$k = n - j = q^m - 1 - j.$$

Example 1. Serial-Type Code Conversion of PRBS

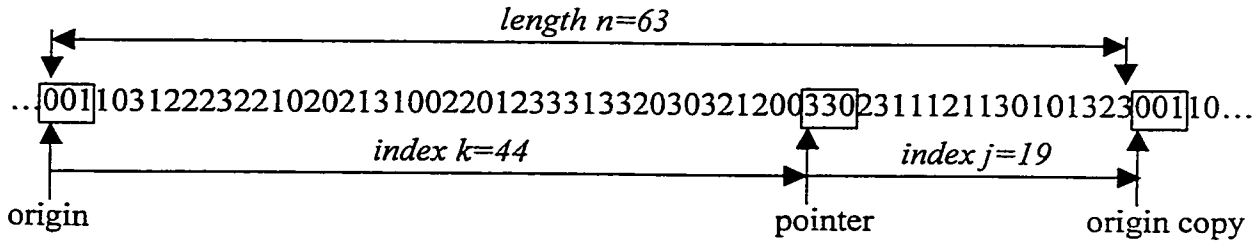
A 5-cell LFSR implements the irreducible polynomial $h(x) = x^5+x^3+1$ of degree $m=5$ defined over the finite field $GF(2)$ that has $q=2$ elements. The length of the PRBS generated by the LFSR implementing this arithmetic is $n = q^m-1 = 2^5-1 = 31$. The width of the PRBS window is equal with the degree of $h(x)$.



Because any 5-symbol window is unique, by saving the origin window (000010) the PRBS decoding mechanism is aware of the reference point the measurement is referred to. Next the LFSR starts generating the PRS sequence that has the pointer window (11100) as PRBS seed. At the same time the counter keeps track of the index j counting every generated PRBS symbol. When the current PRBS window matches the stored origin window the decoding process stops and the index $j=12$. The distance from pointer to the origin is given by the index k that is calculated according to the formula $k=n-j=31-12=19$. So, the distance from the origin to the pointer is 19 symbols.

Example 2. Serial-Type Code Conversion of PRMVS

A 3-cell LFSR implements the irreducible polynomial $h(x)=x^3+3x^2+3x+2$ of degree $m=3$ defined over the finite field $GF(4)$ that has $q=4$ elements. The length of the PRMVS generated by the LFSR implementing this arithmetic is $n = q^m-1 = 4^3-1 = 63$. The width of the PRMVS window is equal with the degree of $h(x)$.



Because any 3-symbol window is unique, by saving the origin window (001) the PRMVS decoding mechanism is aware of the reference point the measurement is referred to. Next the LFSR starts generating the PRMVS sequence that has the pointer window (330) as PRMVS seed. At the same time the counter keeps track of the index j counting every generated PRMVS symbol. When the current PRMVS window matches the stored origin window the decoding process stops and the index $j=19$. The

distance from pointer to the origin is given by the index k that is calculated according to the formula $k=n-j=63-19=44$. So, the distance from the origin to the pointer is 44 symbols.

The LFSR used by the serial-type code conversion mechanism based on forward counting is identical with the LFSR used to generate the initial PRS and its feedback equation results immediately from the generator polynomial $h(x)$.

The PRS decoding method based on the serial-type code conversion mechanism is simple to implement but it is very slow for applications using high encoding resolution due to the fact that it has to generate all the intermediate PRS symbols between the pointer symbol and the origin symbol.

2.2.2 Parallel-Type Code Conversion

The parallel-type code conversion mechanism is based strictly on storing in ROM a code conversion table for the whole PRS. This method is based on the uniqueness of the symbol window and it requires a conversion table of n entries containing successive PRS windows of size m, where $n=q^m-1$ is the number of symbols in the PRS and m is the degree of the irreducible polynomial generating the sequence.

Then the current PRS is compared simultaneously against the n stored PRS window entries and the distance to the reference point is ready available on matching by reading the index k of the ringing comparator.

Figure 2-16 shows the block diagram of the parallel-type code conversion mechanism.

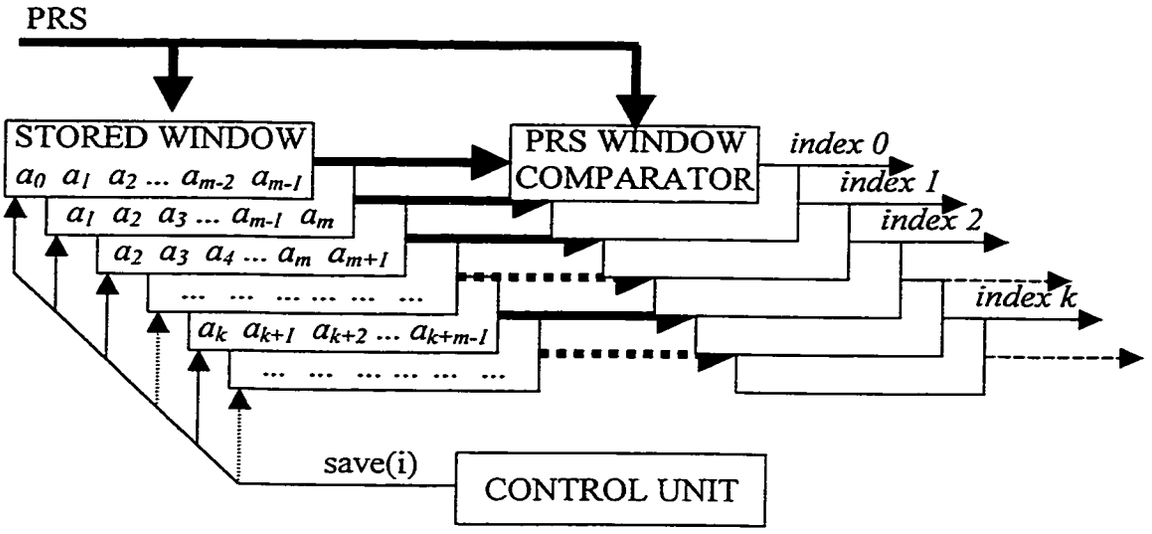
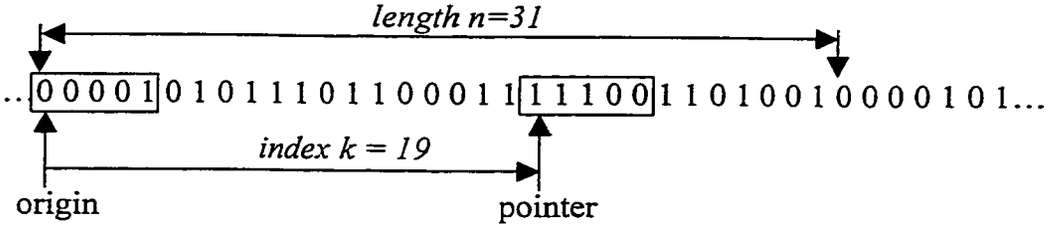


Figure 2-16. Block diagram of the parallel-type code conversion

Example 1. Parallel-Type Code Conversion of PRBS

A 5-cell LFSR implements the irreducible polynomial $h(x) = x^5+x^3+1$ of degree $m=5$ defined over the finite field $GF(2)$ that has $q=2$ elements. The length of the PRBS generated by the LFSR implementing this arithmetic is $n = q^m-1 = 2^5-1 = 31$. The width of the PRBS window is equal with the degree of $h(x)$.



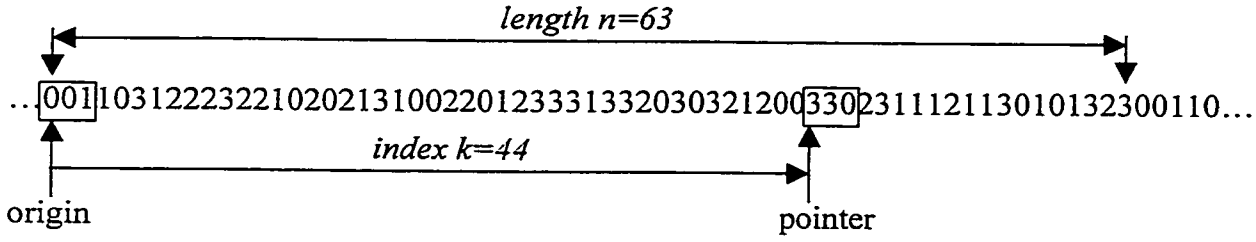
Because any 5-symbol window is unique the mechanism must store a conversion table with $n=31$ entries, each entry of size $m=5$ symbols, as shown in the following table.

Table entry	Conversion index
00001	0
00010	1
00101	2
01010	3
10101	4
...	...
11111	17
11110	18
11100	19
11001	20
...	...
10100	29
01001	30

When loading the decoder with the pointer window (11100), the comparator with the index $k=19$ will ring and the index represent the distance from the PRBS origin to the PRBS pointer, which is 19 symbols.

Example 2. Parallel-Type Code Conversion of PRMVS

A 3-cell LFSR implements the irreducible polynomial $h(x)=x^3+3x^2+3x+2$ of degree $m=3$ defined over the finite field $GF(4)$ that has $q=4$ elements. The length of the PRS generated by the LFSR implementing this arithmetic is $n = q^m-1 = 4^3-1 = 63$. The width of the PRMVS window is equal with the degree of $h(x)$.



Because any 3-symbol window is unique the mechanism must store a conversion table with $n=63$ entries, each entry of size $m=3$ symbols, as shown in the following table.

Table entry	Conversion index
001	0
011	1
110	2
103	3
031	4
312	5
122	6
222	7
...	...
003	42
033	43
330	44
302	45
...	...
230	61
300	62

Table 2-14. Conversion table for parallel-type code conversion

When loading the decoder with the pointer window (330), the comparator with the index *k=44* will ring immediately and the index represent the distance from the PRMVS origin to the PRMVS pointer, which is 44 symbols.

This method, although extremely fast, is the most expensive to implement requiring a lot of hardware resources for applications using high encoding resolution.

2.2.3 Serial-Parallel-Type Code Conversion

A compromised solution represented by a combination of both the above presented methods was developed by Petriu and presented first in his paper [Pe88-1].

This concept could be generalized as follows. The serial-parallel-type code conversion method employs a parallel-type code conversion where only selected PRS windows,

called milestones, are saved in ROM; and, also a serial-type code conversion where feedback shifts are counted forward to get to one of the saved milestones as illustrated in Figure 2-17.

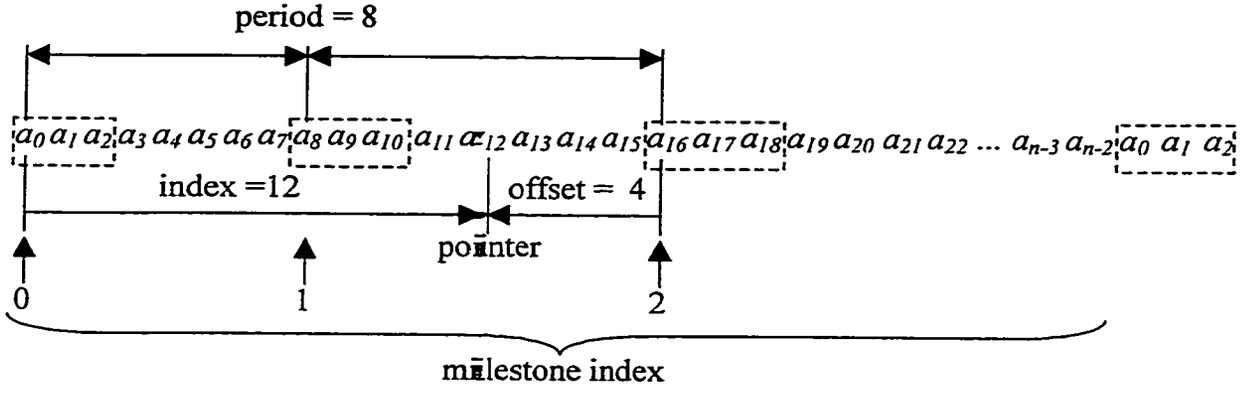


Figure 2-17. Window index recovery for serial-parallel type conversion

Let be assumed that when building the milestone ROM table the PRS was sampled periodically. Then the window index can be recovered using the following formula:

$$\textit{window index} = \begin{cases} \textit{sequence length} - \textit{offset}, & \textit{when (milestone index)} = 0 \\ \textit{period} * (\textit{milestone index}) - \textit{offset}, & \textit{when (milestone index)} > 0 \end{cases}$$

where:

- the *sequence length* is $n = q^m - 1$,
- the *period* represents the number of symbols between two successive milestones,
- the *milestone index* represents the index in the milestone ROM table, and
- the *offset* represents the number of shifts the LFSR takes to move from the current window-state to the closest milestone-state.

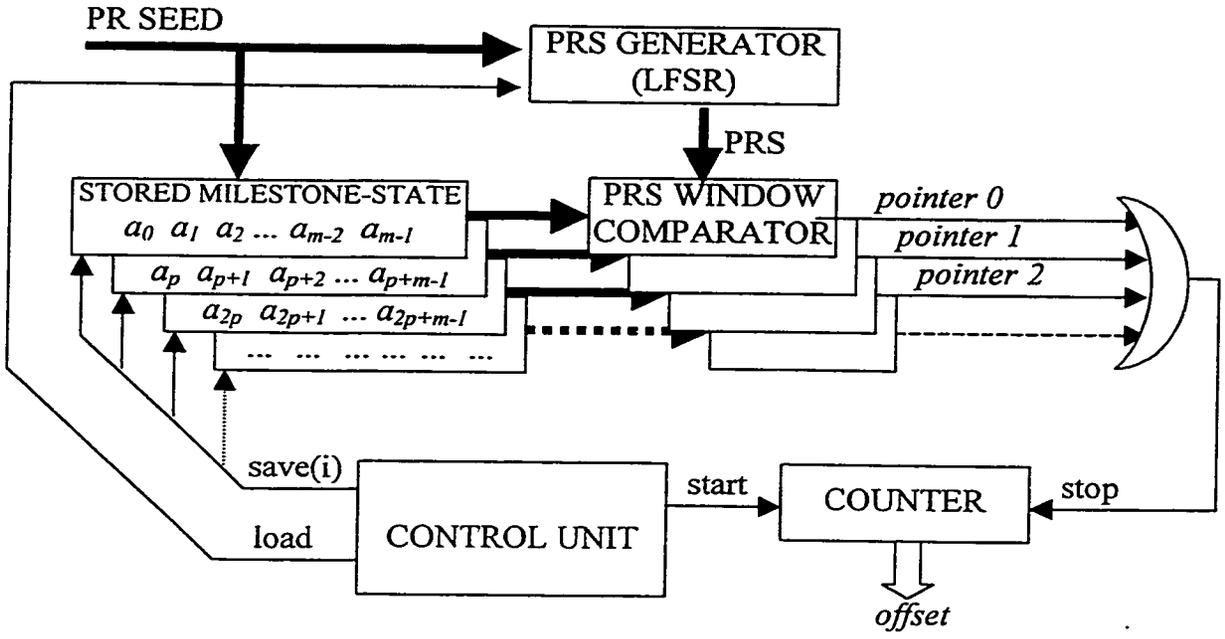
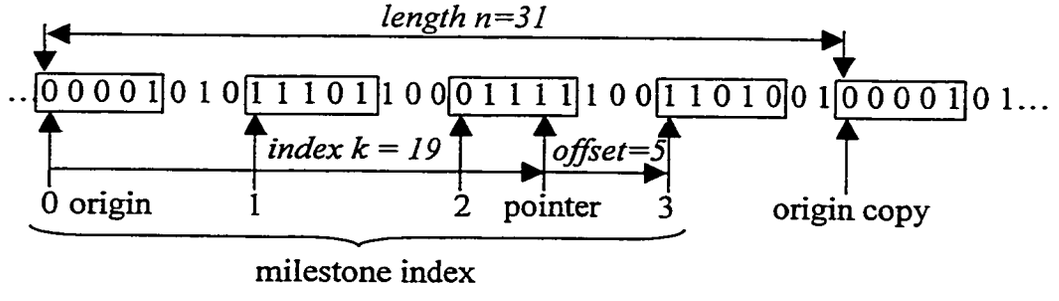


Figure 2-18. Block diagram of the serial-parallel-type code conversion

Example 1. Serial-parallel-Type Code Conversion of PRBS

A 5-cell LFSR implements the irreducible polynomial $h(x) = x^5+x^3+1$ of degree $m=5$ defined over the finite field $GF(2)$ that has $q=2$ elements. The length of the PRBS generated by the LFSR implementing this arithmetic is $n = q^m-1 = 2^5-1 = 31$. The width of the PRBS window is equal with the degree of $h(x)$.



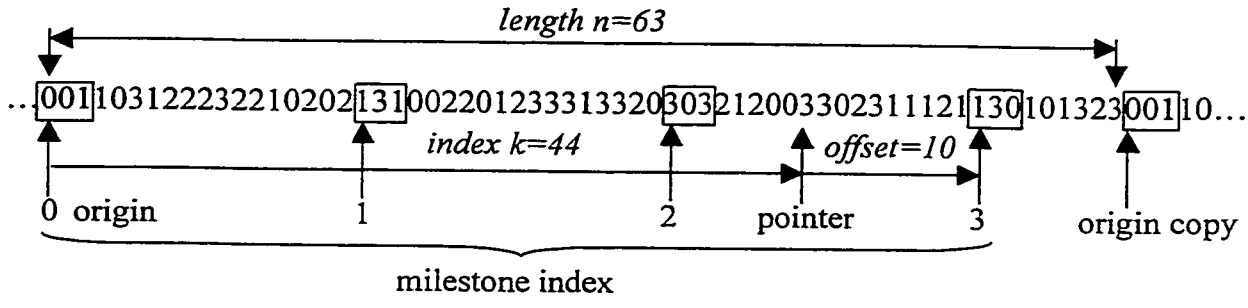
Considering that every 8^{th} 5-symbol window is sampled and stored as a milestone into the milestone database the following milestone table is obtained:

Milestone index	PRS symbol index	Milestone content
0	0	00001
1	8	11101
2	16	01111
3	24	11010

The LFSR starts generating the PRBS that has the pointer window (11100) as PRBS seed. At the same time the counter keeps track of the *offset* counting every generated PRBS symbol. When the current PRBS window matches any milestone window, in our case will be the milestone (11010) with the index 3, the decoding process stops and the counted *offset=5*. The distance from origin to pointer is given by the index *k* that is calculated according to the formula *k=period*(milestone index)-offset=8*3-5=19*. So, the distance from origin to pointer is 19 symbols.

Example 2. Serial-parallel-Type Code Conversion of PRMVS

A 3-cell LFSR implements the irreducible polynomial $h(x)=x^3+3x^2+3x+2$ of degree $m=3$ defined over the finite field *GF(4)* that has $q=4$ elements. The length of the PRMVS generated by the LFSR implementing this arithmetic is $n = q^m-1 = 4^3-1 = 63$. The width of the PRMVS window is equal with the degree of $h(x)$.



Considering that every 18^{th} 3-symbol window is sampled and stored as a milestone into the milestone database the following milestone table is obtained:

Milestone index	PRS symbol index	Milestone content
0	0	001
1	18	131
2	36	303
3	54	130

The LFSR starts generating the PRMVS that has the pointer window (330) as PRMVS seed. At the same time the counter keeps track of the *offset* counting every generated PRMVS symbol. When the current PRMVS window matches any milestone window, in our case it will be the milestone (130) with the index 3, the decoding process stops and the counted *offset=10*. The distance from origin to pointer is given by the index *k* that is calculated according to the formula *k=period*(milestone index)-offset=18*3-10=44*. So, the distance from origin to pointer is 44 symbols.

The serial-parallel-type code conversion method presents an optimum in performance versus cost due to the fact that it takes advantage of both the other code conversion methods by employing:

- a partial parallel-type code conversion when looking for milestone match; conversion which is very fast but expensive requiring the maximum hardware resources used to implement it; and,
- a partial serial-type code conversion when generating all the intermediate PRS symbols between the pointer symbol and the first milestone window, conversion which is slower but cheaper requiring the minimum hardware resources to implement it.

Due to its relatively high conversion speed, the serial-parallel-type code conversion method is suitable for applications using high encoding resolution and can be optimized for a very specific application by fine tuning the rapport between the parallelism versus serialization required by that application.

This type of code conversion algorithm, which presents the best performances among the three types of code conversion algorithms presented in the section 2.2, is chosen for physical implementation into a specialized decoding hardware as further explained in Chapter 3.

Chapter 3. HARDWARE COPROCESSOR FOR PSEUDO-RANDOM MULTI-VALUED SEQUENCES

The coprocessor, further called Pseudo-Random DECoder (PRDEC) coprocessor, implements the serial-parallel-type code conversion algorithm presented in Chapter 2 and it is designed to work in microprocessor-based embedded systems. Because of the large variety of such systems the coprocessor should be easily interfaced to various microprocessors. To satisfy this requirement the PRDEC coprocessor is designed as a memory-mapped device employing a generic 8-bit memory interface for connection to a microprocessor. This type of interface can be connected to all embedded processors that make use of an external memory. The block diagram of connecting the coprocessor to a microprocessor is shown in Figure 3-1.

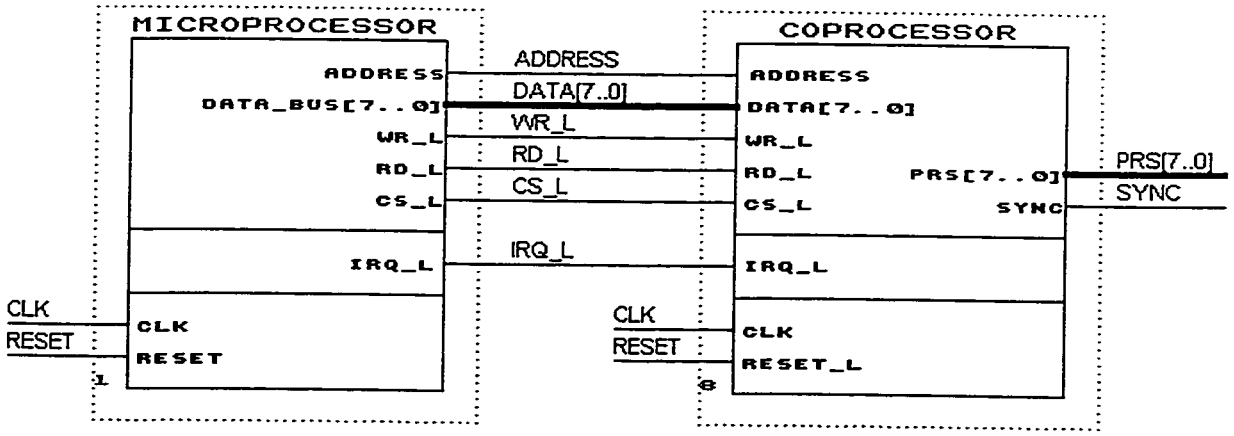


Figure 3-1. Generic memory interface

This architecture is of type master-slave where the μP is the master and the coprocessor is the slave. The μP writes to and reads from the coprocessor using the generic memory interface signals whose functionality is well known.

The interface signals are:

- address bus ADDRESS;
- data bus DATA[7..0];
- read enable RD_L;
- write enable WR_L;
- and, chip select CS_L.

The interrupt request IRQ_L signal is used by the PRDEC coprocessor to signal back to the μP about the status of its inside running processes.

The master μP initializes the PRDEC coprocessor by sending:

1. the algebra of a given finite field (addition and multiplication tables and the coefficients of the generator polynomial);
2. the milestone period;
3. the milestone seed.

After this initialization the coprocessor is able to decode any PRMVS window code received from the master returning corresponding natural-code values for the milestone-index and the milestone-offset.

The coprocessor also generates the PRS and its synchronization signal SYNC.

In order to reduce embedded system complexity some microprocessors (microcontrollers) have on-chip RAM and FLASH memory for instruction code (e.g. Motorola's HC11 family, Intel's 8x51 family, Microchip's PIC family, Hitachi's SH-1 family, etc). They often employ a multiplexed address/data bus architecture with the external world in order to reduce the pin count. For this type of microprocessors a custom interface can be developed easily, interface that will translate the signals of the specialized address/data bus into the signals of the generic memory interface used by the inside processing core.

Data to be decoded is presented to the PRDEC coprocessor as a PRMVS window. Decoding means finding the corresponding natural code associated to PRMVS window. The decoding process is presented in the subchapter 2.2. It requires the necessary algebra

and the PRS milestones to be available inside the PRDEC coprocessor at the processing time. Algebra used by the decoding process is represented by the two operations, addition and multiplication, defined on a specific Galois field and the irreducible polynomial. The pseudo-random sequence depends only on the irreducible polynomial for a specific Galois field. The Galois field specifies the symbol alphabet used to encode the sequence and also defines the addition and multiplication operations used to build that sequence. The degree and the coefficients of the irreducible polynomial determine the length of the sequence.

For a specific application the algebra and the milestone table will be hardwired into the chip in order to reduce the silicon resources and to speed up the processing. There have been attempts in literature to implement the Galois field operations, addition and multiplication, as true hardware adders and multipliers [Paa96, PaP98] but they usually are implemented over fixed finite fields of order 2 or power of 2. Being hardwired, these implementations are very fast and consume few silicon resources but they lack the flexibility offered by a generic microprogrammed implementation.

For a generalized type of applications the algebra downloaded into the coprocessor defines the GF employed by the application, the parameters of the irreducible polynomial, and the PRS milestone period. Such a generic programmable PRDEC coprocessor will use a larger amount of silicon resources then a specialized one. It will also work at a lower speed. However the generic PRDEC coprocessor could be used for decoding a multitude of PRMVSs while a specialized one is restricted to only one PRMVS.

The pros and cons for the two implementations of the PRDEC coprocessor are best described by the classical hardware paradigm of universal programmable processor versus hardwired application-specific processor (ASIC).

3.1 Coprocessor Design

The generic PRDEC coprocessor requires a initialization phase when specific operational parameters are programmed. These operational parameters are:

- the multiplication and addition tables of the Galois field,
- the degree and the coefficients of the irreducible polynomial,
- the milestone seed and the milestone period desired.

By using RAM to hold the operational parameters the PRDEC coprocessor presents the most flexible structure that can process a large variety of PRS.

The registers of the PRDEC coprocessor are assigned to the memory space of the microprocessor according to Table 3-1.

Address	Operation	Register Implementation
0	1st WRITE	Number *(q-1)* of Alphabet Symbols corresponding to *GF(q)*
	next *qxq* WRITEs	Addition Table as RAM
	next *qxq* WRITEs	Multiplication Table as RAM
	next WRITE	Degree *(m-1)* of the Irreducible Polynomial *h(x)*
	next *m-1* WRITEs	Irreducible Polynomial Coefficient Table as FIFO memory
	next WRITE	Milestone Period *(p-1)*
	next *m* WRITEs	Milestone Seed
	READ	Status Register
1	*m* WRITEs	Sequence Window as FIFO memory
	1st READ	Natural Code corresponding to milestone cluster
	2nd READ	Natural Code corresponding to milestone index
	3rd READ	Natural Code corresponding to milestone offset

Table 3-1. PRDEC coprocessor register mapping

It can be seen that the coprocessor registers are split in two sections:

- the setting register section implemented as successive writes to the address 0, and
- the processing register section implemented as successive writes to the address 1.

This implementation style was chosen for two reasons:

1. the registers in each section are correlated to each other and it makes sense to set all of them at once by using a convenient mechanism and,
2. it uses the least internal and external hardware resources.

The fallback of this implementation style is that the order these registers are written must be strictly respected by the software running on the external microprocessor.

The Status Register is read from the address 0 while the Natural Code corresponding to the processed PRS window is implemented as successive reads from the address 1.

The PRDEC coprocessor operation is described by the following step-by-step procedure and also by the flowchart shown in Figure 3-3.

0. Power Up the PRDEC coprocessor. Upon power up all the registers of the coprocessor are reset and the coprocessor goes into idle mode waiting for commands from the μP. The μP issues commands by writing data to specific registers of the PRDEC coprocessor.

1. The μP reads the Status Register, which is located at the address 0. The Status Register implements five flags that are read-only (denoted R) flags.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
0	0	0	interrupt	overflow	decode	ready	algebra
---	---	---	R	R	R	R	R

Table 3-2. Status register flags

The five flags have the following signification:

bit0. *algebra* flag

0: the algebra parameters are not loaded

1: the algebra parameters are loaded

bit1. *ready* flag

0: not ready for a PRS decoding

1: ready for a PRS decoding

bit2. *decode* flag

0: idle mode

1: a decoding process is running

bit3. *overflow* flag

0: milestone database is in boundaries

1: milestone database is out of boundaries

bit4. *interrupt* flag

0: no result available

1: the result of the decoding process is available

bit7..5 not implemented

2. If the *ready* flag in Status Register is set then go to step 5, else continue with the next step.

3. Load the setting registers located at the address 0.
   a. First it is written the parameter "Number of Alphabet Symbols" that represents the number q of symbols contained by a specified $GF(q)$. It also describes the size qxq of the two operation tables defined on this Galois field. Also the first writing operation to address 0 always clears the flags *algebra*, *ready*, *overflow*, and *interrupt* in the Status Register.
   Actually this parameter is written by the µP as $q-1$ because the coprocessor's hardware counts from 0.

b. Next the μP initializes the operating algebra by loading the addition and multiplication tables into the coprocessor's RAM. Because the operating algebra is commutative it does not matter what is loaded first, rows or columns. The software running on the μP should make sure that first the addition table is loaded followed by the multiplication table, and that exactly *qxq* data entries are written into each table.

c. The μP writes the "Degree of the Irreducible Polynomial" parameter that represents the degree *m* of the monic irreducible polynomial *h(x)* used to decode the PRS in the specified *GF(q)*. It also describes the size *m* of the FIFO memory that will hold the coefficients of *h(x)*.

   Actually the microprocessor writes this parameter as *m-1* because the coprocessor's hardware counts from 0.

d. Next the μP initializes the coefficient table implemented as FIFO memory. The software running on the μP should make sure that exactly *m-1* data entries are written into this table. There are initialized only *m-1* entries instead of *m* entries because the generator polynomial is a monic polynomial, which by definition has the leading coefficient always equal to 1.

e. Once the coprocessor is programmed with the operational parameters it sets the flag *algebra* thus signaling to the μP that the algebra parameters have been loaded into memory.

f. The parameter "Milestone Period" *p* is written next. This parameter will be used on building the milestone table, which will be saved in coprocessor's internal memory.

   This parameter is actually written by the μP as *p-1* because the coprocessor's hardware counts from 0.

g. Next the μP writes a PRS window of exactly *m* entries. This window is used as the Milestone Seed when building the milestone table and represents the first milestone recorded into the table. Also the milestone seed is saved in a separate register that is used by the milestone initialization process to detect the beginning of a new PRS cycle.

h. The coprocessor is ready to generate and fill in internally the milestone table. It does that by self-initiating a decoding cycle having the milestone seed as origin and recording every *p* internal PRS window as a milestone into the milestone table. During the executing of the decoding process the coprocessor sets the flag *decode* thus signaling the μP about its status.

i. On sequence matching between the current PRS window and the saved milestone seed, which corresponds to parsing a full PRS cycle, the decoding process stops, the flag *decode* is cleared and because the milestone table is completed the flag *ready* is set. The milestone database has a finite size and there is the possibility that the number of milestones required to be saved is greater than the storage capacity of the database. In this case the database unit sets the flag *overflow* that signals to μP that the storage capacity .of the milestone database overflowed.

j. The coprocessor signals the μP that it is ready to accept a PRS window for decoding or that the storage capacity of the milestone database overflowed by generating an interrupt to the μP. Also on generating the interrupt to the μP the flag *interrupt* in the Status Register is set.

4. Go to step 1.

5. Load a PRS window to the processing registers located at the address 1.
   a. The μP writes a PRS window of exactly *m* data entries. Also the first write to the address 1 always clears the flag *interrupt* in the Status Register.
   b. On receiving the last data entry the coprocessor starts decoding the PRS window and sets the flag *decode* thus signaling the μP about its status.
   c. On sequence matching the decoding process stops and the flag *decode* in the Status Register is cleared.
   d. The coprocessor signals the μP that it is ready to accept a new PRS window for decoding by generating an interrupt to the μP. Also on generating the interrupt to the μP the flag *interrupt* in the Status Register is set.

6. Go to step 1.

The PRDEC coprocessor implements a *m*-wide serial-parallel conversion from the *GF(q)* space into the natural code space. When decoding is done the PRDEC coprocessor generates an interrupt to the μP and then goes into idle mode. The μP will then read the results (milestone-cluster, -index, and -offset) presented as contents of the memory address 1. The milestone cluster is available on the first read, the milestone index inside of the specified cluster on the second read, and the offset from that milestone on the third read.

The pseudo-random-to-natural code conversion implies calculating the PRS window index with respect to a reference point according to the equations shown in Figure 3-2.

```
window_index = (milestone_cluster * (polynom_degree+2) + milestone_index) *
                    milestone_period – milestone_offset

if window_index < 0 then
        window_index = window_index + sequence_length
end if
```

Figure 3-2. Pseudo-random-to-natural code conversion algorithm

Once the μP obtains the decoded word from the coprocessor it can initiate the next decoding operation by writing a new PRS window to the coprocessor.

The algebra inside the coprocessor can be changed by writing to the corresponding parameter registers only when the coprocessor is in idle mode.

The coprocessor expects the μP to send always the right type and amount of data as its decoding parameters.

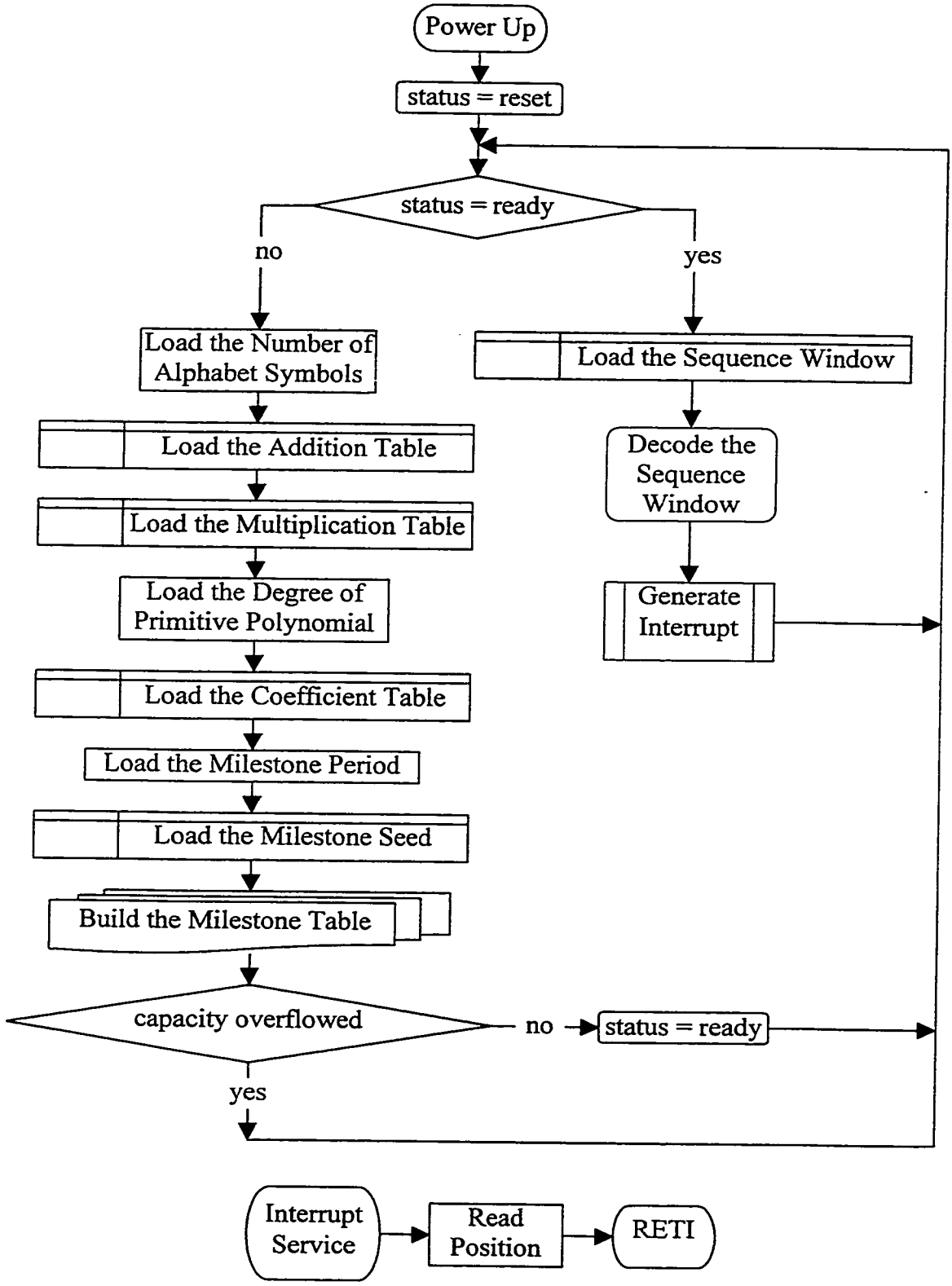


Figure 3-3. PRDEC coprocessor operational flowchart

The coprocessor consists of two major blocks named the I/O Unit and the PRDEC Core, as shown in Figure 3-4.

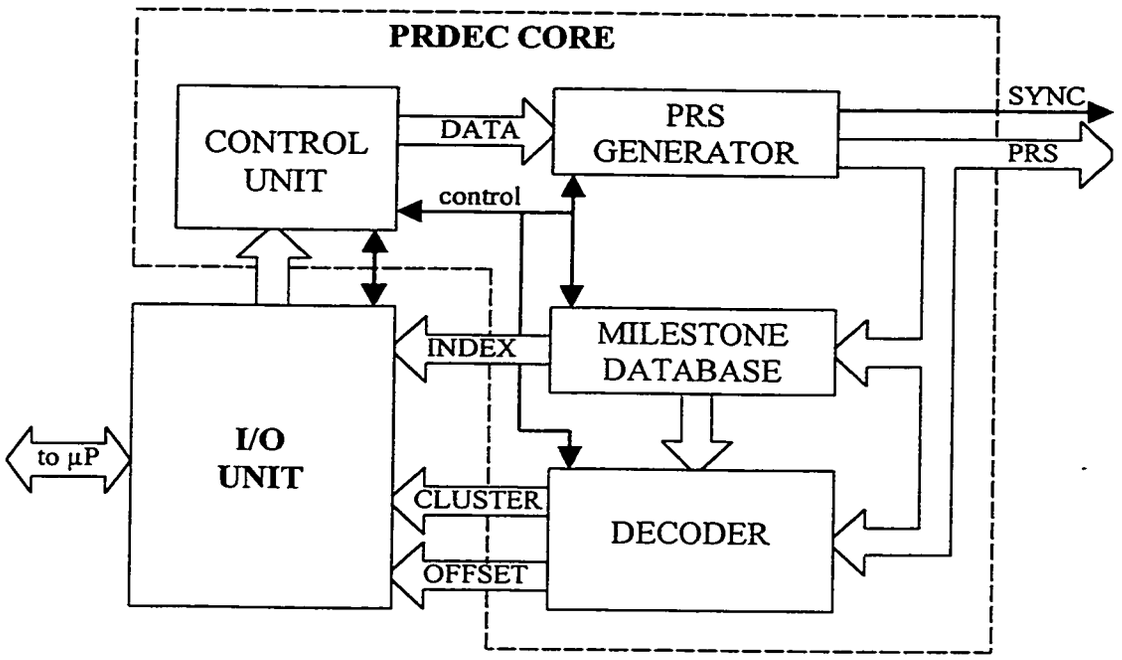


Figure 3-4. Block diagram of the coprocessor

The I/O Unit represents the bridge between the external embedded μP and the PRDEC Core.

The PRDEC Core implements all functions related to generating and decoding a PRMVS according to a specified algebra. It can be easily connected to a large variety of processors by using the generic I/O Unit presented above or a customized μP interface, the ultimate target being to build affordable Systems-On-Chip (SOC) aimed to the mobile robot industry. The PRDEC Core is divided further in smaller functional blocks which perform specialized tasks.

The Control Unit represents the managerial block in the sense that it generates specific orders to the other units, manages the data flow to and from the I/O Unit, and keeps track of the processing status. This unit coordinates the other units through a control bus.

The PRS Generator stores all the parameters of a specific algebra needed to generate PRSs using a sequence seed as the starting point. Both the algebra and the PRS window seed are received from the Control Unit. The output of this module is represented by the generated PRS and the symbol synchronization signal. This module is driven by and provides feedback to the Control Unit through signals on the control bus.

The Milestone Database Unit has two operating modes:

- In the acquisition mode it samples the PRS and stores certain sequence windows, called milestones, as entries into its FIFO type memory building a milestone database. The acquisition mode is triggered by the Control Unit on receiving the first element of the PRS seed just after the algebra parameters have been loaded. During the acquisition mode the unit outputs the PRS window seed to the Comparator Unit.
- In the normal mode this unit delivers the stored milestones as PRS windows to the Comparator Unit in a cycling fashion. The milestone index, which represents the index of the current output milestone in the database, is always available to the I/O Unit.

The milestones are stored in the database as clusters of size $m+2$ milestones as further explained in the section 3.1.3.

The Decoder Unit compares the two PRS windows, one coming from the PRS Generator and the other from the Milestone Database Unit, and signals the Control Unit on the matching condition. The milestone offset, which represents the number of PRS shifts from the matched milestone, is always available to the I/O unit.

3.1.1 Control Unit

The main component of the control unit is represented by a Finite State Machine (FSM) used to time and control the flow of the processes in the other modules. The FSM diagram is shown in Figure 3-5 and resembles very closely the flowchart described in Figure 3-3.

The FSM is driven by the writing operations at specific memory locations coupled with the values of the flags in the status register.

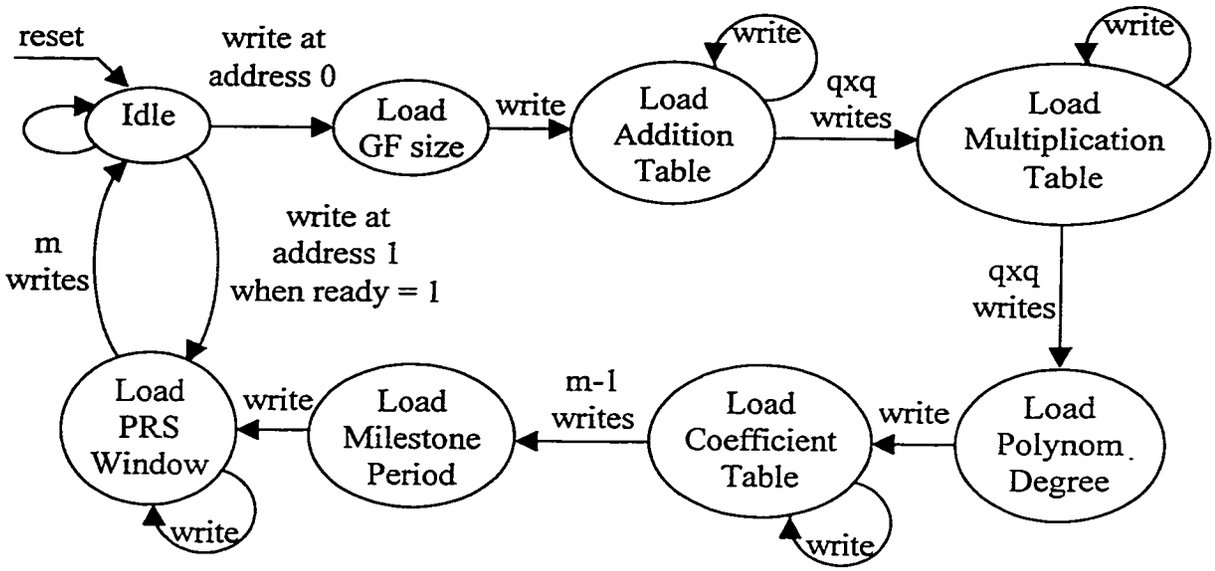


Figure 3-5. Control unit FSM

3.1.2 PRS Generator Unit

The main component of the PRS generator unit is represented by a multi-valued Linear Feedback Shift Register (LFSR) that implements the necessary logic to generate a PRS. Figure 3-6 presents the generic parallel architecture of a LFSR.

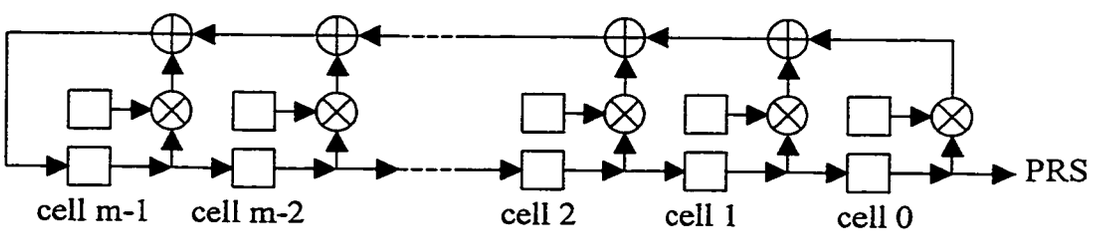


Figure 3-6. Parallel architecture of an m-cell LFSR

The hardware resources used to implement the parallel architecture of an *m*-cell LFSR consist of:

- *m*-cells for the irreducible polynomial coefficients;
- *m*-cells for the multiplication table;
- *(m-1)*-cells for the addition table;
- *m*-cells for the shift register;
- symbol generator cycle is *1* clock.

The behavior of the parallel processing architecture of a m-cell LFSR is described by the following equations:

$$cell(k-1) = cell(k), \qquad k = 1...m-1$$
$$cell(m-1) = \sum_{i=0}^{m-1} (cell(i) \otimes coef(i))$$

Equation 3-1. Behavior of the parallel m-cell LFSR

where the symbol "Σ" represents the addition operation and the symbol "⊗" represents the multiplication operation defined on a specific GF.

Example. 2-cell LFSR and its PRMVS output.

The PRMVS generator implemented with a 2-cell LFSR is defined by the irreducible polynomial $h(x) = x^2+x+2$ on $GF(4) = \{0, 1, t, t^2\}$ with $t^2+t+1=0$ and $t^3=1$. By replacing the symbols 0, 1, t, t^2, and t^3 with the natural numbers 0, 1, 2, 3, and respectively 4 the above statements can be rewritten as $GF(4) = \{0, 1, 2, 3\}$ with $3+2+1=0$ and $4=1$.

The LFSR corresponding to this irreducible polynomial is represented in Figure 3-7 and has the following feedback equation $a_{k+2} = a_{k+1} + 2 \cdot a_k$, due to the fact that $1+1=0$ and $2+2=0$ on $GF(4)$.

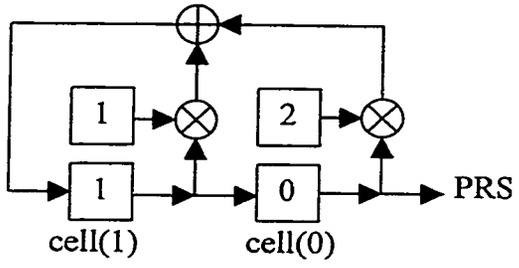


Figure 3-7. Parallel architecture of a 2-cell LFSR

Table 3-3 shows the cell states of a 2-cell LFSR where a symbol is available from the cell(0) output on every clock. The LFSR cells are the only clocked elements of this architecture.

step	x^1_k	x^0_k	$x^1_{k+1} = x^1_k + 2 \cdot x^0_k$	$x^0_{k+1} = x^1_k$	Note
0	1	0	$1 = 1 + 2 \cdot 0$	1	initial step
1	1	1	$3 = 1 + 2 \cdot 1$	1	
2	3	1	$1 = 3 + 2 \cdot 1$	3	
3	1	3	$0 = 1 + 2 \cdot 3$	1	$2 \cdot 3 = t \cdot t^2 = t^3 = 1$
4	0	1	$2 = 0 + 2 \cdot 1$	0	
5	2	0	$2 = 2 + 2 \cdot 0$	2	
6	2	2	$1 = 2 + 2 \cdot 2$	2	$2 \cdot 2 = t \cdot t = t^2 = 3$
7	1	2	$2 = 1 + 2 \cdot 2$	1	$2 \cdot 2 = t \cdot t = t^2 = 3$
8	2	1	$0 = 2 + 2 \cdot 1$	2	
9	0	2	$3 = 0 + 2 \cdot 2$	0	$2 \cdot 2 = t \cdot t = t^2 = 3$
10	3	0	$3 = 3 + 2 \cdot 0$	3	
11	3	3	$2 = 3 + 2 \cdot 3$	3	$2 \cdot 3 = t \cdot t^2 = t^3 = 1$
12	2	3	$3 = 2 + 2 \cdot 3$	2	$2 \cdot 3 = t \cdot t^2 = t^3 = 1$
13	3	2	$0 = 3 + 2 \cdot 2$	3	$2 \cdot 2 = t \cdot t = t^2 = 3$
14	0	3	$1 = 0 + 2 \cdot 3$	0	$2 \cdot 3 = t \cdot t^2 = t^3 = 1$
15	1	0	$1 = 1 + 2 \cdot 0$	1	replicate step 0
1	1	1	$3 = 1 + 2 \cdot 1$	1	replicate step 1

Table 3-3. Cell states of the parallel architecture of a 2-cell LFSR

By bundling up a LFSR cell together with a coefficient cell, a multiplication cell, and an addition cell to create a macrocell the parallel architecture diagram from above can be redrawn as in Figure 3-8.

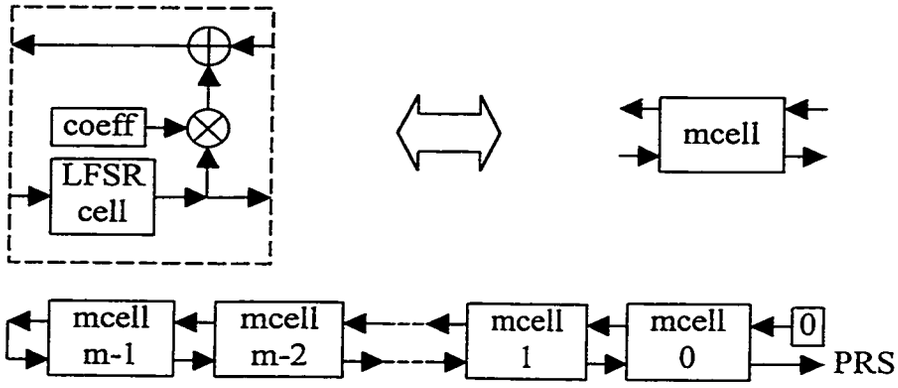


Figure 3-8. Simplified parallel architecture of a m-cell LFSR

The design should be flexible enough to accommodate a large variety of algebra over various Galois fields implying that the parameter storing cells have to be implemented as programmable memory cells.

Considering that every *GF(q)* has a finite number of elements, the easiest way to implement the two operations "+" (addition) and "*" (multiplication) defined over a specific *GF(q)* is by using programmable look-up tables (LUT).

Example

Each operation table defined on *GF(4) = {0, 1, 2, 3}* can be implemented as a look-up table of 4 rows by 4 columns, as shown in Figure 3-9. Considering that every element from *GF(4)* can be encoded binary in two bits then a 4-input 2-output LUT will be necessary to implement each of the two operations. Taking into account that in the case of the parallel

processing architecture each LFSR cell has its own pair of operation tables, the resources used increase linear with the length of the LFSR.

+	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

*	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

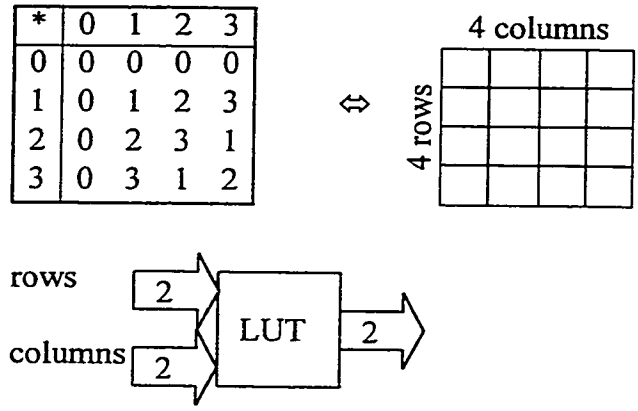


Figure 3-9. Operation tables over GF(4) implemented in LUT

It can be observed that all the LFSR macrocells employ identical LUTs for the arithmetic operations, and by converting the parallel processing architecture of the LFSR into a serial processing architecture, which uses a pipelined design, the hardware resources used will decrease significantly.

By using only one addition LUT and one multiplication LUT for all LFSR cells, this pipelined serial processing architecture represents the most economical implementation. Of course this comes at the expense of decreasing the PRS generating speed; where a symbol will not be available for every clock but every many clocks and it will depend mainly of the LFSR length.

Next there is shown how the parallel processing architecture can be converted to a serial processing architecture by employing the pipeline concept.

Considering the notation *mul(k) = cell(k) * coef(k)* and *add(k) = mul(k) + add(k-1)* with *add(0) = mul(0) + 0* the Equation 3-1, which represents the behavior of the parallel architecture of a m-cell LFSR, can be rewritten as:

$$cell(k-1) = cell(k), \qquad k = 1, ..., m-1$$

$$\begin{aligned} cell(m-1) &= \sum_{i=0}^{m-1} (cell(i) * coef(i)) \\ &= \sum_{i=0}^{m-1} mul(k) \\ &= mul(m-1) + mul(m-2) + ... + mul(2) + mul(1) + mul(0) + 0 \\ &= mul(m-1) + mul(m-2) + ... + mul(2) + mul(1) + add(0) \\ &= mul(m-1) + mul(m-2) + ... + mul(2) + add(1) \\ &\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ &= mul(m-1) + add(m-2) \\ &= add(m-1) \end{aligned}$$

$$add(k) = mul(k) + add(k-1), \qquad k = 1, ..., m-1$$

$$add(0) = mul(0) + 0$$

Equation 3-2. Behavior of the serial m-cell LFSR

The detailed architecture implementing the above equation is shown in Figure 3-10, and represents the serial processing architecture that has been converted from the parallel processing architecture in Figure 3-6. This pipelined architecture uses a single pair of addition and multiplication cells making it the most efficient implementation, from the point of view of hardware resources used.

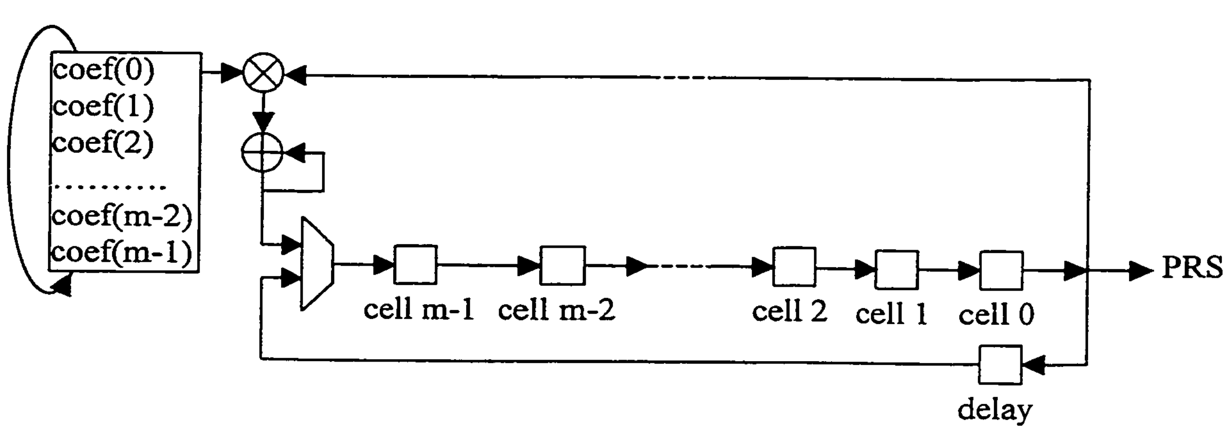


Figure 3-10. Pipelined architecture of a m-cell LFSR

It should be noted that in order to synchronize the new architecture a delay cell must be inserted. The multiplier's coefficients form a FIFO type memory structure.

Example

Figure 3-11 represents the pipelined architecture converted from the architecture presented in Figure 3-7.

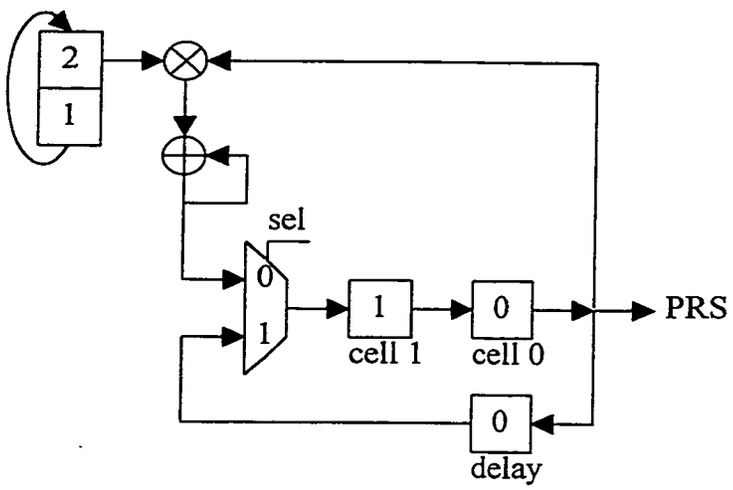


Figure 3-11. Pipelined architecture of a 2-cell LFSR

Considering that the pipelined architecture has its elements (LFSR cells, polynomial coefficient FIFO cells, addition and multiplication cells) clocked then the equations for the 2-cell pipelined LFSR are as follows:

$$
\begin{aligned}
&mul(k) = coef(k-1) * cell(0)(k-1) \\
&add(k) = add(k-1) + mul(k-1) \\
&cell(1)(k) = \begin{cases} add(k-1), & for\ k(\mathrm{mod}\,4) = 0 \\ delay(k-1), & for\ k(\mathrm{mod}\,4) \neq 0 \end{cases} \\
&cell(0)(k) = cell(1)(k-1) \\
&delay(k) = cell(0)(k-1)
\end{aligned}
$$

Equation 3-3. Equations of the pipelined architecture for the 2-cell LFSR

Table 3-4 shows the states of the 2-cell LFSR pipelined architecture where a symbol is available from the cell(0) output every 4 clocks. Every row in the table represents the status of the cells at a specific clock.

step	μstep	sel	coef	*	+	cell(1)	cell(0)	delay	Comments
0	0	1	2	—	—	1	0	—	initial state
	1	1	1	0=2·0	0	—	1	0	reset the adder
	2	1	—	1=1·1	0=0+0	0	—	1	
	3	0	—	—	1=0+1	1	0	—	mux = adder
1	0	1	2	—	—	1	**1**	0	1st symbol
	1	1	1	2=2·1	0	0	1	1	reset the adder
	2	1	—	1=1·1	2=0+2	1	0	1	
	3	0	—	—	3=2+1	1	1	0	mux = adder
2	0	1	2	—	—	3	**1**	1	2nd symbol
	1	1	1	2=2·1	0	1	3	1	reset the adder
	2	1	—	3=1·3	2=0+2	1	1	3	
	3	0	—	—	1=2+3	3	1	1	mux = adder
3	0	1	2	—	—	1	**3**	1	3rd symbol
	1	1	1	1=2·3	0	1	1	3	reset the adder
	2	1	—	1=1·1	1=0+1	3	1	1	
	3	0	—	—	0=1+1	1	3	1	mux = adder
4	0	1	2	—	—	0	**1**	3	4th symbol
	1	1	1	2=1·1	0	3	0	1	reset the adder
	2	1	—	0=1·0	2=0+2	1	3	0	
	3	0	—	—	2=2+0	0	1	3	mux = adder
5	0	1	2	—	—	2	**0**	1	5th symbol
	1	1	1	0=2·0	0	1	2	0	reset the adder
	2	1	—	2=1·2	0=0+0	0	1	2	
	3	0	—	—	2=0+2	2	0	1	mux = adder
6	0	1	2	—	—	2	**2**	0	6th symbol
	1	1	1	3=2·2	0	0	2	2	reset the adder
	2	1	—	2=1·2	3=0+3	2	0	2	
	3	0	—	—	1=3+2	2	2	0	mux = adder
7	0	1	2	—	—	1	**2**	2	7th symbol
	1	1	1	3=2·2	0	2	1	2	reset the adder
	2	1	—	1=1·1	3=0+3	2	2	1	
	3	0	—	—	2=3+1	1	2	2	mux = adder
8	0	1	2	—	—	2	**1**	2	8th symbol
	1	1	1	2=2·1	0	2	2	1	reset the adder
...	...	...	...	...	...	...	...	...	...

Table 3-4. Cell states of the pipelined architecture of a 2-cell LFSR

The number of microsteps (clocks) necessary to get a new symbol from the LFSR is defined as microcycle; and every microcycle the symbol counter (step) is incremented.

It can be seen from the Table 3-4 that all the cells of the pipelined architecture are synchronous and have a linear behavior except the addition cell that has to be cleared every four clocks at the beginning of a new microcycle (μstep = 1), and the coefficient FIFO memory that has to be stopped from rolling for two cycles at the end of the microcycle (μstep = 2, 3).

The pipelined LFSR implementing the irreducible polynomial *h(x)* of degree *m* defined on *GF(q)* requires *m+2* clocks to generate a symbol.

Hardware resources used to implement the pipelined (serial) processing architecture are:

- *m*-cells for the irreducible polynomial coefficients;
- *1*-cell for the multiplication table;
- *1*-cell for the addition table;
- *(m+1)*-cells for the shift register (*1* cell used to implement a delay cell);
- symbol generator cycle is *m+2* clocks.

The Table 3-5 shows a comparison between the two types of architectures discussed for a *m*-cell LFSR meant to implement the PRS generator unit.

Resources	Parallel architecture	Pipelined architecture
coefficient cells	*m*	*m*
multiplication cells	*m*	*1*
addition cells	*m-1*	*1*
shift register cells	*m*	*m+1*
symbol cycle	*1*	*m+2*

Table 3-5. Resource comparison for the two architectures

It can be seen that the pipelined architecture uses by far less resources than the parallel architecture at the expense of increasing the processing time.

Between the two, the pipelined architecture is chosen for implementation.

3.1.3 Milestone Database Unit

The milestone database unit is used to store snapshots of certain PRS windows during the milestone initialization process and to retrieve them during the decoding process. The milestone storage/retrieving process is controlled by the flags *decode* and *ready* provided by the control unit. The flag *decode* activates the milestone database and the flag *ready* specifies the type of operation performed by the database; milestone storage operation when *ready* flag is cleared and milestone retrieving operation when *ready* flag is set.

The milestone database unit provides saved PRS window data to the comparator cell during the PRS decoding process in a serial-parallel fashion. Because of the pipelined architecture of the PRS generator unit a symbol is generated every $m+2$ clock cycles, where m is the degree of the irreducible polynomial and the extra two cycles are due to the latency introduced by the two operation cells. A new symbol available every $m+2$ clock cycles means that a new PRS window is available with the same frequency; or reversibly, a PRS window is stable for $m+2$ clock cycles.

The PRS decoding process consists of comparing the current PRS window with the milestones saved in the database. Knowing that a PRS window is stable for $m+2$ clock cycles and the decoding process is synchronous with the clock then every PRS window can be compared with $m+2$ milestones using a single comparator cell. This concept is illustrated in the Figure 3-12 and represents the serial aspect of the decoding process when by executing a comparison every clock cycle the comparator cell is used at full bandwidth.

In order to speed up the decoding process for long PRS cycles multiple comparator cells can process in parallel the current PRS window, where each comparator is cycling its own set of *m+2* milestones. Employing multiple comparator cells in parallel to process the same PRS window represents the parallel aspect of the decoding process. To accommodate easily the serial-parallel decoding process the milestone database is divided in clusters each having the size of *m+2* milestones with each cluster supplying its milestones to a certain comparator cell assigned to it.

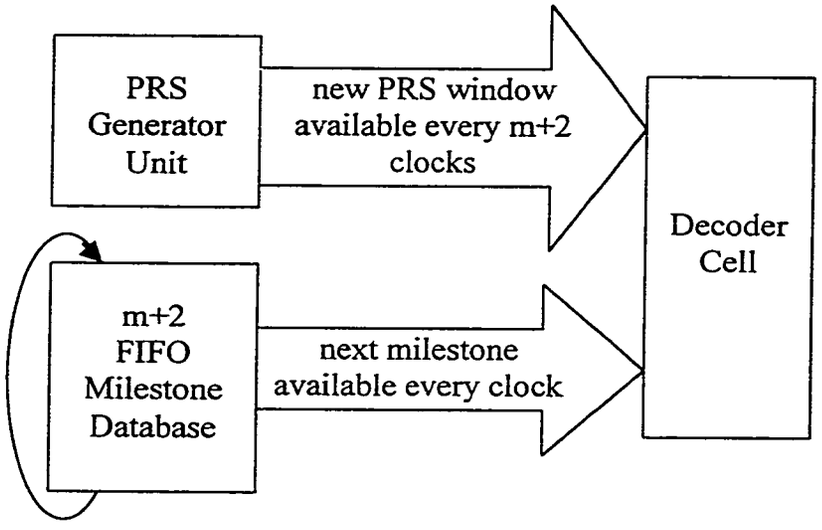


Figure 3-12. Decoding a PRS window using a pipelined comparator

Figure 3-13 illustrates the cluster architecture of the milestone database.

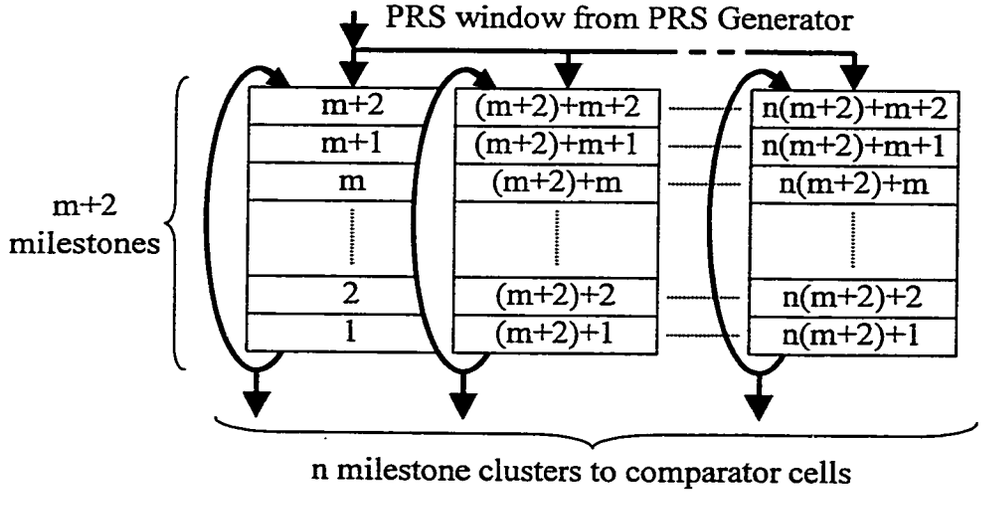


Figure 3-13. Milestone database architecture

The initialization process fills up successively each milestone cluster with certain PRS windows according to the milestone period parameter. Once a milestone cluster is full then the next cluster is filled up.

During the retrieving process all milestone clusters roll synchronously presenting continuously milestones with the same cluster index to all the comparator cells at the same time.

The cardinal k of a milestone located in the cluster i having index j inside that cluster can be calculated using the formula $k = i \bullet (m+2) + j$ and represents the $(p \bullet k)$-th PRS window from the origin window, where p is the milestone period.

3.1.4 Decoder Unit

The PRS decoding process consists of comparing the current PRS window with the milestones saved in the Milestone Database Unit. The decoding process ends when the current PRS window matches a milestone. The control unit makes the μP aware of the matching process by generating an interrupt. On servicing the interrupt the μP reads the result of the decoding process, result that is used to compute the natural code of the loaded PRS window.

The Decoding Unit is comprised of multiple comparator cells that compare in parallel the current PRS window available from the PRS Generator Unit with the milestones saved in the Milestone Database Unit. Each comparator cell is assigned to a milestone database cluster, as shown in Figure 3-14.

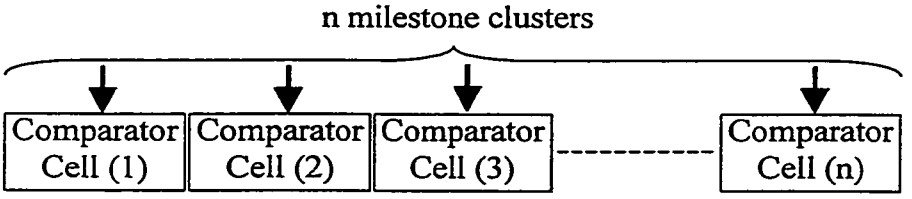


Figure 3-14. Decoder unit architecture

3.1.5 Input/Output Unit

The I/O Unit constitutes the bridge between the external embedded microprocessor and the PRDEC Core. This unit connects to the μP through a generic memory interface that employs a 1-bit address bus, an 8-bit data bus, and the corresponding control signals, as shown in Figure 3-1.

The PRDEC Core is connected to the I/O Unit through a proprietary interface. The I/O Unit interprets the commands that come from the μP and generates the proper control signals used by the PRDEC Core. If necessary this I/O Unit can be removed from the design and replaced with a custom interface unit tailored to a specific μP.

3.2 Coprocessor Implementation

Hardware Description Languages (HDLs) are used to model the operations of hardware architectures. There are two aspects of hardware description that a HDL facilitates [Smi96]:

1) *Abstract behavior modeling* implying that the language is declarative in order to allow the abstract description of hardware behavior for specific purposes. This behavior is not prejudiced by structural or design aspects of the hardware intent.
2) *Hardware structure modeling* implying that the hardware structure is capable of being modeled in a hardware description language irrespective of design's behavior.

During the design process, the behavior of the hardware may be modeled and represented at various levels of abstraction; thus, higher level models describe the operation of hardware abstractly, while lower models include more detail, such as inferred hardware structure [Smi96].

In 1980 the US government developed the Very High Speed Integrated Circuit (VHSIC) project to enhance the electronic design process, technology, and procurement, spawning the development of many advanced integrated circuit process technologies. This was followed by the arrival of VHSIC Hardware Description Language (VHDL), which was endorsed by IEEE in 1986 in its attempt for standardization [Cha97]. The resulting standard, IEEE 1076-1987, is the basis for virtually every simulation and synthesis product sold today. An enhanced and updated version of the language, IEEE 1076-1993, was released in 1994, and VHDL tool vendors have been responding by adding these new language features to their products [Pel95].

The PRDEC coprocessor implementation was done in VHDL at Register Transfer Logic (RTL) level by employing a scaleable design methodology that makes it is easy to fine tune the implementation for a very specific application by choosing the corresponding generic parameters. The entity section of the hierarchy top file showing the implementation generic parameters, which are self explanatory due to the specific nature of the VHDL, is presented in Figure 3-15.

```
entity Coprocessor is
    -- Coprocessor Unit component
    generic (
        ADDRESS_SIZE:           INTEGER:= 2;   -- width of the address bus
        DATA_SIZE:              INTEGER:= 8;   -- width of the data bus
        GF_WIDTH:               INTEGER:= 2;   -- width of GF elements
        DOUBLE_GF_WIDTH:        INTEGER:= 4;   -- width of entry address for
                                               -- GF operation tables = 2*GF_WIDTH
        MAX_POLYNOM_DEGREE:     INTEGER:= 4;   -- maximum degree of the
                                               -- irreducible polynomial
        POLYNOM_DEGREE_WIDTH:INTEGER:= 3;      -- width of MAX_POLYNOM_DEGREE equal
                                               -- to LOG2(MAX_POLYNOM_DEGREE) + 1
                                               -- rounded up to the near integer
        MILESTONE_SIZE:         INTEGER:= 8;   -- the size (in binary bits) of a
                                               -- milestone word saved in FIFO equal
                                               -- to MAX_POLYNOM_DEGREE * GF_WIDTH
        MAX_NUMWORDS:           INTEGER:= 6    -- maximum number of milestones
                                               -- saved in FIFO equal to
                                               -- MAX_POLYNOM_DEGREE + 2 (latency)
        MILESTONE_CLUSTERS:     INTEGER:= 4;   -- number of milestone clusters
        MILESTONE_CLUSTER_WIDTH:INTEGER:= 2    -- milestone cluster width
                                               -- equal to LOG2(MILESTONE_CLUSTERS)
    );
```

Figure 3-15. Generic implementation parameters of the PRDEC coprocessor

Also by implementing the design using VHDL, which is a standard language, the implementation of the PRDEC coprocessor becomes vendor independent.

The PRDEC coprocessor is implemented according to a hierarchical architecture. The implementation hierarchy of the coprocessor is shown in Figure 3-16 and it resembles the block diagram shown in Figure 3-4. It can be seen that the coprocessor contains two major blocks, the generic Input/Output interface unit to the μP and the PRDEC Core, which itself is divided in four functional units: the Control Unit, the PRS Generator Unit, the Milestone Database Unit, and the PRS Decoder Unit.

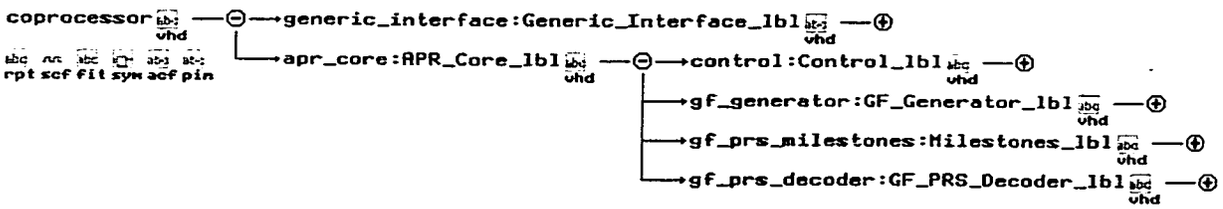


Figure 3-16. Coprocessor implementation hierarchy

While the design follows the top-down approach, the simulation follows a more natural bottom-up approach. By respecting strictly either a top-down or a bottom-up approach, some important aspects of either the design or the simulation may become obscured. In order to overcome this, all circuit modules are organized into VHDL libraries according to their hierarchical levels and the presentation then flows according to the logical structure of the libraries.

The practical implementation of the coprocessor design requires the hardware medium:

- to be easy to prototype with,
- to be re-programmable,
- to have a LUT based internal architecture, and
- to have internal RAM.

The Altera's FLEX10K family, which is a SRAM-based Complex Programmable Logic Device (CPLD), was chosen for implementation of the coprocessor design because it fulfils all the above requirements. The Altera's MAX+PLUS II, version 9.01, software package was used as design environment.

Although the design was implemented in Altera's FLEX10K technology there is only one design library that makes the link to that technology. To validate the design for other vendor's technology this library has to be simply remapped to the new vendor's library.

Annex A presents the VHDL libraries used to implement and simulate the PRDEC coprocessor. Following there is a short description of the content of this annex.

A-1. The lower in hierarchy is the *VENDOR LIBRARY*, described by the package *VENDOR_package*, that contains basic generic components (primitives) from which all the other component libraries have been derived. These primitives are:

- various flip-flops (D, JK, and T type) with and without enable signal,
- the binary counter,
- the data register, and
- RAM and ROM architectures.

The mappings of these primitives are tuned toward the Altera's architecture. This library is presented in Annex A-1.

The *VHDL LIBRARY*, described by the package *VHDL_package*, contains generic components based solely on *VENDOR LIBRARY*. These components are:

- double flip-flops,
- pulse edge detector, and
- generic encoder.

This library is also presented in Annex A-1.

A-2. The *PRDEC LIBRARY*, described by the package *PRDEC_package*, contains the functional units of the coprocessor core along with their subcomponents. The functional units contained in this library, as shown in the Figure 3-4, are:

- the Control Unit,
- the PRS Generator Unit,
- the Milestone Database Unit, and
- the Decoder Unit.

The subcomponents used to build these functional units are:

- the multidimensional shift register with taps used to implement the LFSR,
- the FIFO memory with taps used in implementing the polynomial coefficient memory and the Milestone Database Unit,
- the tap decoder.

This library is presented in Annex A-2.

A-3. The *INTERFACE LIBRARY*, described by the package *INTERFACE_package*, contains I/O interfaces for various bus architectures. These interfaces are:

- the Generic Memory Interface, and
- the HC11 Interface.

This library is presented in Annex A-3.

A-4. The *MICROPROCESSOR LIBRARY*, described by the package *MICROPROCESSOR_package*, contains functional interfaces that emulate various bus architectures used to exercise the PRDEC coprocessor design. These emulation interfaces are:

- the Generic Microprocessor Emulation, and
- the HC11 Microprocessor Emulation.

This library is presented in Annex A-4.

Following, the implementations of the coprocessor's functional units are described in detail and the hierarchical architecture and simulation waveforms of each of them is shown.

3.2.1 Control Unit

The Control Unit implements the FSM shown in Figure 3-5 as a Moore state machine. This unit performs the following tasks during a state machine cycle:

- stores the parameter "Number q of Alphabet Symbols corresponding to *GF(q)*" ;
- stores the parameter "Degree m of the Irreducible Polynomial *h(x)*" when the FSM = Load Polynomial Degree state;
- stores the parameter "Milestone Period" when the FSM = Load Milestone Period;
- asserts the signal LOAD_ADD while the FSM = Load Addition Table state;
- asserts the signal LOAD_MUL while the FSM = Load Multiplication Table state;
- asserts the signal LOAD_COEF while the FSM = Load Coefficient Table state;
- asserts the signal LOAD_WINDOW while the FSM = Load PRS Window state.

The parameter "Number q of Alphabet Symbols corresponding to *GF(q)*" specifies how long the FSM will stay in the states corresponding to the loading process of the addition and multiplication tables. This parameter actually specifies the amount of entries into the two operation tables, addition and multiplication, defined on a specific *GF(q)*. Data is loaded into the RAM-based addition and multiplication tables while the signals LOAD_ADD and respectively LOAD_MUL are asserted.

The parameter "Degree m of the Irreducible Polynomial *h(x)*" is decoded into the *taps* signal. The *taps* signal is used by the PRS Generator Unit to set the length of the irreducible polynomial coefficient table implemented as FIFO memory and also to set the length of the LFSR. This parameter also specifies how long the FSM will stay in the states corresponding to the loading processes of the coefficient table and respectively the PRS window. Because data is loaded into the FIFO-based irreducible polynomial coefficient table while the signal LOAD_COEF is asserted and respectively into the LFSR cells while the signal LOAD_WINDOW is asserted, this parameter controls the amount of data entered into the two FIFO-based memories.

The parameter "Milestone Period" is used by the Milestone Database Unit to control which PRS windows are stored into the database as milestones.

Feedback from the other design entities is represented by the signals *match* and *milestone_overflow*.

The signal *match* specifies that the current PRS window from the PRS Generator Unit matches the current milestone presented at the output of one of the milestone clusters of the Milestone Database Unit. This signal is used to control the status flag *ready*.

The signal *milestone_overflow* specifies that the amount of the milestones required to be stored into the Milestone Database Unit exceeds the storage capacity of this database.

The hierarchical architecture of the Control Unit is shown in Figure 3-17.

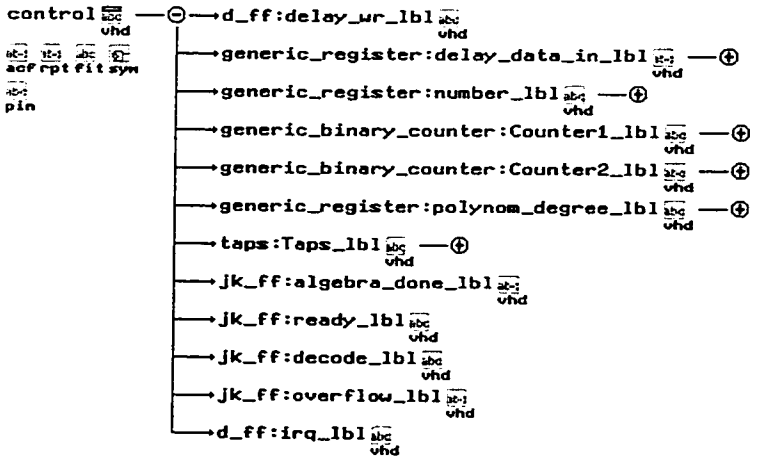


Figure 3-17. Implementation hierarchy of the Control Unit

The VHDL implementation code of the Control Unit is presented in Annex A-1.

The simulation waveform diagrams of the Control Unit are shown in Figure 3-18 and Figure 3-19.

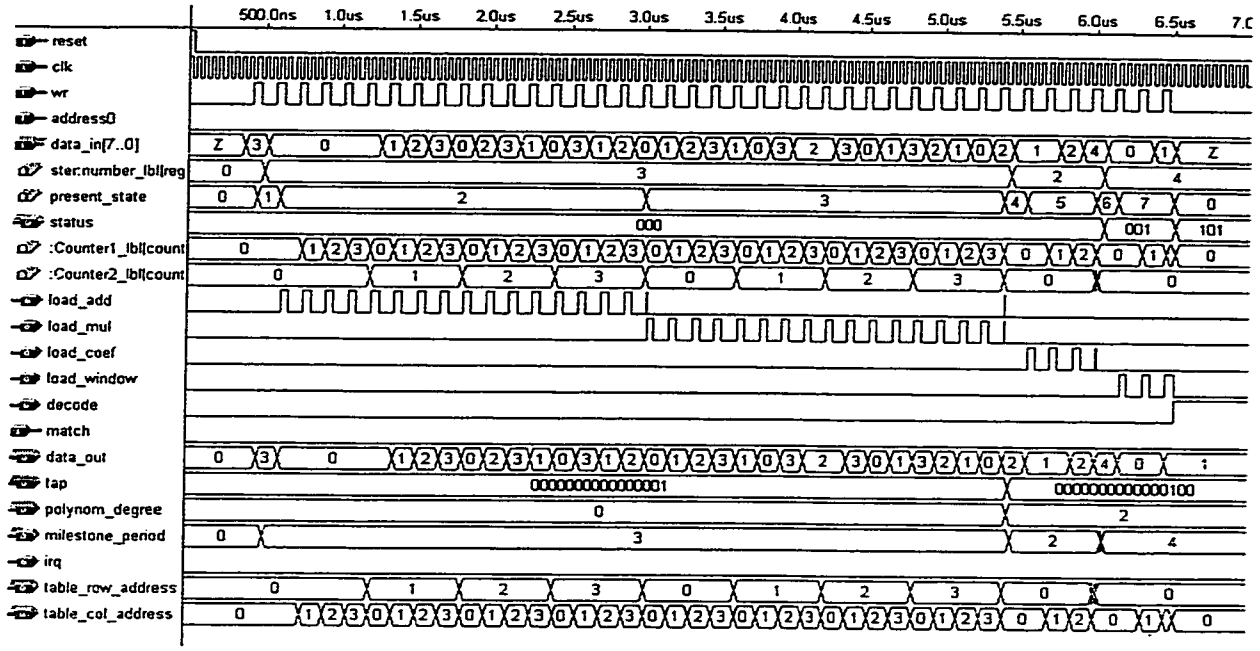


Figure 3-18. Waveform diagram: Control Unit writing algebra

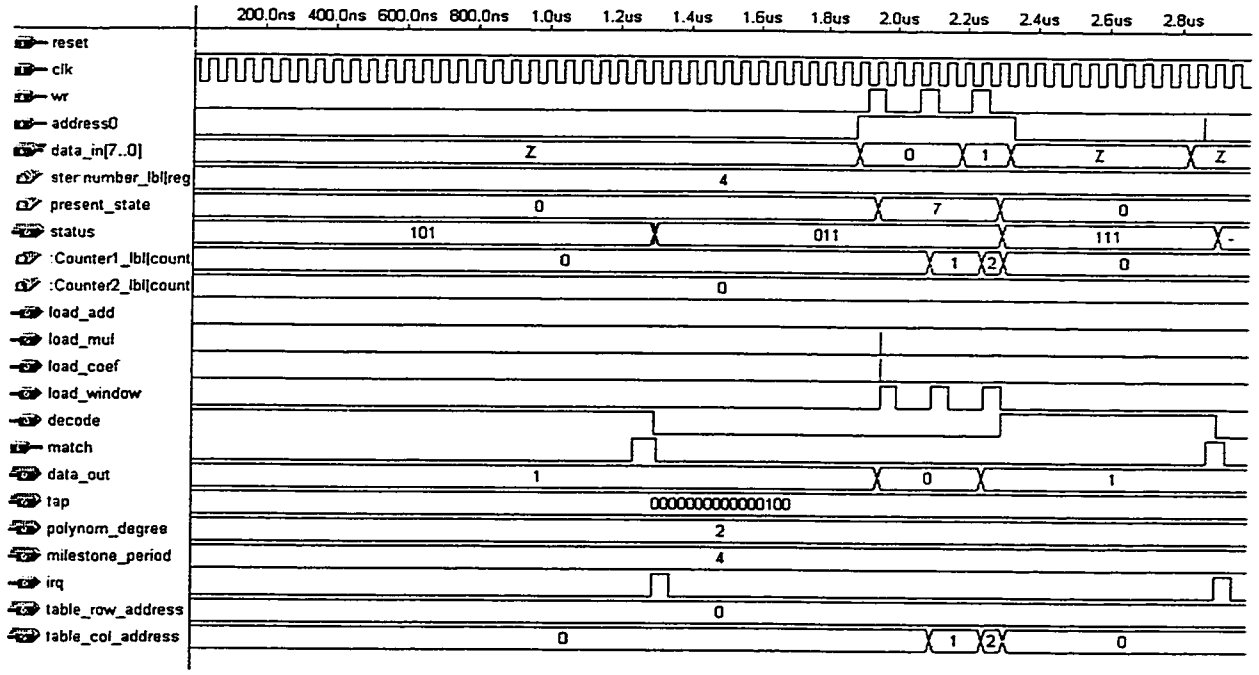


Figure 3-19. Waveform diagram: Control Unit writing a PRS window

3.2.2 PRS Generator Unit

The pipelined architecture of a *m*-cell LFSR shown in Figure 3-10 has to be completed with some extra circuitry that implements the following features:

- load the addition table;
- load the multiplication table;
- load the irreducible polynomial coefficient table;
- load the LFSR cells with the initial PRS window;
- allow for variable length of the FIFO memory that implements the irreducible polynomial coefficient table;
- allow for variable length of the LFSR.

Figure 3-20 shows a detailed pipelined architecture of a generic *m*-cell LFSR.

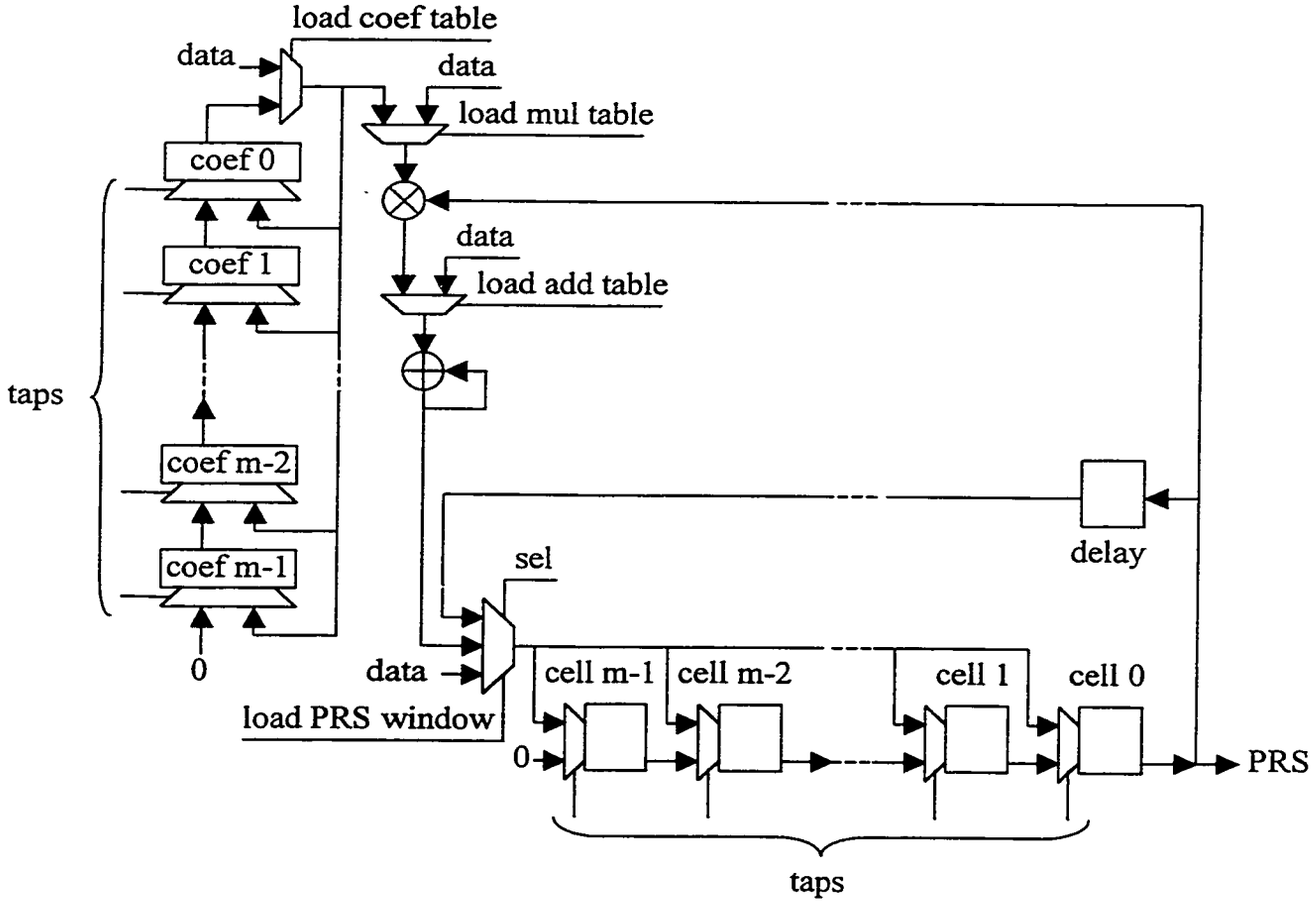


Figure 3-20. Detailed architecture of the generic pipelined m-cell LFSR

It should be noted that the symbols used to identify the adder and the multiplyer actually represent the addition and multiplication tables implemented as RAM.

Depending of the application, the coefficient table and the LFSR are required to have a variable length. This is done by using multiplexers whose select signals represent taps specified by decoding the programmed parameter "Degree of the Irreducible Polynomial".

It should be noted that the multiplexers used for data loading operations and for controlling the length of the coefficient table and the LFSR are driven by the control unit.

The hierarchical architecture of the PRS Generator Unit is shown in Figure 3-21.

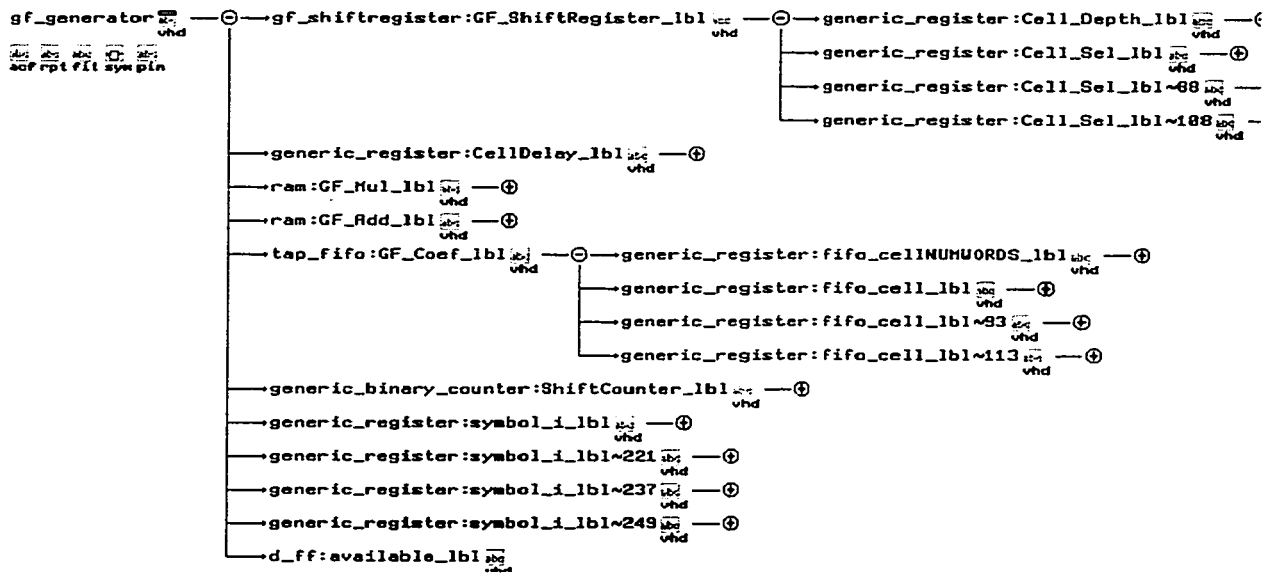


Figure 3-21. Implementation hierarchy of the PRS Generator Unit

The VHDL implementation code of the PRS Generator Unit is presented in Annex A-2.

The simulation waveform diagrams of the PRS Generator Unit are shown in Figure 3-22 and Figure 3-23.

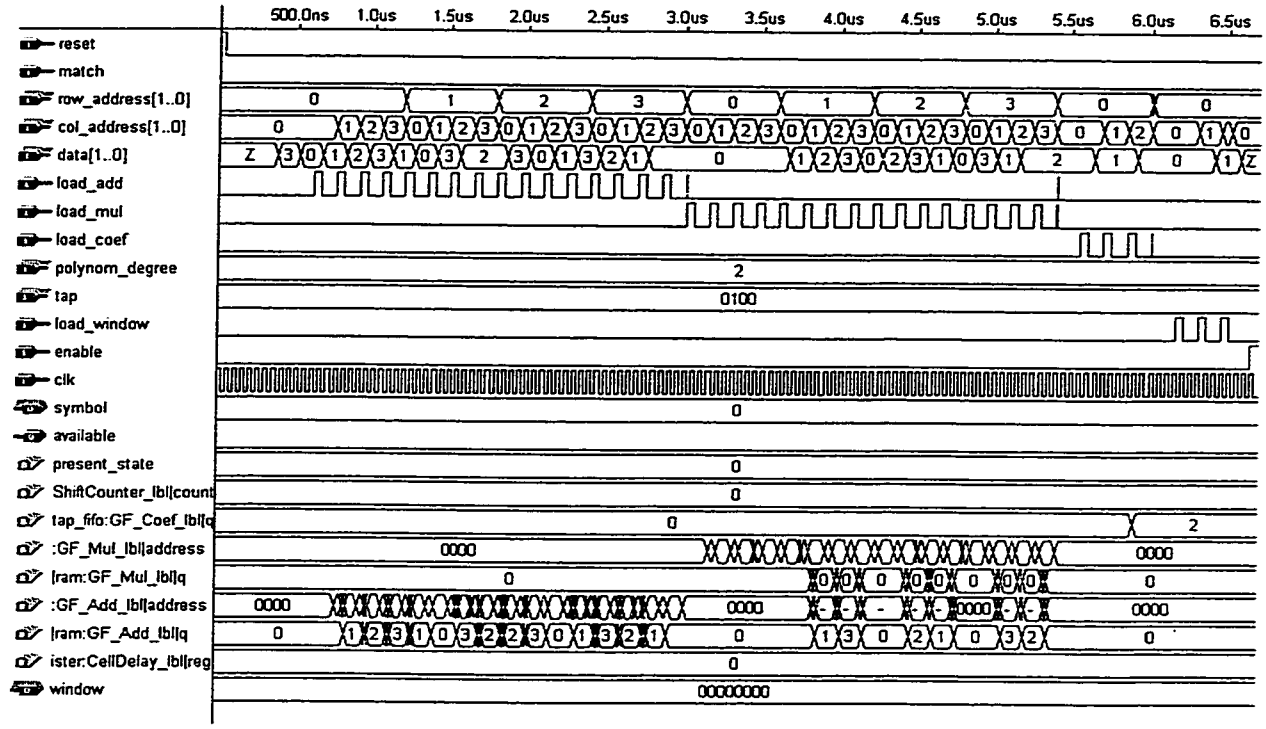


Figure 3-22. Waveform diagram: PRS Generator Unit loading algebra

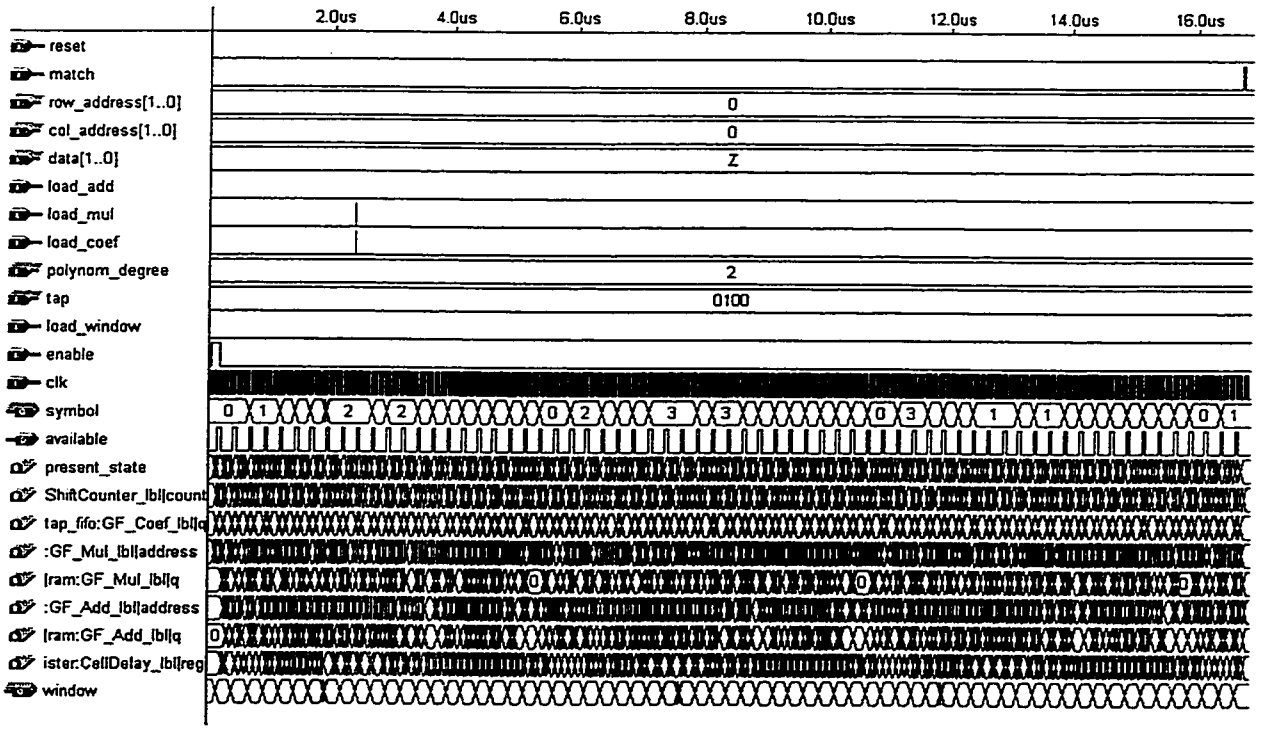


Figure 3-23. Waveform diagram: PRS Generator Unit generating a PRS

3.2.3 Milestone Database Unit

The Milestone Database Unit is comprised of multiple clusters each storing $m+2$ milestones, where m is the size of the LFSR in the PRS Generator Unit. The detailed architecture of the milestone storage mechanism is shown in Figure 3-24.

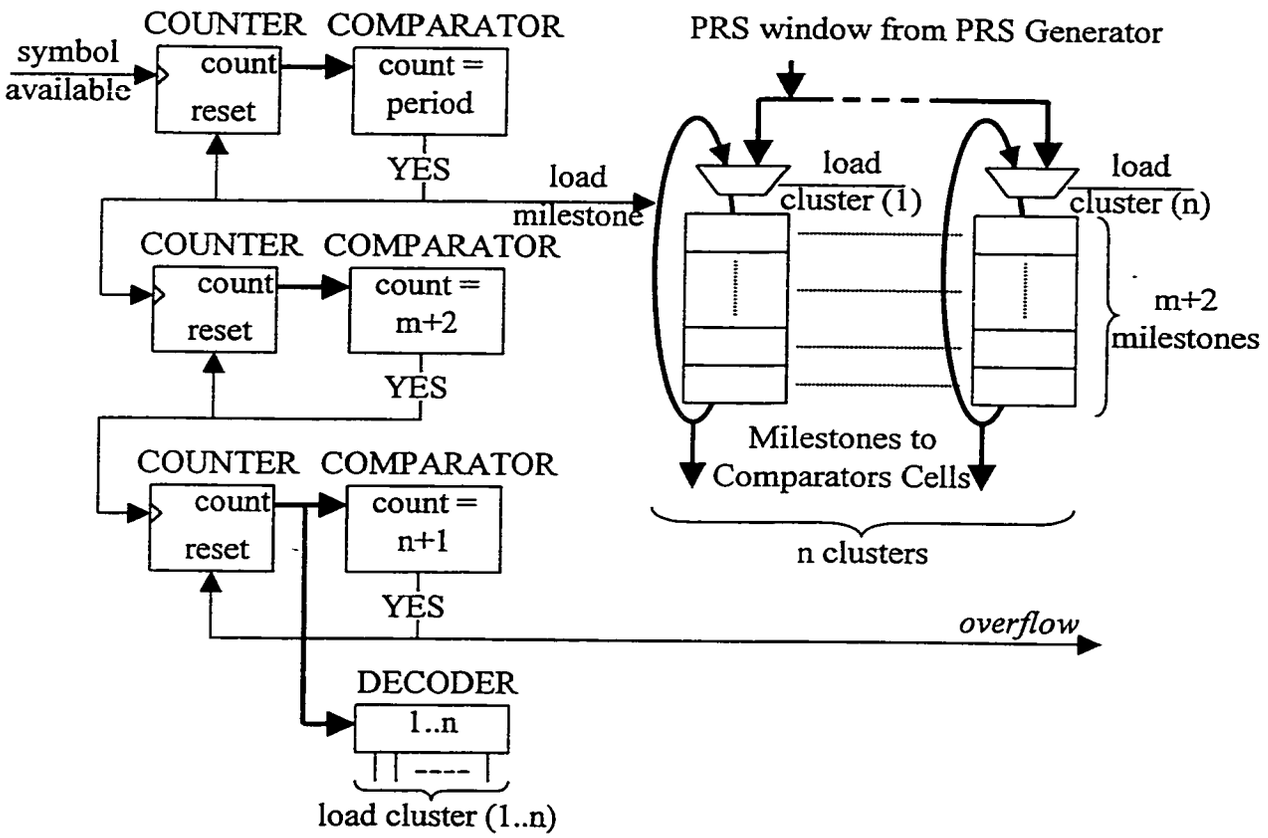


Figure 3-24. Detailed architecture of the milestone storage mechanism

The initialization process of the Milestone Database Unit is controlled by a series of successive counters and comparators that define the PRS window sampling moment and drive the milestone filling process in conjunction with selecting the active cluster, which is driven by a decoder. Also the logic signals the database *overflow* condition on a request for storing a milestone when the milestone database is already full (no milestone clusters available).

The hierarchical architecture of the Milestone Database Unit is shown in Figure 3-25.

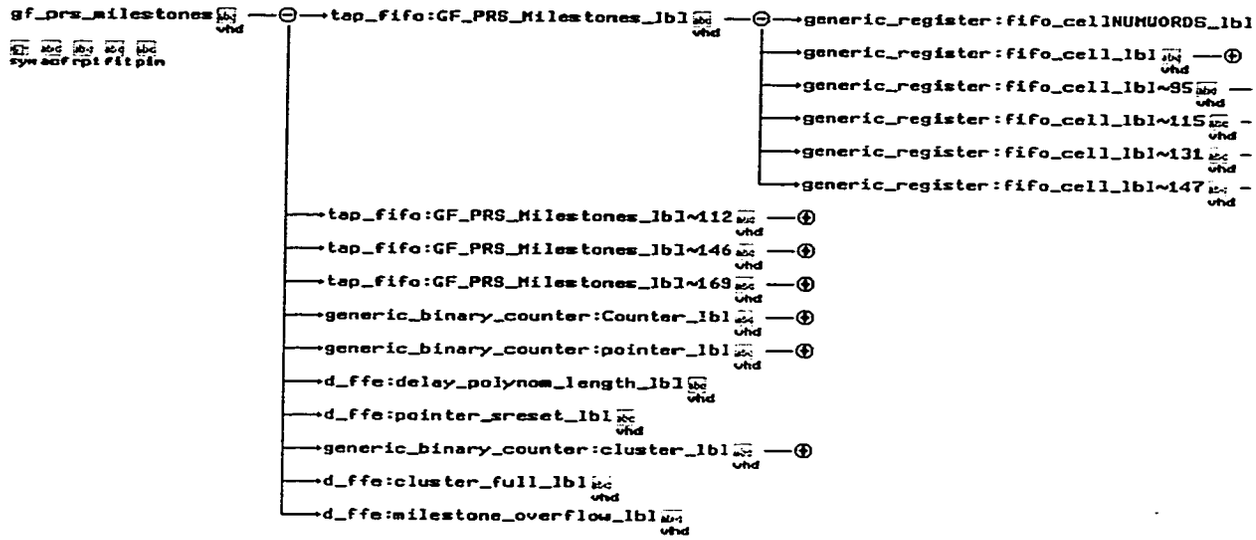


Figure 3-25. Implementation hierarchy of the Milestone Database Unit

Figures 3-26 and 3-27 show the simulation waveform diagrams of the Milestone Database Unit.

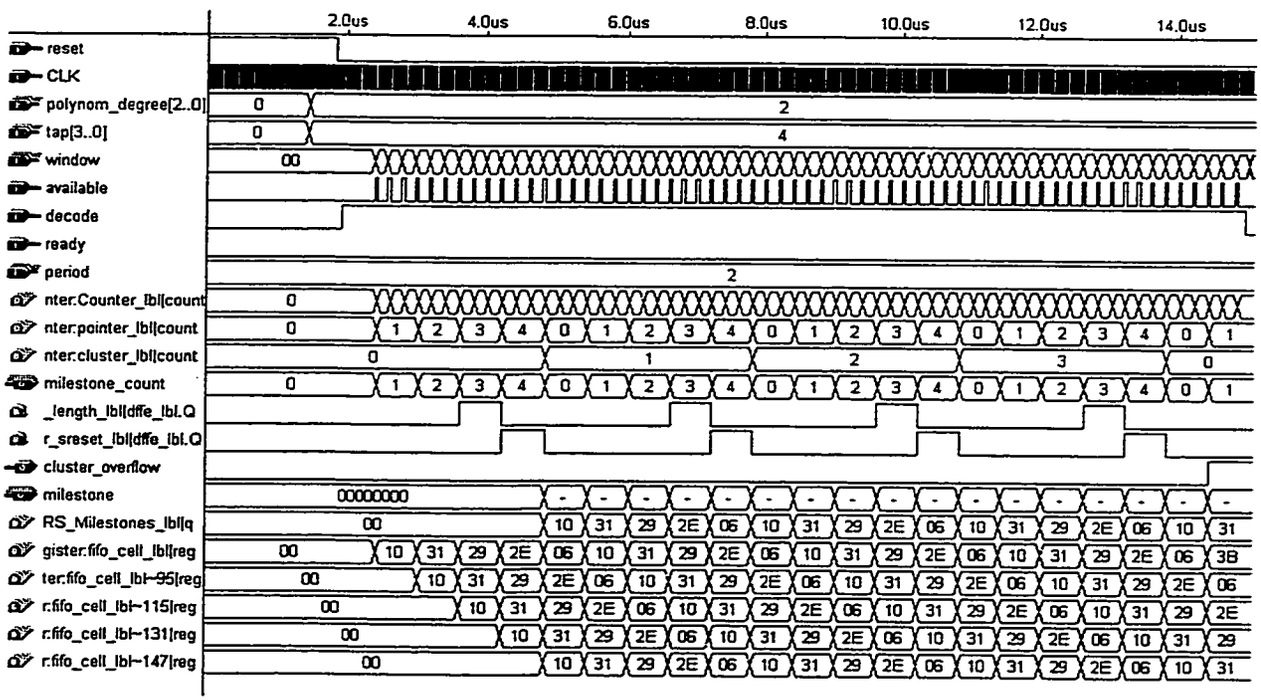


Figure 3-26. Waveform diagram: Milestone Database Unit storing data

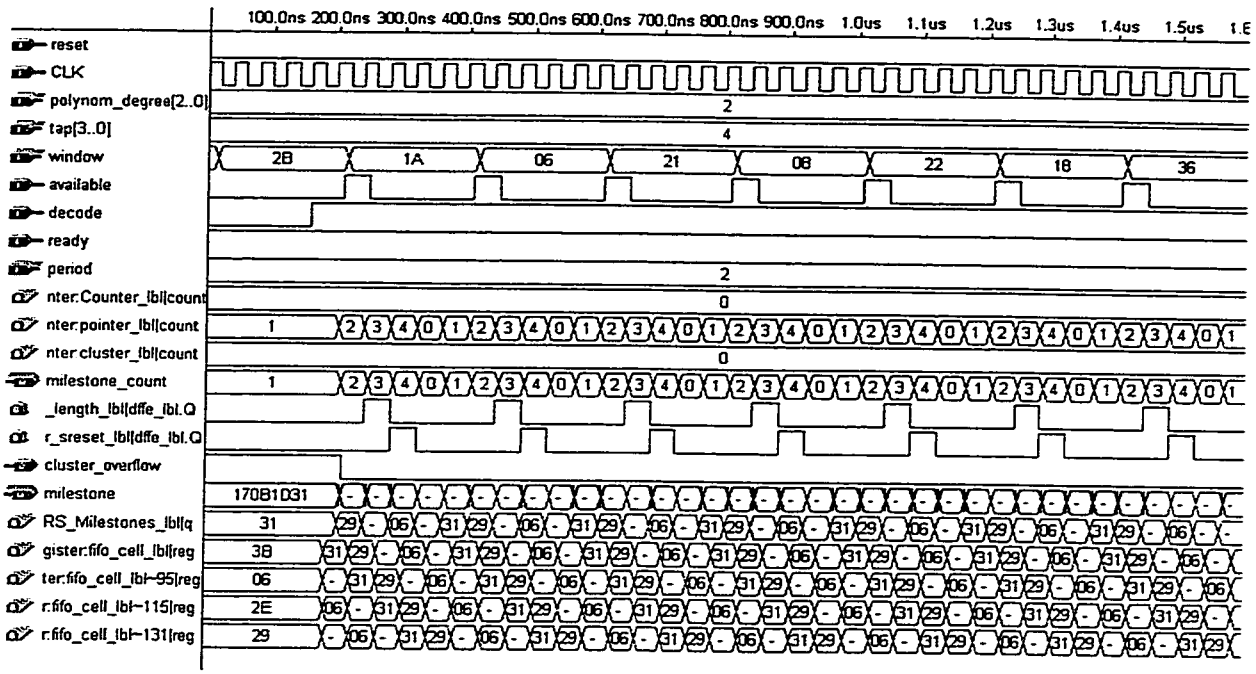


Figure 3-27. Waveform diagram: Milestone Database Unit retrieving data

The VHDL implementation code of the Milestone Database Unit is shown in Annex A-2.

3.2.4 Decoder Unit

The implementation of this unit is straightforward and contains an array of comparator cells running in parallel. This array of comparator cells compares the current PRS window from the PRS Generator Unit against the milestones provided by the Milestone Database Unit.

On a matching condition the Decoder Unit asserts the signal *match* and save into buffers the milestone cluster, index, and offset counters. The content of these counters is used by the μP to calculate the pseudo-random-to-natural code according to the formula shown in Figure 3-2.

The hierarchical architecture of the Decoder Unit is shown in Figure 3-28.

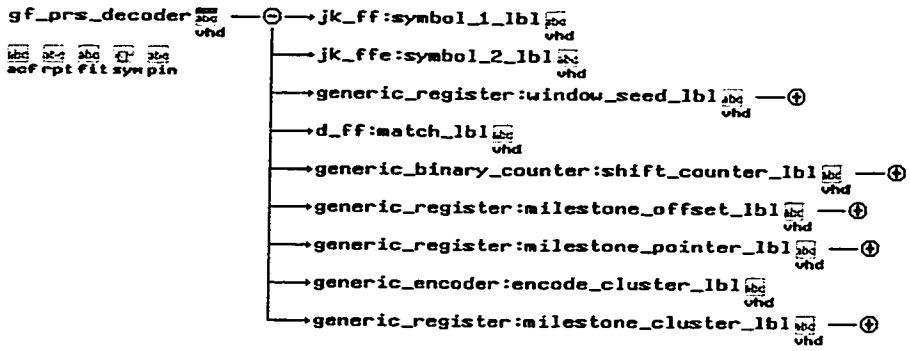


Figure 3-28. Implementation hierarchy of the Decoder Unit

The VHDL implementation code of the Decoder Unit is presented in Annex A-2.

The simulation waveform diagrams of the Decoder Unit are shown in Figure 3-29 and Figure 3-30.

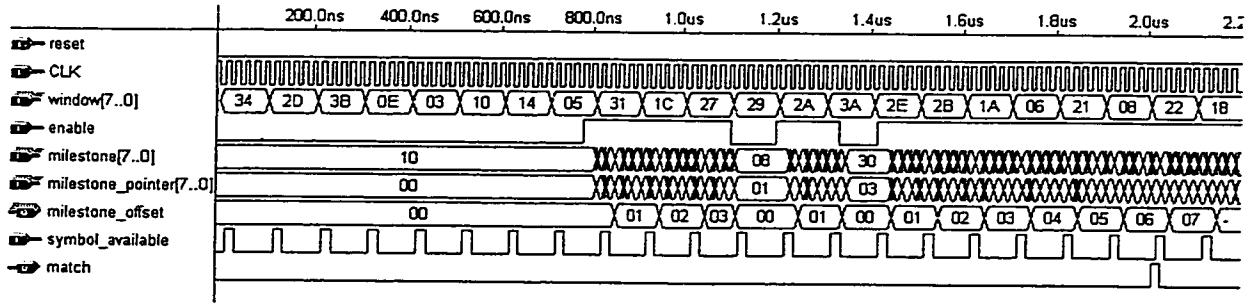


Figure 3-29. Waveform diagram: Decoder Unit matching process

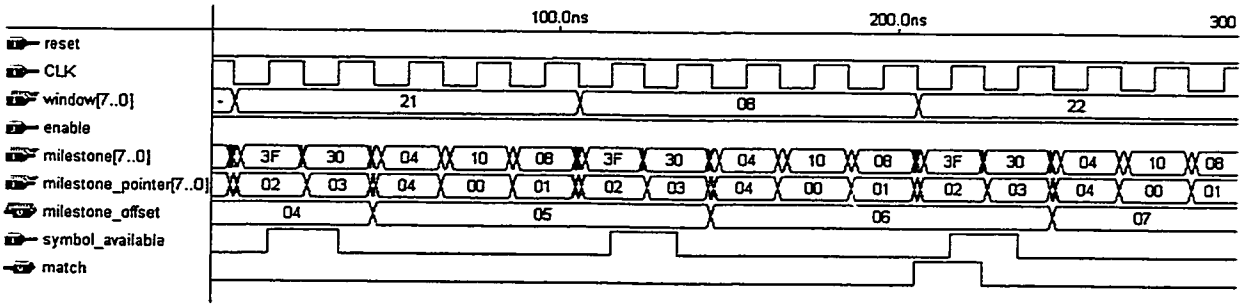


Figure 3-30. Waveform diagram: Detail of the matching process

3.2.5 Input/Output Unit

The I/O Unit synchronizes the external incoming signals with the PRDEC Core internal clock during the writing operation to the PRDEC coprocessor by the microprocessor.

During a reading operation the I/O Unit puts the status register on the μP data bus when the μP reads from address 0, or the current milestone cluster, index, and offset on successive readings from address 1.

The simulation waveform diagrams of the I/O Unit are shown in Figure 3-31.

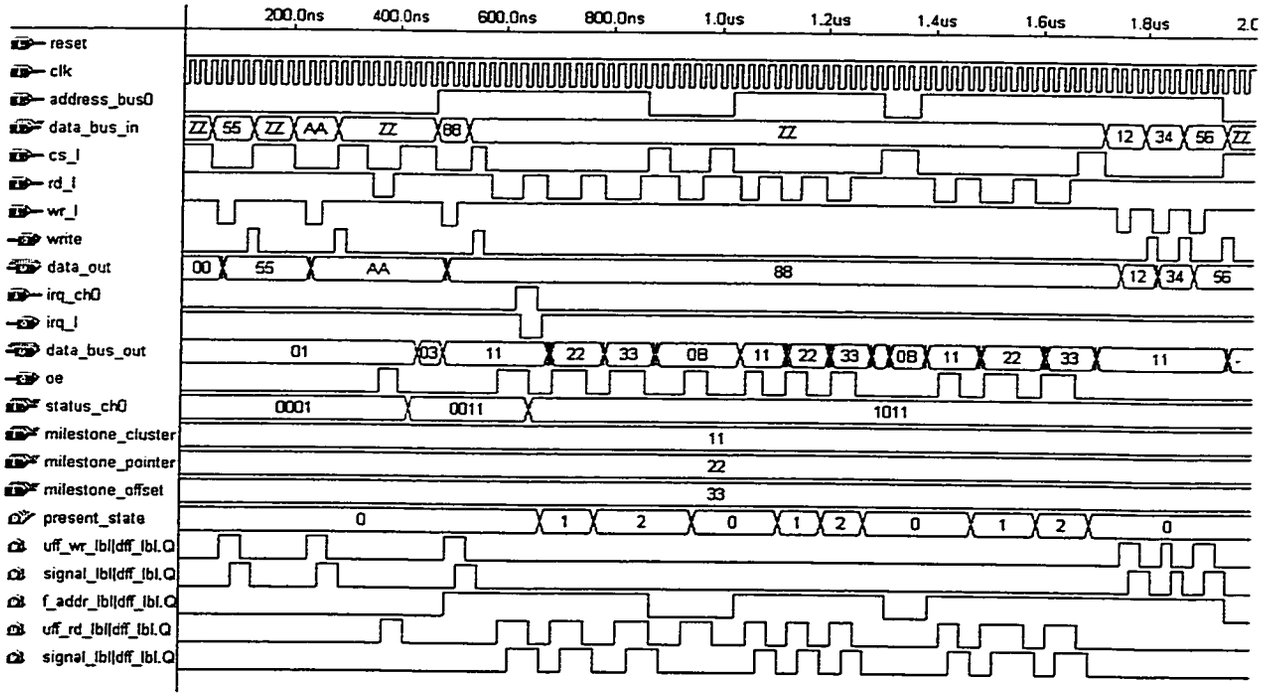


Figure 3-31. Waveform diagram: Input/output operations

The hierarchical architecture of the I/O Unit is shown in Figure 3-32.

The VHDL implementation code of the I/O Unit is presented in Annex A-3.

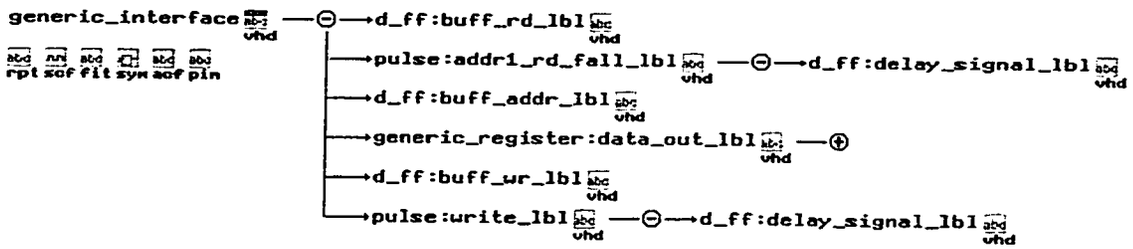


Figure 3-32. Implementation hierarchy of the Input/Output Unit

3.3 Microprocessor Interface Emulator

The PRDEC coprocessor implementation is exercised during the simulation process by a driver that emulates the behavior of a microprocessor interface.

The hierarchical architecture of this driver is shown in Figure 3-33.

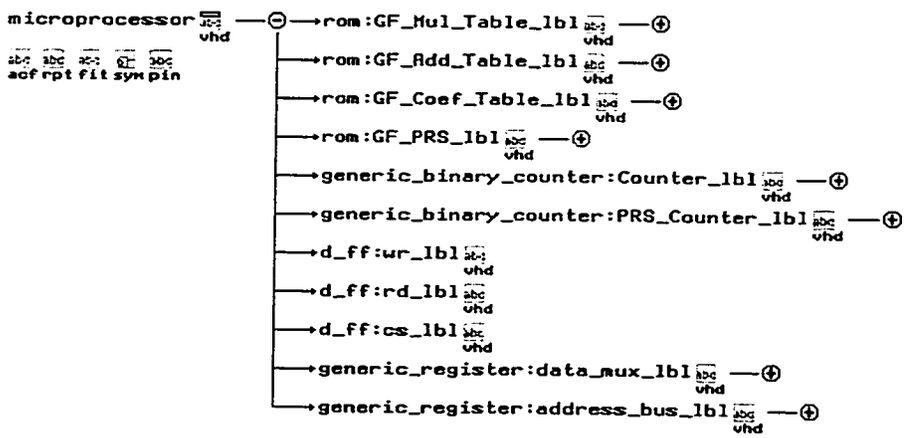


Figure 3-33. Implementation hierarchy of the Microprocessor Interface Driver

The VHDL implementation code of this driver is presented in Annex A-4.

The simulation waveform diagrams of this driver is shown in Figure 3-34.

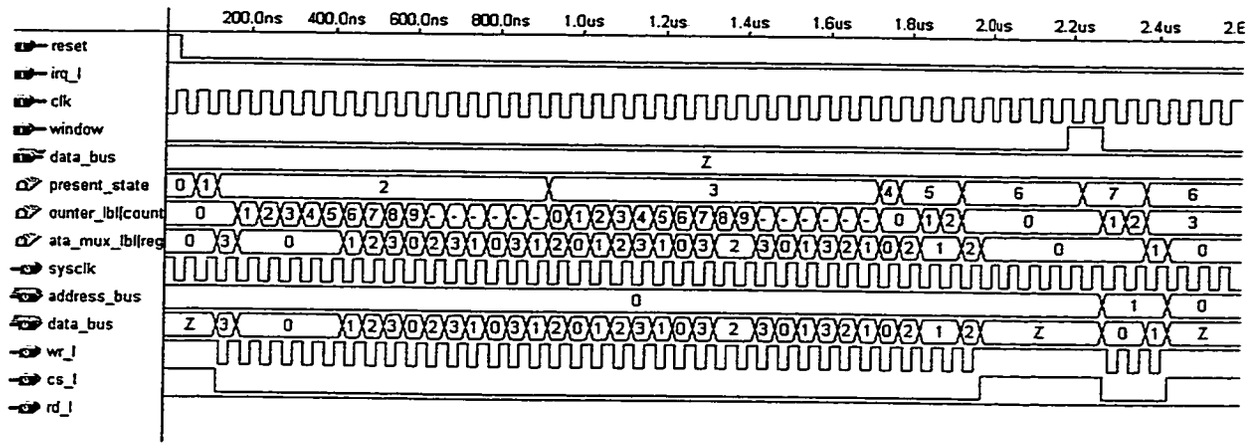


Figure 3-34. Waveform diagram: Microprocessor Interface Driver

3.4 Experimental Results

Following, an in-depth analysis is done for a specific pseudo-random decoding application showing the PRDEC coprocessor implementation corresponding to that application and the PRS extraction from the simulation data. The PRS used by the application is generated by the generator polynomial $h(x)=x^3+x^2+x+2$ defined over the finite field *GF(4)*.

3.4.1 PRDEC Coprocessor Implementation

The PRDEC coprocessor implementation is controlled by the generic parameters shown in Figure 3-35. According to these parameters the synthesized hardware structure of the PRDEC coprocessor is able to accommodate:

- a finite field of maximum 4 elements,
- a generator polynomial of degree 4 maximum, and
- 4 milestone clusters each having 6 milestones.

```
entity Coprocessor is          -- Coprocessor Unit component
   generic (
      ADDRESS_SIZE:            INTEGER:= 2;   -- width of the address bus
      DATA_SIZE:               INTEGER:= 8;   -- width of the data bus
      GF_WIDTH:                INTEGER:= 2;   -- width of GF elements
      DOUBLE_GF_WIDTH:         INTEGER:= 4;   -- width of entry address for
                                              -- GF operation tables = 2*GF_WIDTH
      MAX_POLYNOM_DEGREE:      INTEGER:= 4;   -- maximum degree of the
                                              -- irreducible polynomial
      POLYNOM_DEGREE_WIDTH:INTEGER:= 3;       -- width of MAX_POLYNOM_DEGREE equal
                                              -- to LOG2(MAX_POLYNOM_DEGREE) + 1
                                              -- rounded up to the near integer
      MILESTONE_SIZE:          INTEGER:= 8;   -- the size (in binary bits) of a
                                              -- milestone word saved in FIFO equal
                                              -- to MAX_POLYNOM_DEGREE * GF_WIDTH
      MAX_NUMWORDS:            INTEGER:= 6;   -- maximum number of milestones
                                              -- saved in FIFO equal to
                                              -- MAX_POLYNOM_DEGREE + 2 (latency)
      MILESTONE_CLUSTERS:      INTEGER:= 4;   -- number of milestone clusters
      MILESTONE_CLUSTER_WIDTH:INTEGER:= 2     -- milestone cluster width
                                              -- equal to LOG2(MILESTONE_CLUSTERS)
   );
```

Figure 3-35. Generic implementation parameters of the PRDEC coprocessor

The device summary report shown in Figure 3-36 was obtained on compiling the PRDEC coprocessor with the above parameters for Altera FLEX10K family.

```
** DEVICE SUMMARY **

Chip/                          Input Output Bidir  Memory  Memory          LCs
POF       Device               Pins  Pins   Pins   Bits    %Utilized  LCs  %Utilized

coprocessor
      EPF10K10LC84-3            6     10     8     64        1  %    554      96 %

User Pins:                      6     10     8
```

Figure 3-36. Device summary report

The simulation waveform diagram of the PRDEC coprocessor for the generator polynomial $h(x)=x^3+x^2+x+2$ over the finite field $GF(4)$ is shown in Figures 3-38. When the PRS window seed is "*001*" the LFSR defined by this polynomial generates the PRS shown in Figure 3-37.

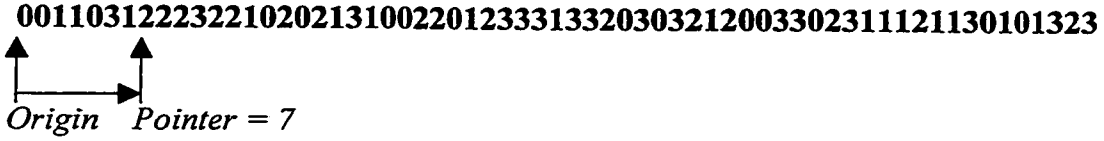
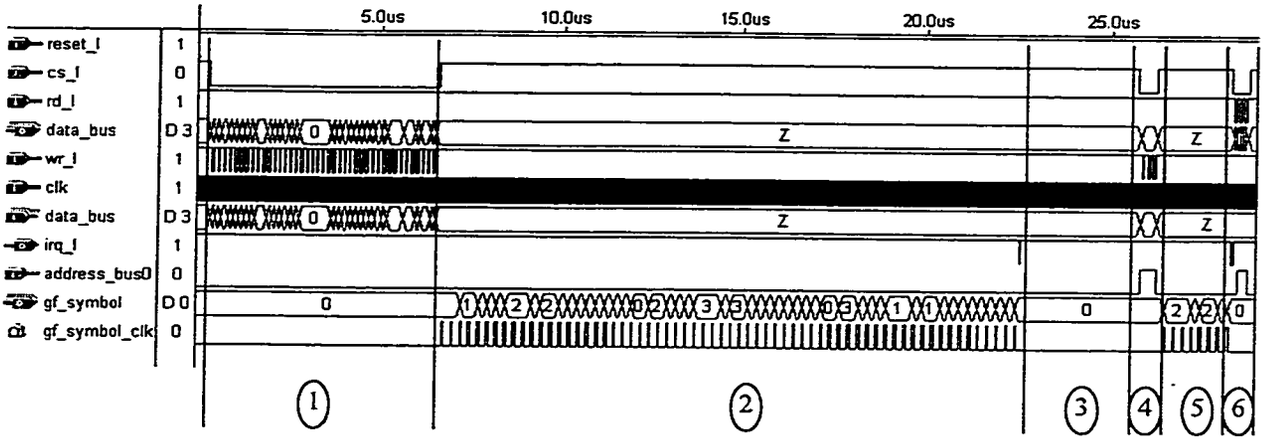


Figure 3-37. PRS extracted visually from the experimental results

The simulation waveform diagram in Figure 3-38 shows a full simulation cycle of the PRDEC coprocessor functionality that resembles very close the PRDEC coprocessor operational flowchart shown in Figure 3-3.



(1) Load algebra corresponding to GF(4) and $h(x)=x^3+x^2+x+2$

(2) Build up the Milestone Database while generating the PRMVS

(3) Idle: waiting for a new PRS window to be loaded for decoding

(4) Load a PRS window for decoding

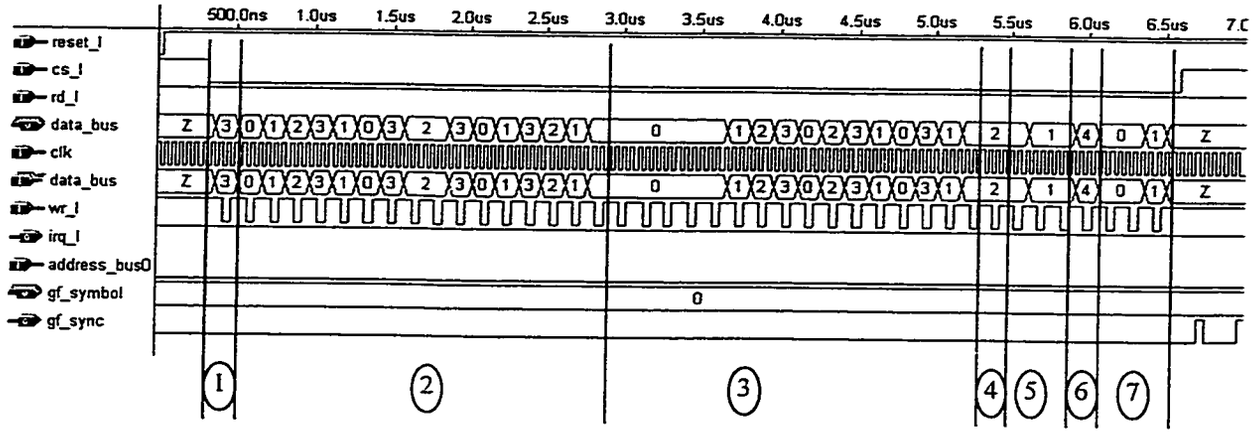
(5) Decode the PRS window

(6) Read out the result of Pseudo-Random-to-Natural Code Conversion

Figure 3-38. Coprocessor simulation

The coprocessor states, shown in the simulation waveform diagram in Figure 3-38, are detailed further in Figures 3-39, 3-40, and 3-41.

The simulation waveform diagram in Figure 3-39 shows the algebra loading operation corresponding to the finite field *GF(4)* described by the addition and multiplication tables shown in Figure 2-4. This waveform diagram resembles very close the step-by-step loading procedure (step 3.a-g) presented in section 3-1.



(1) Load the Number *(q -1) = 3* of Alphabet Symbols corresponding to *GF(q)*

(2) Load the *qxq* entries of the Addition Table

(3) Load the *qxq* entries of the Multiplication Table

(4) Load the Degree *(m-1) = 2* of the Monic Irreducible Polynomial *h(x)*

(5) Load $m = 3$ entries into the Coefficient Table for the monic polynomial $h(x)=x^3+x^2+x+2$

(6) Load the Milestone Period *(p-1)=4*

(7) Load the Milestone PRS Seed: 001

Figure 3-39. Loading algebra corresponding to GF(4) and $h(x)=x^3+x^2+x+2$

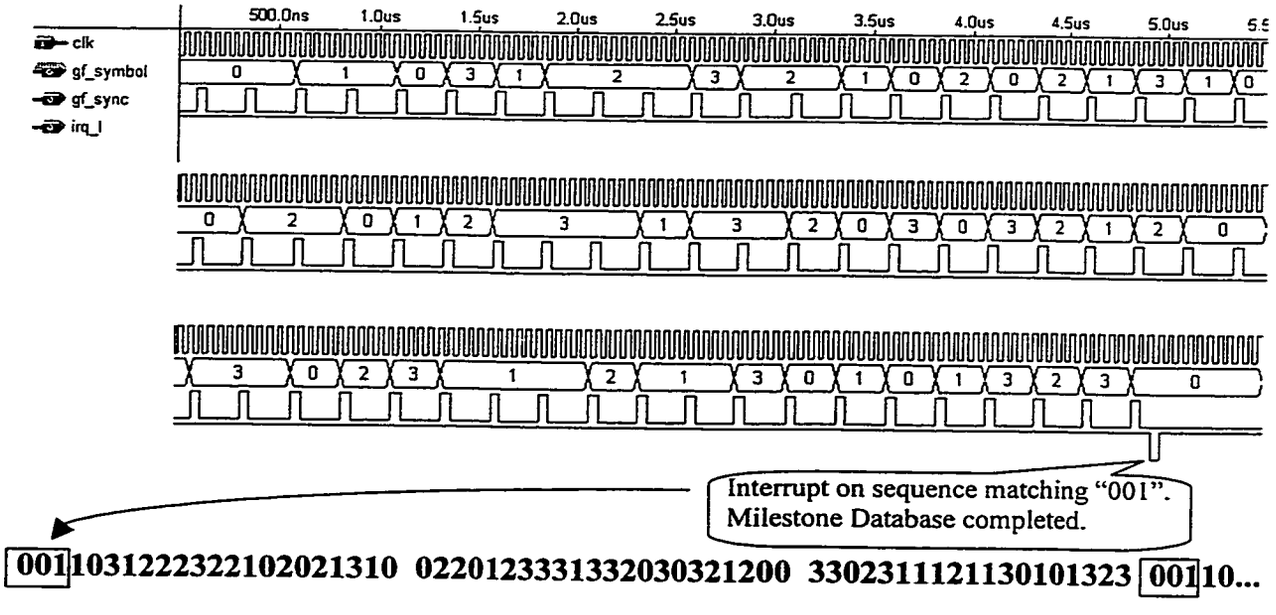


Figure 3-40. Generating the PRMVS of period $4^3-1=63$

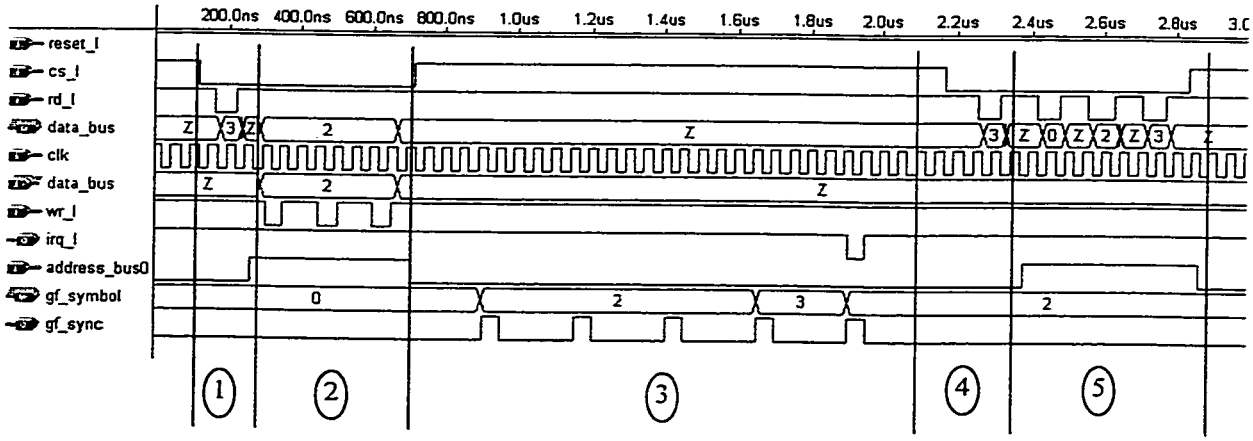
Next, once the algebra parameters have been loaded, the PRDEC coprocessor automatically generates the corresponding PRS and builds its milestone database, as shown in Figure 3-40,. Upon finishing a PRS cycle the PRDEC coprocessor generates an interrupt to let the μP know that the milestone database is completed and it is ready to receive a PRS window for decoding.

Figure 3-41 shows the simulation waveform diagram of the PRDEC coprocessor during the decoding process of the PRS window "*222*". It can be seen from this diagram that at the end of the decoding process the PRDEC coprocessor provides the following pseudo-random-to-natural code results for this PRS window:

milestone cluster = 0,

milestone index = 2,

milestone offset = 3.



(1) Read Status: 3 decimal = 00011 binary (ready = 1, algebra = 1)

(2) Load a PRS window for decoding: "222"

(3) Decode the PRS window

(4) Read Status: 3 decimal = 00011 binary (ready = 1, algebra = 1)

(5) Read out the result of Pseudo-Random-to-Natural Code Conversion
(milestone_cluster = 0, milestone_index = 2, milestone_offset = 3)

window_index = *(milestone_cluster * (polynom_degree+2) + milestone_index) * milestone_period − milestone_offset*
= (0(4+1)+2)*5-3=7*

Figure 3-41. Decoding the PRMVS window "222"

These results are used by the microprocessor to calculate the window index recovery according to the equations presented in Figure 3-2.

Performing the calculus according to these equations the window index obtained is equal to 7 that means that the PRS window "*222*" is located at a distance of 7 symbols from the PRS seed "*001*". This result can be verified visually by analyzing the PRS shown in Figure 3-37 and also the waveform diagram in Figure 3-40.

3.4.2 PRS Extraction Procedure from Simulation Data

The pseudo-random sequences can be extracted automatically from the simulation waveform files generated by the Altera MAX+PLUS II environment by a software program developed in C language whose source code is shown in Annex B. The software program is a specialized text processor which extracts the required information from a formatted text file (generated by the MAX+PLUS II environment) containing the PRS.

The program looks inside the waveform text file for a specific pattern (signal toggling from 1 to 0) on the first waveform and when this pattern is found the second waveform is saved in a separate file. The first waveform always must be the PRS synchronization clock while the second waveform must be the PRS itself,

The procedure to extract the PRS from the simulation files is presented next:

1. Inside the MAX+PLUS II environment save the waveform section showing the generated PRS in a new simulation waveform file (extension .scf), as shown in Figure 3-42.

2. Open this new created simulation file. In the waveform diagram representing this file make sure that the PRS synchronization clock waveform (the signal SYNC) is the first waveform followed by the PRS waveform itself, as shown in Figure 3-43.

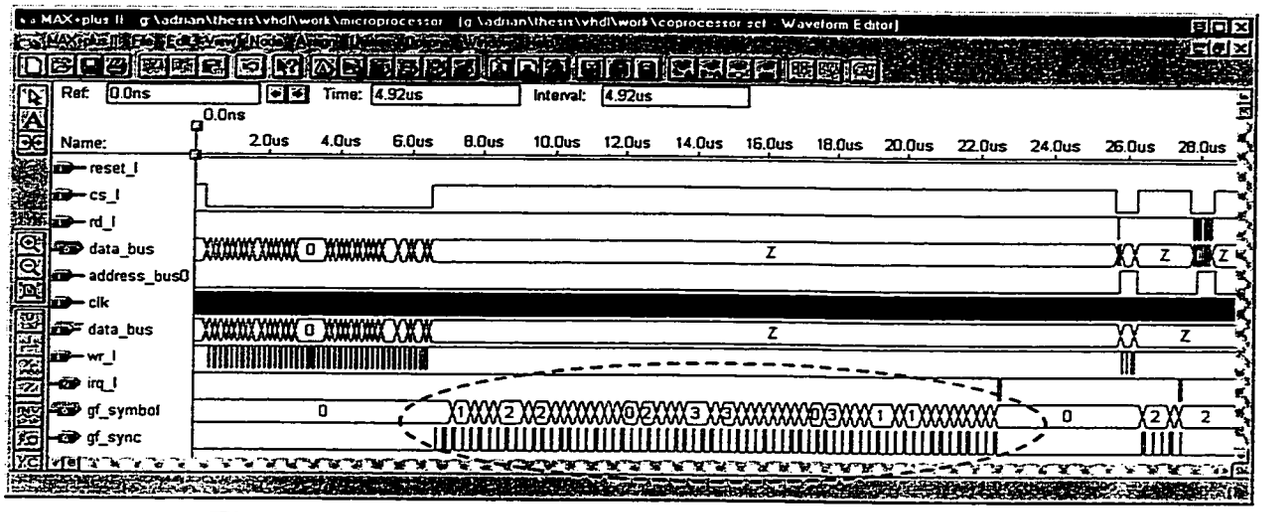


Figure 3-42. Coprocessor simulation waveform diagram

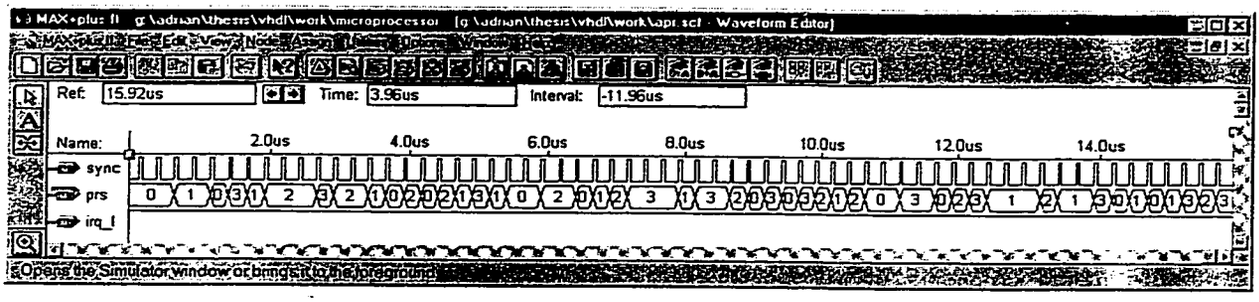


Figure 3-43. Extracted PRS waveform diagram

3. Create a table file (extension .tbl) from this simulation waveform diagram by executing the menu command:

File ⇒ Create Table File...

The new created .tbl file has the following content:

```
%
MAX+plus II 9.0    Date: 02/20/00 12:33:55
File Generated From: g:\adrian\thesis\vhdl\work\PRDEC.scf

Copyright (C) 1988-1998 Altera Corporation
Any megafunction design, and related net list (encrypted or decrypted),
support information, device programming or simulation file, and any other
associated documentation or information provided by Altera or a partner
under Altera's Megafunction Partnership Program may be used only to
program PLD devices (but not masked PLD devices) from Altera.  Any other
use of such megafunction design, net list, support information, device
programming or simulation file, or any other related documentation or
information is prohibited for any other purpose, including, but not
limited to modification, reverse engineering, de-compiling, or use with
any other silicon devices, unless such use is explicitly licensed under
```

a separate agreement with Altera or a megafunction partner. Title to the intellectual property, including patents, copyrights, trademarks, trade secrets, or maskworks, embodied in any such megafunction design, net list, support information, device programming or simulation file, or any other related documentation or information provided by Altera or a megafunction partner, remains with Altera, the megafunction partner, or their respective licensors. No other licenses, including any licenses needed under any third party's intellectual property, are provided herein.
%

```
GROUP CREATE prs = gf_symbol1 gf_symbol0 ;
OUTPUTS sync prs irq_l ;
UNIT ns ;
RADIX HEX ;
PATTERN
%                 i %
%           s     r %
%           y p q %
%           n r _ %
%           c s l %

    0.0> = 0 0 1
  171.2> = 1 0 1
  221.2> = 0 0 1
  421.2> = 1 0 1
  471.2> = 0 0 1
  670.0> = 0 1 1
  671.2> = 1 1 1
  721.2> = 0 1 1
  921.2> = 1 1 1
  971.2> = 0 1 1
 1170.0> = 0 0 1
 1171.2> = 1 0 1
 1221.2> = 0 0 1
 1420.0> = 0 1 1
 1420.1> = 0 3 1
 1421.2> = 1 3 1
 1471.2> = 0 3 1
 1670.1> = 0 1 1
 1671.2> = 1 1 1
 1721.2> = 0 1 1
 1920.0> = 0 0 1
 1920.1> = 0 2 1
 1921.2> = 1 2 1
 1971.2> = 0 2 1
 2171.2> = 1 2 1
 2221.2> = 0 2 1
 2421.2> = 1 2 1
 2471.2> = 0 2 1
 2670.0> = 0 3 1
 2671.2> = 1 3 1
 2721.2> = 0 3 1
 2920.0> = 0 2 1
 2921.2> = 1 2 1
 2971.2> = 0 2 1
 3171.2> = 1 2 1
 3221.2> = 0 2 1
 3420.0> = 0 3 1
 3420.1> = 0 1 1
 3421.2> = 1 1 1
 3471.2> = 0 1 1
 3670.0> = 0 0 1
 3671.2> = 1 0 1
```

```
3721.2> = 0 0 1
3920.1> = 0 2 1
3921.2> = 1 2 1
3971.2> = 0 2 1
4170.1> = 0 0 1
4171.2> = 1 0 1
4221.2> = 0 0 1
4420.1> = 0 2 1
4421.2> = 1 2 1
4471.2> = 0 2 1
4670.0> = 0 3 1
4670.1> = 0 1 1
4671.2> = 1 1 1
4721.2> = 0 1 1
4920.1> = 0 3 1
4921.2> = 1 3 1
4971.2> = 0 3 1
5170.1> = 0 1 1
5171.2> = 1 1 1
5221.2> = 0 1 1
5420.0> = 0 0 1
5421.2> = 1 0 1
5471.2> = 0 0 1
5671.2> = 1 0 1
5721.2> = 0 0 1
5920.1> = 0 2 1
5921.2> = 1 2 1
5971.2> = 0 2 1
6171.2> = 1 2 1
6221.2> = 0 2 1
6420.1> = 0 0 1
6421.2> = 1 0 1
6471.2> = 0 0 1
6670.0> = 0 1 1
6671.2> = 1 1 1
6721.2> = 0 1 1
6920.0> = 0 0 1
6920.1> = 0 2 1
6921.2> = 1 2 1
6971.2> = 0 2 1
7170.0> = 0 3 1
7171.2> = 1 3 1
7221.2> = 0 3 1
7421.2> = 1 3 1
7471.2> = 0 3 1
7671.2> = 1 3 1
7721.2> = 0 3 1
7920.1> = 0 1 1
7921.2> = 1 1 1
7971.2> = 0 1 1
8170.1> = 0 3 1
8171.2> = 1 3 1
8221.2> = 0 3 1
8421.2> = 1 3 1
8471.2> = 0 3 1
8670.0> = 0 2 1
8671.2> = 1 2 1
8721.2> = 0 2 1
8920.1> = 0 0 1
8921.2> = 1 0 1
8971.2> = 0 0 1
9170.0> = 0 1 1
9170.1> = 0 3 1
```

```
 9171.2> = 1 3 1
 9221.2> = 0 3 1
 9420.0> = 0 2 1
 9420.1> = 0 0 1
 9421.2> = 1 0 1
 9471.2> = 0 0 1
 9670.0> = 0 1 1
 9670.1> = 0 3 1
 9671.2> = 1 3 1
 9721.2> = 0 3 1
 9920.0> = 0 2 1
 9921.2> = 1 2 1
 9971.2> = 0 2 1
10170.0> = 0 3 1
10170.1> = 0 1 1
10171.2> = 1 1 1
10221.2> = 0 1 1
10420.0> = 0 0 1
10420.1> = 0 2 1
10421.2> = 1 2 1
10471.2> = 0 2 1
10670.1> = 0 0 1
10671.2> = 1 0 1
10721.2> = 0 0 1
10921.2> = 1 0 1
10971.2> = 0 0 1
11170.0> = 0 1 1
11170.1> = 0 3 1
11171.2> = 1 3 1
11221.2> = 0 3 1
11421.2> = 1 3 1
11471.2> = 0 3 1
11670.0> = 0 2 1
11670.1> = 0 0 1
11671.2> = 1 0 1
11721.2> = 0 0 1
11920.1> = 0 2 1
11921.2> = 1 2 1
11971.2> = 0 2 1
12170.0> = 0 3 1
12171.2> = 1 3 1
12221.2> = 0 3 1
12420.1> = 0 1 1
12421.2> = 1 1 1
12471.2> = 0 1 1
12671.2> = 1 1 1
12721.2> = 0 1 1
12921.2> = 1 1 1
12971.2> = 0 1 1
13170.0> = 0 0 1
13170.1> = 0 2 1
13171.2> = 1 2 1
13221.2> = 0 2 1
13420.0> = 0 3 1
13420.1> = 0 1 1
13421.2> = 1 1 1
13471.2> = 0 1 1
13671.2> = 1 1 1
13721.2> = 0 1 1
13920.1> = 0 3 1
13921.2> = 1 3 1
13971.2> = 0 3 1
14170.0> = 0 2 1
```

```
14170.1> = 0 0 1
14171.2> = 1 0 1
14221.2> = 0 0 1
14420.0> = 0 1 1
14421.2> = 1 1 1
14471.2> = 0 1 1
14670.0> = 0 0 1
14671.2> = 1 0 1
14721.2> = 0 0 1
14920.0> = 0 1 1
14921.2> = 1 1 1
14971.2> = 0 1 1
15170.1> = 0 3 1
15171.2> = 1 3 1
15221.2> = 0 3 1
15420.0> = 0 2 1
15421.2> = 1 2 1
15471.2> = 0 2 1
15670.0> = 0 3 1
15671.2> = 1 3 1
15721.2> = 0 3 1
15920.0> = X X X;
```

4. Execute the PRS extractor program with the table file as the first parameter. The second parameter should be the name of the destination file where the PRS will be saved as ASCII text along with the number of the symbols contained in the sequence.

prs_extractor source.tbl dest.prs

Figure 3-44 shows a MS-DOS screenshot during executing the extract PRS command where the name of the generated table file is **apr.tbl**

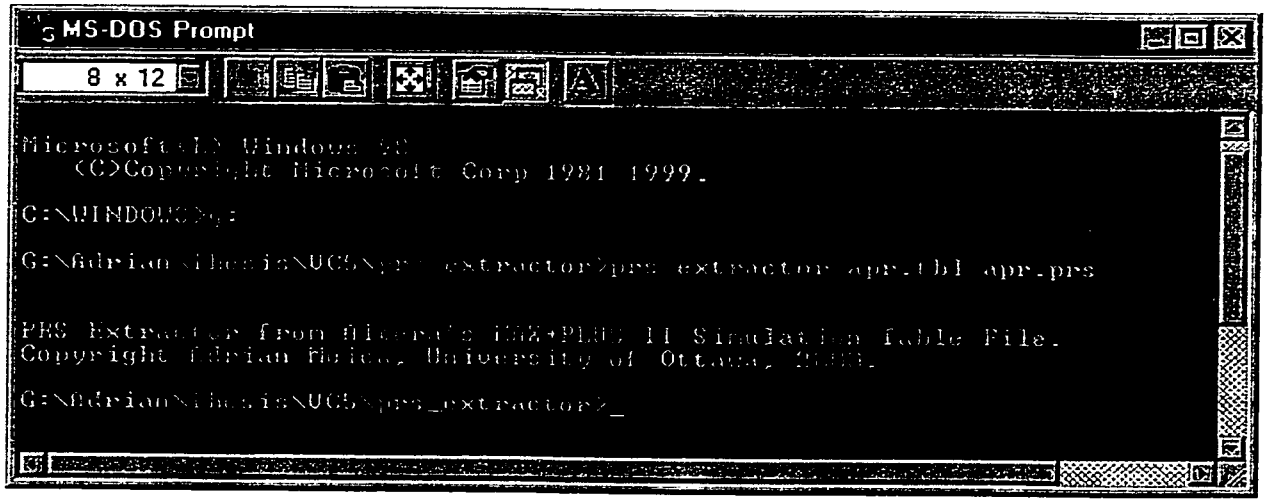


Figure 3-44. Screenshot during executing Extract PRS command

The resulted file, named **apr.prs**, has the following content:

```
0 0 1 1 0 3 1 2 2 2 3 2 2 1 0 2 0 2 1 3 1 0 0
2 2 0 1 2 3 3 3 1 3 3 2 0 3 0 3 2 1 2 0 0 3 3
0 2 3 1 1 1 2 1 1 3 0 1 0 1 3 2 3
length = 63
```

and represents the PRS generated by the irreducible polynomial $h(x)=x^3+x^2+x+2$ defined over the finite field *GF(4)* with the PRS window seed "*001*".

3.5 More Experimental Results

This section contains a number of experimental results for the window index recovery on various finite fields and generator polynomials. The window coordinates with respect to the milestones are generated by the PRDEC coprocessor and the window index recovery is computed using the equations presented in Figure 3-2.

3.5.1 Finite field GF(2)

The finite field *GF(2)* has the order $q = p^m = 2^1 = 2$ where the prime order $p = 2$, and the positive integer $m = 1$.

```
ADDRESS_SIZE:          INTEGER:= 2;   -- width of the address bus
DATA_SIZE:             INTEGER:= 8;   -- width of the data bus
GF_WIDTH:              INTEGER:= 1;   -- width of GF elements
DOUBLE_GF_WIDTH:       INTEGER:= 2;   -- width of entry address for
                                      -- GF operation tables = 2*GF_WIDTH
MAX_POLYNOM_DEGREE:    INTEGER:= 10;  -- maximum degree of the
                                      -- irreducible polynomial
POLYNOM_DEGREE_WIDTH:INTEGER:= 4;     -- width of MAX_POLYNOM_DEGREE equal
                                      -- to LOG2(MAX_POLYNOM_DEGREE) + 1
                                      -- rounded up to the near integer
MILESTONE_SIZE:        INTEGER:= 10;  -- the size (in binary bits) of a
                                      -- milestone word saved in FIFO equal
                                      -- to MAX_POLYNOM_DEGREE * GF_WIDTH
MAX_NUMWORDS:          INTEGER:= 12;  -- maximum number of milestones
                                      -- saved in FIFO equal to
                                      -- MAX_POLYNOM_DEGREE + 2 (latency)
MILESTONE_CLUSTERS:    INTEGER:= 4;   -- number of milestone clusters
MILESTONE_CLUSTER_WIDTH:INTEGER:= 2   -- milestone cluster width
                                      -- equal to LOG2(MILESTONE_CLUSTERS)
```

Figure 3-45. Implementation parameters for GF(2)

The PRDEC coprocessor was synthesized to accommodate:

- a binary LFSR described by a generator polynomial of maximum 10^{th} degree, and
- a milestone database composed of (4 clusters) x (12 words) x (10 bits);

The generic parameters for the chosen implementation are shown in Figure 3-45, while the synthesizing results for this implementation are shown in Figure 3-46.

```
** DEVICE SUMMARY **

Chip/                          Input Output Bidir  Memory  Memory          LCs
POF       Device               Pins  Pins   Pins   Bits    %Utilized  LCs  %Utilized

coprocessor
      EPF10K20TC144-3          6     10     8      8          0 %    889      77 %

User Pins:                     6     10     8
```

Figure 3-46. Synthesis result for GF(2) implementation

Generator Polynomial of 4^{th} degree

The generator polynomial $h(x) = x^4+x^3+1$ of degree $m = 4$ has the feedback equation represented by $a_{k+4} = a_{k+3} + a_k$ and generates a PRBS of length $n = q^m-1 = 2^4-1 = 15$.

0 0 0 1 1 **1** 1 0 1 0 1 **1 0 0 1**
length = 15

The following table shows several experimental results where the milestone period p = 3.

PRBS window	Milestone			Computed Index Recovery
	cluster	index	offset	
0001	0	0	0	0
1101	0	2	1	5
1001	0	4	1	11
0010	0	4	0	12
1000	0	0	1	14

Table 3-6. Experimental results for GF(2) and a 4^{th} degree polynomial

The tested PRBS windows shown in the above experimental results table are marked in bold face while their beginning are marked by underscore.

Generator Polynomial of 5th degree

The generator polynomial $h(x) = x^5+x^3+1$ of degree $m = 5$ has the feedback equation represented by $a_{k+5} = a_{k+3} + a_k$ and generates a PRBS of length $n = q^m-1 = 2^5-1 = 31$.

0 0 0 0 1 **0** 1 0 1 1 1 0 **1** 1 0 0 0 **1** 1 1 1 1 0 0 **1** 1 0 **1** 0 0 1
length = 31

The following table shows several experimental results where the milestone period p = 5.

PRBS window	Milestone			Computed Index Recovery
	cluster	index	offset	
11000	0	3	3	12
11111	0	4	3	17
10010	0	6	3	27
11010	0	5	1	24
01011	0	1	0	5

Table 3-7. Experimental results for GF(2) and a 5th degree polynomial

Generator Polynomial of 6th degree

The generator polynomial $h(x) = x^6+x^5+x^2+x+1$ of degree $m = 6$ has the feedback equation represented by $a_{k+6} = a_{k+4} + a_{k+3} + a_{k+1} + a_k$ and generates a maximum length PRBS of $n = q^m-1 = 2^6-1 = 63$ symbols.

0 **0** 0 0 0 1 1 1 1 **0** 0 **1** 0 0 1 **0** 1 0 1 0 0 1 1 0 **1** 0 0 0 0 **1** 0 0 0 **1** 0 **1** 1 0 1 1
1 1 **1** 1 **0** 1 **0** **1** 1 1 0 0 0 1 1 0 0 1 1 1 0 1 1
length = 63

The following table shows several experimental results where the milestone period p = 3.

PRBS window	Milestone			Computed Index Recovery
	cluster	index	offset	
000101	1	2	0	30
110101	1	6	0	42
100101	0	4	1	11
001001	0	3	0	9
100001	1	0	0	24

Table 3-8. Experimental results for GF(2) and a 6th degree polynomial

Generator Polynomial of 7th degree

The generator polynomial $h(x) = x^7 + x^5 + x^4 + x^3 + 1$ of degree $m = 7$ has the feedback equation represented by $a_{k+7} = a_{k+5} + a_{k+4} + a_{k+3} + a_k$ and generates a maximum length PRBS of $n = q^m - 1 = 2^7 - 1 = 127$ symbols.

0 0 0 **0 0 0 1 0 1 1** 0 0 0 0 0 1 1 1 **0 1 0 0 0 0 1** 0 0 1 1 1 0 0 0 1 1 0 1 0 0 1
0 0 1 0 1 1 1 0 1 1 0 1 1 1 0 0 1 1 0 1 **1 0 0 1 0 1 0 1 1 0 1 0 1 1** 1 1 1 0 1 1
1 1 0 0 0 0 1 1 0 **0 0 1 0 0 0 1** 0 1 0 0 1 1 0 0 1 1 1 1 0 1 0 1 0 1 0 0 0 1 1 1
1 1 1 1 0 0 1
length = 127

The following table shows several experimental results where the milestone period p = 5.

PRBS window	Milestone			Computed Index Recovery
	cluster	index	offset	
0001011	0	1	2	3
1101011	1	5	3	67
1001010	1	3	0	60
0010001	2	0	1	89
0100001	0	4	2	18

Table 3-9. Experimental results for GF(2) and a 7th degree polynomial

Generator Polynomial of 8th degree

The generator polynomial $h(x) = x^8 + x^7 + x^6 + x^3 + x^2 + x + 1$ of degree $m = 8$ has the feedback equation represented by $a_{k+8} = a_{k+7} + a_{k+6} + a_{k+3} + a_{k+2} + a_{k+1} + a_k$ and generates a maximum length PRBS of $n = q^m - 1 = 2^8 - 1 = 255$ symbols.

0 0 0 0 0 0 0 1 1 0 1 1 1 0 1 0 0 1 0 0 0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 0 1 0 1 0
0 1 1 0 1 0 1 1 0 0 0 0 1 1 0 0 0 1 1 0 1 0 0 0 0 1 1 1 1 0 0 0 1 1 1 0 0 1 1 1
1 0 1 1 1 0 0 1 0 0 1 1 0 0 1 1 0 1 1 0 1 1 1 1 0 0 1 0 1 0 1 1 1 0 1 1 1 1 1 1
1 1 0 1 1 0 1 0 0 1 1 1 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 1 1 0 0 1 1 1 0 1 1 0
0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 1 1 1 **1 1 0 1 0 1 1 1** 1 0 1 0 0 0 1 **0 1 0 0 1 0 1**
1 1 0 0 0 1 0 0 0 1 0 0 1 0 **0 1 0 1 0 0 0 1** 1 0 0 1 0 1 **1 0 1 0 1 0 1 0** 1 1 0 1
1 0 0 0 1 0 1 1 1 1 1 1 0 0 1 1
length = 255

The following table shows several experimental results where the milestone period p = 7.

PRBS window	Milestone			Computed Index Recovery
	cluster	index	offset	
01001011	2	8	3	193
11010111	2	6	4	178
10101010	3	3	3	228
00010001	2	9	1	202
01010001	3	1	3	214

Table 3-10. Experimental results for GF(2) and a 8th degree polynomial

Generator Polynomial of 9th degree

The generator polynomial $h(x) = x^9+x^8+x^4+x+1$ of degree $m = 9$ has the feedback equation represented by $a_{k+9} = a_{k+8} + a_{k+4} + a_{k+1} + a_k$ and generates a maximum length PRBS of $n = q^m-1 = 2^9-1 = 511$ symbols.

0 0 0 0 0 0 0 0 1 1 1 1 1 0 1 0 0 1 1 0 0 1 1 1 1 0 1 1 0 0 **0 1 0 1 0 0 0 0 1** 0
1 0 1 0 0 1 0 0 1 0 0 1 1 1 **0 0 0 0 1 0 0 0 1** 0 0 0 1 1 1 1 0 0 1 0 0 0 0 1 0 0
1 0 1 1 1 1 0 1 1 1 1 1 0 1 1 1 0 0 1 0 0 1 1 0 1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 0
0 1 0 0 0 0 1 1 1 0 1 0 1 0 1 0 0 0 1 1 0 1 0 0 1 1 1 1 0 0 0 1 1 1 1 1 1 0 1 1
0 1 1 0 1 0 0 1 0 0 0 1 1 0 0 1 1 0 **0 1 0 0 0 1 0 1 1** 0 0 0 1 1 0 1 1 1 0 0 0 1
1 0 0 0 0 1 1 0 0 1 0 1 1 0 1 0 1 1 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 1 1 0 0 0 1
0 0 1 1 1 1 1 1 1 0 0 1 0 1 1 1 0 1 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0 1 1 0 1 1 1 0
1 1 0 0 1 0 1 0 1 0 1 0 1 1 0 0 1 1 1 0 1 0 0 1 0 1 1 0 0 1 0 0 1 0 1 0 0 0 1 0
1 0 1 1 1 0 1 1 1 1 0 1 0 0 0 0 1 1 0 1 0 1 0 0 0 0 0 1 0 1 0 0 1 1 0 1 0 0 0 0
0 0 1 0 1 1 1 0 0 1 1 1 0 0 1 1 0 1 1 0 0 0 0 1 0 1 1 0 1 1 0 0 1 1 0 1 0 1 1 1
1 1 1 1 1 1 0 1 0 1 0 0 1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 1 1 1 0 0 1 1 1 1 1 0 0 1 1
0 0 0 1 1 1 0 0 0 1 0 1 1 1 1 1 0 0 0 0 1 1 1 1 0 1 0 1 1 1 0 0 0 0 0 1 1 1 0 0
1 **0 1 0 0 1 0 1 0** 1 1 0 1 0 0 0 1 1 1 0 1 **1** **0 1 0 1 0 1 0 1 1 1** 1
length = 511

The following table shows several experimental results where the milestone period p=15.

PRBS window	Milestone			Computed Index Recovery
	cluster	index	offset	
010001011	1	1	2	178
101010111	3	1	9	501
101001010	2	10	0	480
000010001	0	4	6	54
010100001	0	2	0	30

Table 3-11. Experimental results for GF(2) and a 9th degree polynomial

Generator Polynomial of 10th degree

$h(x) = x^{10}+x^9+x^8+x^7+x^3+x^2+1 \quad \Rightarrow \quad a_{k+10} = a_{k+9} + a_{k+8} + a_{k+7} + a_{k+3} + a_{k+2} + a_k$

Polynomial degree $m = 10$. PRBS of maximum length $n = q^m-1 = 2^{10}-1 = 1023$ symbols.

0 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 1 1 0 1 0 1 1 0 0 1
1 0 1 1 1 0 0 1 0 1 **1 0 0 0 0 1 0 0 0 1** 0 1 1 1 1 0 1 1 1 0 0 0 0 0 1 1 0 1 1 0
0 0 0 1 1 1 0 **1 1 0 1 0 1 0 1 1 1** 1 1 0 0 0 1 1 1 0 0 0 1 0 1 0 1 1 1 1 0 1 1 0 1
1 0 0 1 0 0 1 0 0 0 0 0 1 1 0 1 0 0 1 0 1 1 0 0 1 1 1 0 1 1 1 1 1 1 1 1 0 0 0 1 0
0 0 0 1 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1 1 0 1 1 1 1 1 0 0 1 0 1 1 1 1 0 0
0 1 0 1 1 0 1 1 1 0 1 1 1 1 0 0 1 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 1 1 1 1 1 1 1 0 0 1
1 1 0 1 0 0 1 1 0 0 1 1 0 0 1 0 0 1 1 1 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 1 1 0 0 0
0 1 0 1 0 0 0 0 1 0 0 1 1 1 1 0 1 0 1 0 1 1 0 0 1 0 1 1 1 0 1 0 0 0 1 1 0 0 1 1
1 0 0 0 1 1 0 0 1 0 0 0 0 1 0 1 1 0 1 0 0 0 1 0 0 1 1 0 1 1 1 1 1 1 0 0 0 0 1 0
0 1 0 0 1 0 0 1 1 1 1 1 0 0 1 1 0 1 1 0 1 0 1 1 0 1 1 0 0 0 1 0 1 1 1 0 1 1 1 1 0
1 0 1 0 0 1 1 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 1 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 0 1 0 0 1
0 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1 1 1 0 1 1 0 0 1 1 1 0 0 0 1 0 1 0 0 0 1
0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 1 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0 1 0 0 1
0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 0 0 0 1 1 1 1 0 0 0 1 1 0 1 0 0 0 1 1 1 1 1 1
0 1 0 1 1 1 0 0 1 1 1 0 0 1 0 1 0 1 0 0 1 0 0 1 0 1 0 1 1 0 1 0 0 1 1 1 0 1 1 1 0
0 1 1 0 1 0 1 1 0 0 0 0 0 1 0 0 0 1 1 0 1 1 0 1 1 0 1 0 0 0 1 0 1 0 1 1 0 1 1
1 1 0 1 1 0 1 0 0 1 1 1 0 0 1 1 0 0 1 1 1 1 1 0 1 0 1 0 0 0 0 0 0 1 1 0 1 0 1 0
0 1 0 1 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1 1 0 0 1 0 1 0 0 0 0 1 1 1 0 0 1 0 0 1 1 0 1
0 0 1 1 0 1 0 1 0 1 0 0 1 1 1 1 0 0 1 1 0 0 0 0 1 1 0 1 1 1 1 0 0 0 0 0 1 1 1 1 0
1 0 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1 0 0 1
0 0 0 1 0 0 1 0 1 1 1 0 0 1 0 0 0 0 0 1 1 1 1 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1 1 0 1
1 0 1 1 1 1 1 0 1 0 0 1 0 0 1 1 0 0 1 0 1 0 1 1 1 1 0 1 0 0 0 0 1 1 0 0 1 0 1 1
0 1 1 0 1 1 1 0 0 0 1 1 1 1 1 0 1 1 0 0 0 1 1 0 1 **1 1 0 1 0 0 1 0 1 0 1 0 1 0 0**
0 0 1 1 1 1 1 0 0 0 0 0 1 0 1 1 1 1 1 0 1 1 1 1 0 1 0 1 1 0 **1** 0 1 0 0 0 1 1 0 1
0 1 1 1 1 0 0 1 0 0 1 0 1 1 0 1 0 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 0 0 0 1 0 **1 0 1**
0 0 0 1 0 1 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 0 1
length = 1023

The following table shows several experimental results where the milestone period p=25.

PRBS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
1010001011	3	4	3	997
1101010111	0	4	13	87
1101001010	3	1	20	905
1000010001	0	2	0	50
1010100001	3	1	12	913

Table 3-12. Experimental results for GF(2) and a 10^{th} degree polynomial

3.5.2 Finite field GF(3)

The finite field *GF(3)* has the order $q = p^m = 3^1 = 3$ where the prime order $p = 3$, and the positive integer $m = 1$.

```
ADDRESS_SIZE:          INTEGER:= 2;   -- width of the address bus
DATA_SIZE:             INTEGER:= 8;   -- width of the data bus
GF_WIDTH:              INTEGER:= 2;   -- width of GF elements
DOUBLE_GF_WIDTH:       INTEGER:= 4;   -- width of entry address for
                                      -- GF operation tables = 2*GF_WIDTH
MAX_POLYNOM_DEGREE:    INTEGER:= 6;   -- maximum degree of the
                                      -- irreducible polynomial
POLYNOM_DEGREE_WIDTH:INTEGER:= 3;     -- width of MAX_POLYNOM_DEGREE equal
                                      -- to LOG2(MAX_POLYNOM_DEGREE) + 1
                                      -- rounded up to the near integer
MILESTONE_SIZE:        INTEGER:= 12;  -- the size (in binary bits) of a
                                      -- milestone word saved in FIFO equal
                                      -- to MAX_POLYNOM_DEGREE * GF_WIDTH
MAX_NUMWORDS:          INTEGER:= 8;   -- maximum number of milestones
                                      -- saved in FIFO equal to
                                      -- MAX_POLYNOM_DEGREE + 2 (latency)
MILESTONE_CLUSTERS:    INTEGER:= 4;   -- number of milestone clusters
MILESTONE_CLUSTER_WIDTH:INTEGER:= 2   -- milestone cluster width
                                      -- equal to LOG2(MILESTONE_CLUSTERS)
```

Figure 3-47. Implementation parameters for GF(3)

The PRDEC coprocessor was synthesized to accommodate:

- a 3-value LFSR described by a generator polynomial of maximum 6^{th} degree, and
- a milestone database composed of (4 clusters) x (8 words) x (12 bits);

The generic parameters for the chosen implementation are shown in Figure 3-47, while the synthesizing results for this implementation are shown in Figure 3-48.

```
** DEVICE SUMMARY **

Chip/                          Input Output Bidir  Memory  Memory          LCs
POF        Device              Pins  Pins   Pins   Bits    %Utilized  LCs  %Utilized

coprocessor
      EPF10K20TC144-3          6     10     8      64        0 %     809      70 %

User Pins:                     6     10     8
```

Figure 3-48. Synthesizing results for GF(3) implementation

Generator Polynomial of 2nd degree

The generator polynomial $h(x) = x^2+2x+2$ of degree $m = 2$ has the feedback equation represented by $a_{k+2} = a_{k+1} + a_k$ and generates a PRMVS of length $n = q^m-1 = 3^2-1 = 8$.

0 1 1 2 0 2 2 1
length = 8

The following table shows several experimental results where the milestone period p = 4.

PRMVS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
10	0	0	1	7
12	0	1	2	2
22	0	0	3	5
20	0	1	1	3
11	0	1	3	1

Table 3-13. Experimental results for GF(3) and a 2nd degree polynomial

The tested PRMVS windows shown in the above experimental results table are marked in bold face while their beginning are marked by underscore.

Generator Polynomial of 3rd degree

The generator polynomial $h(x) = x^3+2x+1$ of degree $m = 3$ has the feedback equation represented by $a_{k+3} = a_{k+1} + 2a_k$ and generates a PRMVS of length $n = q^m-1 = 3^3-1 = 26$.

0 0 **1** **0** **1** 2 **1** **1** **2** **0** 1 1 1 0 0 2 0 2 **1** **2** **2** 1 0 2 **2** **2** **0**

length = 27

The following table shows several experimental results where the milestone period p = 4.

PRMVS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
101	0	1	2	2
122	1	0	2	18
220	1	1	0	24
201	0	2	0	8
112	0	2	2	6

Table 3-14. Experimental results for GF(3) and a 3rd degree polynomial

Generator Polynomial of 4th degree

The generator polynomial $h(x) = x^4+x+2$ of degree $m = 4$ has the feedback equation represented by $a_{k+4} = 2a_{k+1} + a_k$ and generates a PRMVS of length $n = q^m-1 = 3^4-1 = 80$.

0 0 0 1 0 0 2 1 0 1 1 1 2 0 **0** **2** **2** **0** 1 0 2 2 **1** **1** **0** **1** 0 1 **2** **1** **2** **2** **1** **2** **0** **1** 2 2 2 2
0 0 0 2 0 0 1 2 0 2 2 2 1 0 0 1 1 0 2 0 1 1 2 2 0 2 0 2 1 **2** **1** **1** **2** 1 0 2 1 1 1 1
length = 80

The following table shows several experimental results where the milestone period p = 4.

PRMVS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
1101	1	0	2	22
2122	1	1	0	28
0220	0	4	2	14
1201	1	2	0	32
2112	3	0	3	69

Table 3-15. Experimental results for GF(3) and a 4th degree polynomial

Generator Polynomial of 5th degree

The generator polynomial $h(x) = x^5+2x+1$ of degree $m = 5$ has the feedback equation represented by $a_{k+5} = a_{k+1} + 2a_k$ and generates a PRMVS of length $n=q^m-1 = 3^5-1=242$.

0 0 0 0 1 0 0 0 1 2 0 0 1 1 1 0 1 0 0 2 1 2 0 **2 2 1 1 2** 0 2 0 1 1 **2 1 1 0 1** 2 0
2 1 1 1 2 2 0 0 1 0 1 0 1 2 1 2 1 1 2 1 2 0 1 2 1 1 1 1 2 0 0 0 1 1 0 0 1 0 2 0
1 2 2 1 1 1 0 2 0 0 2 2 1 0 2 0 2 2 2 1 2 0 0 2 1 **1 0 2 2 0** 2 2 **0 1 2 0 1** 1 1 1
1 0 0 0 0 2 0 0 0 2 1 0 0 2 2 2 0 2 0 0 1 2 1 0 1 1 2 2 1 0 1 0 2 2 1 2 2 0 2 1
0 1 2 2 2 1 1 0 0 2 0 2 0 2 1 2 1 2 2 1 2 1 **0 2 1 2 2** 2 2 1 0 0 0 2 2 0 0 2 0 1
0 2 1 1 2 2 2 0 1 0 0 1 1 2 0 1 0 1 1 1 2 1 0 0 1 2 2 0 1 1 0 1 1 0 2 1 0 2 2 2
2 2
length = 242

The following table shows several experimental results where the milestone period p=10.

PRMVS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
21101	0	4	7	33
02122	2	5	8	182
10220	1	4	5	105
01201	1	5	8	112
22112	0	3	7	23

Table 3-16. Experimental results for GF(3) and a 5th degree polynomial

Generator Polynomial of 6th degree

The generator polynomial $h(x) = x^6+x+2$ of degree $m = 6$ has the feedback equation represented by $a_{k+6} = 2a_{k+1} + a_k$ and generates a PRMVS of length $n=q^m-1 = 3^6-1=728$.

0 0 0 0 0 1 0 0 0 0 2 1 0 0 0 1 1 1 0 0 2 0 0 1 0 1 2 0 2 1 2 2 2 1 1 2 0 0 1 0
2 2 0 2 1 1 **0 2 1 1 0 1** 1 1 0 1 2 0 0 1 2 2 2 0 2 2 0 0 2 1 0 2 0 1 1 1 1 2 2 0
0 0 2 0 2 0 0 1 2 1 2 0 2 2 1 2 2 **1 0 1 2 0 1** 1 2 2 2 2 0 2 0 0 0 2 1 2 0 0 1 1
2 2 0 2 0 2 0 2 1 2 1 2 1 1 2 1 2 1 0 2 1 2 1 1 1 1 2 1 0 0 0 2 1 1 0 0 1 1 0 1
0 2 0 1 2 1 1 2 2 2 1 0 2 0 0 1 1 1 2 0 2 0 0 0 2 2 1 2 0 1 0 1 2 2 2 1 2 2 0 0 1
2 0 2 0 2 2 2 1 2 1 0 0 1 2 1 1 0 2 2 1 0 1 1 0 1 1 2 0 1 2 0 2 2 2 2 2 1 0 0 0
0 1 1 0 0 0 2 0 1 0 0 1 2 **2 1 0 2 2 0** 1 1 1 0 2 2 0 0 1 1 0 2 0 2 0 1 1 2 1 2 2
0 2 1 2 0 2 1 1 2 2 1 1 0 2 0 1 0 1 1 2 2 1 2 0 2 0 1 2 2 1 2 2 2 0 1 2 0 0 2 2
2 2 0 1 0 0 0 2 2 1 0 0 1 0 1 1 0 2 1 2 0 1 1 1 2 2 2 0 0 2 0 0 2 0 1 2 0 1 2 2
2 2 2 2 0 0 0 0 0 2 0 0 0 0 1 2 0 0 0 2 2 2 0 0 1 0 0 2 0 2 1 0 1 2 1 1 1 2 2 1
0 0 2 0 1 1 0 1 2 2 0 1 2 2 0 2 2 2 0 2 1 0 0 2 1 1 1 0 1 1 0 0 1 2 0 1 0 2 2 2
2 1 1 0 0 0 1 0 1 0 0 2 1 2 1 0 1 1 2 1 1 2 0 2 1 0 2 2 1 1 1 1 0 1 0 0 0 1 2 1
0 0 2 2 1 1 0 1 0 1 0 1 2 1 2 1 2 2 1 2 1 2 0 1 2 1 2 2 2 2 1 2 0 0 0 1 2 2 0 0
2 2 0 2 0 **1 0 2 1 2 2** 1 1 1 2 0 1 0 0 2 2 2 1 0 1 0 0 1 1 2 1 0 2 0 2 1 1 1 2 1
1 0 0 2 1 0 1 0 1 1 1 2 1 2 0 0 2 1 2 2 0 1 1 2 0 2 2 0 2 2 1 0 2 1 0 1 1 1 1 1
2 0 0 0 0 2 2 0 0 0 1 0 2 0 0 2 1 1 2 0 1 1 0 2 2 2 0 1 1 0 0 2 2 0 1 0 1 0 2 2
1 2 1 1 0 1 2 1 0 1 2 2 1 1 2 2 0 1 0 2 **0 2 2 1 1 2** 1 0 1 0 2 1 1 2 1 1 1 0 2 1
0 0 1 1 1 1 0 2 0 0 0 1 1 2 0 0 2 0 2 2 0 1 2 1 0 2 2 2 1 1 1 0 0 1 0 0 1 0 2 1
0 2 1 1 1 1 1 1
length = 728

The following table shows several experimental results where the milestone period p=25.

PRMVS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
021101	0	2	4	46
102122	2	5	0	525
210220	1	3	22	253
101201	0	4	3	97
022112	3	3	15	660

Table 3-17. Experimental results for GF(3) and a 6th degree polynomial

3.5.3 Finite field GF(5)

The finite field *GF(5)* has the order $q = p^m = 5^1 = 5$ where the prime order $p = 5$, and the positive integer $m = 1$.

The PRDEC coprocessor was synthesized to accommodate:

- a 5-value LFSR described by a generator polynomial of maximum 5th degree, and
- a milestone database composed of (4 clusters) x (7 words) x (15 bits);

The generic parameters for the chosen implementation are shown in Figure 3-49, while the synthesizing results for this implementation are shown in Figure 3-50.

```
ADDRESS_SIZE:           INTEGER:= 2;   -- width of the address bus
DATA_SIZE:              INTEGER:= 8;   -- width of the data bus
GF_WIDTH:               INTEGER:= 3;   -- width of GF elements
DOUBLE_GF_WIDTH:        INTEGER:= 6;   -- width of entry address for
                                       -- GF operation tables = 2*GF_WIDTH
MAX_POLYNOM_DEGREE:     INTEGER:= 5;   -- maximum degree of the
                                       -- irreducible polynomial
POLYNOM_DEGREE_WIDTH:INTEGER:= 3;      -- width of MAX_POLYNOM_DEGREE equal
                                       -- to LOG2(MAX_POLYNOM_DEGREE) + 1
                                       -- rounded up to the near integer
MILESTONE_SIZE:         INTEGER:= 15;  -- the size (in binary bits) of a
                                       -- milestone word saved in FIFO equal
                                       -- to MAX_POLYNOM_DEGREE * GF_WIDTH
MAX_NUMWORDS:           INTEGER:= 7;   -- maximum number of milestones
                                       -- saved in FIFO equal to
                                       -- MAX_POLYNOM_DEGREE + 2 (latency)
MILESTONE_CLUSTERS:     INTEGER:= 4;   -- number of milestone clusters
MILESTONE_CLUSTER_WIDTH:INTEGER:= 2    -- milestone cluster width
                                       -- equal to LOG2(MILESTONE_CLUSTERS)
```

Figure 3-49. Implementation parameters for GF(5)

```
** DEVICE SUMMARY **

Chip/                          Input  Output  Bidir  Memory  Memory          LCs
POF       Device               Pins   Pins    Pins   Bits    %Utilized  LCs  %Utilized

coprocessor
      EPF10K20TC144-3          6      10      8      384       3 %      887      76 %

User Pins:                     6      10      8
```

Figure 3-50. Synthesizing results for GF(5) implementation

Generator Polynomial of 2nd degree

The generator polynomial $h(x) = x^2+x+2$ of degree $m = 2$ has the feedback equation represented by $a_{k+2} = 4a_{k+1} + 3a_k$ and generates a PRMVS of length $n = q^m-1 = 5^2-1=24$.

0 **1** 4 4 3 4 0 2 3 3 1 **3** **0** 4 1 1 **2** **1** **0** **3** 2 **2** **4** 2
length = 24

The following table shows several experimental results where the milestone period p = 4.

PRBS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
21	1	0	4	16
03	1	0	2	18
14	0	1	4	1
24	0	0	3	21
30	0	3	4	11

Table 3-18. Experimental results for GF(5) and a 2nd degree polynomial

The tested PRMVS windows shown in the above experimental results table are marked in bold face while their beginning are marked by underscore.

Generator Polynomial of 3rd degree

The generator polynomial $h(x) = x^3+ x^2+2$ of degree $m = 3$ has the feedback equation represented by $a_{k+3} = 4a_{k+2} + 3a_k$ and generates a PRMVS of length $n=q^m-1 = 5^3-1=124$.

0 **0** **1** **4** 1 2 0 3 3 2 2 2 4 2 4 3 3 4 0 4 3 2 0 4 2 3 4 2 2 0 1 0 0 3 2 3 1 0 4 4
1 1 1 2 **1** **2** **4** 4 2 0 2 4 1 0 2 1 4 2 1 1 0 3 0 0 4 1 4 3 0 2 2 3 3 3 1 3 1 2 2 1
0 1 **2** **3** **0** 1 **3** **2** **1** 3 3 0 4 0 0 2 3 2 4 0 1 1 4 4 4 3 4 3 1 1 3 0 3 1 **4** **0** **3** 4 1 3
4 4 0 2
length = 124

The following table shows several experimental results where the milestone period p = 7.

PRBS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
321	2	3	5	86
403	3	2	5	114
014	0	1	6	1
124	1	2	5	44
230	2	2	2	82

Table 3-19. Experimental results for GF(5) and a 3rd degree polynomial

Generator Polynomial of 4th degree

The generator polynomial $h(x) = x^4 + x^3 + x + 3$ of degree $m = 4$ has the feedback equation represented by $a_{k+4} = 4a_{k+3} + 4a_{k+1} + 2a_k$ and generates a PRMVS of length $n = q^m - 1 = 5^4 - 1 = 624$.

0 0 0 1 4 1 3 0 2 2 4 4 3 2 2 3 1 1 0 0 1 1 4 0 1 2 1 3 2 1 3 1 2 2 3 2 0 1 3 1
3 1 4 0 0 3 0 0 2 4 1 2 3 4 1 0 2 0 2 1 3 0 3 1 0 2 3 4 4 2 0 4 2 2 4 2 0 0 1 3
2 2 2 2 0 2 0 4 4 0 1 3 0 4 0 1 0 3 1 1 1 4 2 4 4 2 3 1 0 1 4 3 1 2 3 2 3 3 1 0
3 2 0 2 2 2 1 1 1 2 4 2 3 2 4 2 2 3 3 4 2 1 1 0 3 3 4 3 0 2 3 3 0 1 2 4 0 0 0 3
2 3 4 0 1 1 2 2 4 1 1 4 3 3 0 0 3 3 2 0 3 1 3 4 1 3 4 3 1 1 4 1 0 3 4 3 4 3 2 0
0 4 0 0 1 2 **3 1 4 2** 3 0 1 0 1 3 4 0 4 3 0 1 4 2 2 1 0 2 1 1 2 1 0 0 3 4 1 1 1 1
0 1 0 2 2 0 3 4 0 2 0 3 0 4 3 3 3 2 1 2 2 1 4 3 0 3 2 4 3 1 4 1 4 4 3 0 4 1 0 1
1 1 3 3 3 1 2 1 4 1 2 1 1 4 4 2 1 3 3 0 4 4 2 4 0 1 4 4 0 3 1 2 0 0 0 4 1 4 2 0
3 3 1 1 2 3 3 2 4 4 0 0 4 4 1 0 4 3 4 2 3 4 2 4 3 3 2 3 0 4 2 4 2 4 1 0 0 2 0 0
3 1 4 3 2 **1 4 0 3** 0 3 4 2 0 2 4 **0 3 2 1** 1 3 0 1 3 3 1 3 0 0 4 2 3 3 3 3 0 3 0 1
1 0 4 2 0 1 0 4 0 2 4 4 4 1 3 1 1 3 2 4 0 4 1 2 4 3 2 3 2 2 4 0 2 3 0 3 3 3 4 4
4 3 1 3 2 3 1 3 3 2 2 1 3 4 4 0 2 2 1 2 0 3 2 2 0 4 3 1 0 0 0 2 3 2 1 0 4 4 3 3
1 4 4 1 2 2 0 0 2 2 3 0 2 4 2 1 4 2 1 2 4 4 1 4 0 2 1 2 1 2 3 0 0 1 0 0 4 3 2 4
1 3 2 0 4 0 4 2 1 0 1 2 0 4 1 3 3 4 0 3 4 4 3 4 0 0 2 1 4 4 4 4 0 4 0 3 3 0 2 1
0 3 0 2 0 1 2 2 2 3 4 3 3 4 1 2 0 2 3 1 2 4 1 4 1 1 **2 0 1 4** 0 4 4 4 2 2 2 4 3 4
1 4 3 4 4 1 1 3 4 2 2 0 1 1 3 1 0 4 1 1 0 2 4 **3**
length = 624

The following table shows several experimental results where the milestone period p=30.

PRBS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
0321	2	1	14	376
1403	2	1	25	365
2014	3	2	14	586
3142	1	1	4	206
3000	0	0	1	623

Table 3-20. Experimental results for GF(5) and a 4th degree polynomial

3.5.4 Finite field GF(4)

The finite field *GF(4)* has the order $q = p^m = 2^2 = 4$ where the prime order $p = 2$, and the positive integer $m = 2$, and it is generated by the primitive polynomial $h(x) = x^2+x+1$ with coefficients from *GF(2)*.

```
ADDRESS_SIZE:           INTEGER:= 2;   -- width of the address bus
DATA_SIZE:              INTEGER:= 8;   -- width of the data bus
GF_WIDTH:               INTEGER:= 2;   -- width of GF elements
DOUBLE_GF_WIDTH:        INTEGER:= 4;   -- width of entry address for
                                       -- GF operation tables = 2*GF_WIDTH
MAX_POLYNOM_DEGREE:     INTEGER:= 5;   -- maximum degree of the
                                       -- irreducible polynomial
POLYNOM_DEGREE_WIDTH:INTEGER:= 3;      -- width of MAX_POLYNOM_DEGREE equal
                                       -- to LOG2(MAX_POLYNOM_DEGREE) + 1
                                       -- rounded up to the near integer
MILESTONE_SIZE:         INTEGER:= 10;  -- the size (in binary bits) of a
                                       -- milestone word saved in FIFO equal
                                       -- to MAX_POLYNOM_DEGREE * GF_WIDTH
MAX_NUMWORDS:           INTEGER:= 7;   -- maximum number of milestones
                                       -- saved in FIFO equal to
                                       -- MAX_POLYNOM_DEGREE + 2 (latency)
MILESTONE_CLUSTERS:     INTEGER:= 4;   -- number of milestone clusters
MILESTONE_CLUSTER_WIDTH:INTEGER:= 2    -- milestone cluster width
                                       -- equal to LOG2(MILESTONE_CLUSTERS)
```

Figure 3-51. Implementation parameters for GF(4)

The PRDEC coprocessor was synthesized to accommodate:

- a 4-value LFSR described by a generator polynomial of maximum 5th degree, and
- a milestone database composed of (4 clusters) x (7 words) x (10 bits);

The generic parameters for the chosen implementation are shown in Figure 3-51, while the synthesizing results for this implementation are shown in Figure 3-52.

```
** DEVICE SUMMARY **

Chip/                          Input  Output  Bidir  Memory  Memory          LCs
POF        Device              Pins   Pins    Pins   Bits    %Utilized  LCs  %Utilized

coprocessor
      EPF10K20TC144-3          6      10      8      64        0 %      682      59 %

User Pins:                     6      10      8
```

Figure 3-52. Synthesizing results for GF(4) implementation

Generator Polynomial of 2nd degree

The generator polynomial $h(x) = x^2+3x+3$ of degree $m = 2$ has the feedback equation represented by $a_{k+2} = 3a_{k+1} + 3a_k$ and generates a PRMVS of length $n = q^m-1 = 4^2-1=16$.

0 1 3 1 1 **0 3 2** 3 3 0 2 1 2 **2**
length = 15

The following table shows several experimental results where the milestone period p = 5.

PRBS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
03	0	1	0	5
13	0	1	4	1
20	0	0	1	14
31	0	1	3	2
23	0	2	3	7

Table 3-21. Experimental results for GF(4) and a 2nd degree polynomial

The tested PRMVS windows shown in the above experimental results table are marked in bold face while their beginning are marked by underscore.

Generator Polynomial of 3rd degree

The generator polynomial $h(x) = x^3+3x^2+x+2$ of degree $m = 3$ has the feedback equation represented by $a_{k+3} = 3a_{k+2} + a_{k+1} + 2a_k$ and generates a PRMVS of maximal length equal to $n = q^m-1 = 4^3-1=64$.

0 0 1 3 3 3 0 2 0 2 2 3 3 2 3 1 **3 2 0 3 1** 0 0 2 1 1 **1 0 3** 0 3 3 1 1 3 1 **2 1 3** 0
1 2 0 0 3 2 2 2 0 1 0 1 1 2 2 **1 2 3** 2 1 0 2 3
length = 63

The following table shows several experimental results where the milestone period p = 5.

PRBS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
103	1	1	4	26
213	1	3	4	36
320	0	4	4	16
031	0	4	2	18
123	2	1	0	55

Table 3-22. Experimental results for GF(4) and a 3rd degree polynomial

Generator Polynomial of 4th degree

The generator polynomial $h(x) = x^4+ x^2+2x+3$ of degree $m = 5$ has the feedback equation represented by $a_{k+5} = a_{k+2} + 2a_{k+1} + 3a_k$ and generates a PRMVS of maximal length equal to $n = q^m-1 = 4^4-1 = 255$.

0 0 0 1 0 1 2 2 0 2 2 3 1 1 1 1 0 0 1 3 1 1 3 1 2 3 2 3 2 2 2 3 0 1 0 3 2 0 3 1
2 0 2 0 3 3 2 2 1 3 3 0 1 3 3 1 1 2 1 3 1 0 3 0 0 1 2 1 0 1 3 2 1 0 0 3 3 3 2 0
1 1 0 3 1 0 0 0 3 0 3 1 1 0 1 1 2 3 3 3 3 0 0 3 2 3 3 2 3 1 2 1 2 1 1 1 2 0 3 0
2 1 0 2 3 1 0 1 0 2 2 1 1 3 2 2 0 3 2 2 3 3 1 3 2 3 0 2 **0 0 3 1** 3 0 **3 2 1** 3 0 0
2 2 **2 1 0 3** 3 0 2 3 0 0 0 2 0 2 3 3 0 3 3 1 2 2 2 2 0 0 2 1 2 **2 1 2 3** 1 3 1 3 3
3 1 0 2 0 1 3 0 1 2 3 0 3 0 1 1 3 3 2 1 1 0 2 1 1 2 2 3 2 1 2 0 1 0 0 2 3 2 0 2
1 3 2 0 0 1 1 1 3 0 2 2 0 1 2
length = 255

The following table shows several experimental results where the milestone period p=15.

PRBS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
2103	1	5	3	162
3213	1	5	11	154
1320	2	4	0	240
0031	1	4	2	148
2123	2	1	4	191

Table 3-23. Experimental results for GF(4) and a 4th degree polynomial

Generator Polynomial of 5th degree

The generator polynomial $h(x) = x^5+x+2$ of degree $m = 5$ has the feedback equation represented by $a_{k+5} = a_{k+1} + 2a_k$ and generates a PRMVS of length $n=q^m-1 = 4^5-1=1023$.

0 0 0 0 1 0 0 0 1 2 0 0 1 0 3 0 1 2 3 1 1 0 0 0 3 2 0 0 3 3 3 0 3 2 2 1 3 3 1 2
1 2 0 0 2 0 3 0 2 3 3 1 2 0 2 0 0 3 2 3 0 3 3 0 1 3 2 1 1 1 3 2 3 3 1 3 0 2 0 1
1 2 3 1 3 0 0 0 1 1 0 0 1 3 2 0 1 1 3 3 1 3 1 2 0 1 0 0 3 1 2 0 3 0 0 3 3 1 0 3
2 0 2 3 3 3 2 0 2 2 3 3 2 1 0 2 3 2 2 2 0 3 1 1 3 3 0 3 1 2 1 3 0 0 2 1 1 0 2 2
3 2 2 1 0 3 1 2 2 3 0 0 1 0 1 0 1 2 1 2 1 0 2 0 2 2 2 3 2 1 1 0 3 2 3 2 3 3 0 3
0 2 1 3 1 2 2 1 0 0 1 2 2 0 1 0 1 3 1 2 1 1 0 0 2 3 2 0 2 0 3 3 2 3 3 2 3 0 2 3
0 1 2 0 1 1 0 3 1 3 2 3 0 1 3 0 1 1 1 1 1 3 3 3 3 1 2 2 2 0 0 1 1 3 0 1 3 1 1 1
1 0 3 3 3 2 3 2 2 3 0 3 1 0 1 3 0 2 1 1 1 2 2 3 3 0 1 0 2 1 1 2 2 2 3 0 1 1 0 1
1 3 2 1 3 1 3 2 1 0 1 3 2 2 1 1 3 1 2 3 1 0 0 0 0 2 0 0 0 2 3 0 0 2 0 1 0 2 3 1
2 2 0 0 0 1 3 0 0 1 1 1 0 1 3 3 2 1 1 2 3 2 3 0 0 3 0 1 0 3 1 1 2 3 0 3 0 0 1 3
1 0 1 1 0 2 1 3 2 2 2 1 3 1 1 2 1 0 3 0 2 2 3 1 2 1 0 0 0 2 2 0 0 2 1 3 0 2 2 1
1 2 1 2 3 0 2 0 0 1 2 3 0 1 0 0 1 1 2 0 1 3 0 3 1 1 1 3 0 3 3 1 1 3 2 0 3 1 3 3
3 0 1 2 2 1 1 0 1 2 3 2 1 0 0 3 2 2 0 3 3 1 3 3 2 0 1 2 3 3 1 0 0 2 0 2 0 2 3 2
3 2 0 3 0 3 3 3 1 3 2 2 0 1 3 1 3 1 1 0 1 0 3 2 1 2 3 3 2 0 0 2 3 3 0 2 **0 2 1 2**
3 2 2 0 0 3 1 3 0 3 0 1 1 3 1 1 3 1 0 3 1 0 2 3 0 2 2 0 1 2 1 3 1 0 2 1 0 2 2 2
2 2 1 1 1 1 2 3 3 3 0 0 2 2 1 0 2 1 2 2 2 2 0 1 1 1 3 1 3 3 1 0 1 2 **0 2 1 0 3** 2
2 2 3 3 1 1 0 2 0 3 2 2 3 3 3 1 0 2 2 0 2 2 1 3 2 1 **2 1 3 2 0** 2 1 3 3 2 2 1 2 3
1 2 0 0 0 0 3 0 0 0 3 1 0 0 3 0 2 0 3 1 2 3 3 0 0 0 2 1 0 0 2 2 2 0 2 1 1 3 2 2
3 1 3 1 0 0 1 0 2 0 1 2 2 3 1 0 1 0 0 2 1 2 0 2 2 0 3 2 1 3 3 3 2 1 2 2 3 2 0 1
0 3 3 1 2 3 2 0 0 0 3 3 0 0 3 2 1 0 3 3 2 2 3 2 3 1 0 3 0 0 2 3 1 0 2 0 0 2 2 3
0 2 1 0 1 2 2 2 1 0 1 1 2 2 1 3 0 1 2 1 1 1 0 2 3 3 2 2 0 2 3 1 3 2 0 0 1 3 3 0
1 1 2 1 1 3 0 2 3 1 1 2 0 0 3 0 3 0 3 1 3 1 3 0 1 0 1 1 1 2 1 3 3 0 2 1 2 1 2 2
0 2 0 1 3 2 3 1 1 **3 0 0 3 1** 1 0 3 0 3 2 3 1 3 3 0 0 1 2 1 0 1 0 2 2 1 2 2 1 2 0
1 2 0 3 1 0 3 3 0 2 3 2 1 2 0 3 2 0 3 3 3 3 3 2 2 2 2 3 1 1 1 0 0 3 3 2 0 3 2 3
3 3 3 0 2 2 2 1 2 1 1 2 0 2 3 0 3 2 0 1 3 3 3 1 1 2 2 0 3 0 1 3 3 1 1 1 2 0 3 3
0 3 3 2 1 3 2 3 2 1 3 0 3 2 1 1 3 3 2 3 1 2 3
length = 1023

The following table shows several experimental results where the milestone period p=40.

PRBS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
02103	2	2	22	634
13213	1	1	8	320
21320	2	3	31	666
30031	3	1	13	889
02123	2	0	18	556

Table 3-24. Experimental results for GF(4) and a 5th degree polynomial

3.5.5 Finite field GF(8)

The finite field *GF(8)* has the order $q = p^m = 2^3 = 8$ where the prime order $p = 2$, and the positive integer $m = 3$, and it is generated by the primitive polynomial $h(x) = x^3+x+1$ with coefficients from *GF(2)*.

```
ADDRESS_SIZE:           INTEGER:= 2;   -- width of the address bus
DATA_SIZE:              INTEGER:= 8;   -- width of the data bus
GF_WIDTH:               INTEGER:= 3;   -- width of GF elements
DOUBLE_GF_WIDTH:        INTEGER:= 6;   -- width of entry address for
                                       -- GF operation tables = 2*GF_WIDTH
MAX_POLYNOM_DEGREE:     INTEGER:= 4;   -- maximum degree of the
                                       -- irreducible polynomial
POLYNOM_DEGREE_WIDTH:INTEGER:= 3;      -- width of MAX_POLYNOM_DEGREE equal
                                       -- to LOG2(MAX_POLYNOM_DEGREE) + 1
                                       -- rounded up to the near integer
MILESTONE_SIZE:         INTEGER:= 12;  -- the size (in binary bits) of a
                                       -- milestone word saved in FIFO equal
                                       -- to MAX_POLYNOM_DEGREE * GF_WIDTH
MAX_NUMWORDS:           INTEGER:= 6;   -- maximum number of milestones
                                       -- saved in FIFO equal to
                                       -- MAX_POLYNOM_DEGREE + 2 (latency)
MILESTONE_CLUSTERS:     INTEGER:= 4;   -- number of milestone clusters
MILESTONE_CLUSTER_WIDTH:INTEGER:= 2    -- milestone cluster width
                                       -- equal to LOG2(MILESTONE_CLUSTERS)
```

Figure 3-53. Implementation parameters for GF(8)

The PRDEC coprocessor was synthesized to accommodate:

- a 8-value LFSR described by a generator polynomial of maximum 4th degree, and
- a milestone database composed of (4 clusters) x (6 words) x (12 bits);

The generic parameters for the chosen implementation are shown in Figure 3-53, while the synthesizing results for this implementation are shown in Figure 3-54.

```
** DEVICE SUMMARY **

Chip/                          Input Output Bidir  Memory  Memory          LCs
POF       Device               Pins  Pins   Pins   Bits    %Utilized  LCs  %Utilized

coprocessor
      EPF10K20TC144-3          6     10     8      384       3 %     719      62 %

User Pins:                     6     10     8
```

Figure 3-54. Synthesizing results for GF(8) implementation

Generator Polynomial of 2nd degree

The generator polynomial $h(x) = x^2+x+4$ of degree $m = 2$ has the feedback equation represented by $a_{k+2} = a_{k+1} + 4a_k$ and generates a PRMVS of length $n = q^m-1 = 8^2-1=63$.

0 1 1 2 1 6 3 5 1 0 4 **4** **5** 4 2 6 1 4 0 7 7 1 7 5 2 4 7 0 3 3 4 3 1 5 7 3 0 6 **6** 7
6 4 **1** 3 6 **0** **2** **2** 3 2 7 4 6 2 0 5 5 6 5 3 7 2 5
length = 63

The following table shows several experimental results where the milestone period p = 5.

PRBS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
02	2	1	0	45
13	2	1	3	42
45	0	3	4	11
67	2	0	2	38
23	2	2	3	47

Table 3-25. Experimental results for GF(8) and a 2nd degree polynomial

The tested PRMVS windows shown in the above experimental results table are marked in bold face while their beginning are marked by underscore.

Generator Polynomial of 3rd degree

The generator polynomial $h(x) = x^3+x+2$ of degree $m = 3$ has the feedback equation represented by $a_{k+3} = a_{k+1} + 2a_k$ and generates a PRMVS of length $n = q^m-1 = 8^3-1=511$.

0 0 1 0 1 2 1 0 7 2 7 4 1 2 6 0 4 7 4 4 2 7 3 1 7 2 6 4 4 5 7 0 2 1 2 7 0 1 1 1
4 4 1 7 6 6 5 2 4 7 6 4 5 5 0 1 6 1 7 3 6 7 3 0 7 4 7 2 4 **4 6** 7 1 0 0 2 0 2 3 2
0 1 3 1 **5 2 3** 7 0 5 1 5 5 3 1 4 **2 1 3** 7 5 5 6 1 0 3 2 3 **1 0 2** 2 2 5 5 2 1 7 7 6
3 5 1 7 5 6 6 0 2 7 2 1 4 7 1 4 0 1 5 1 3 5 5 7 1 2 0 0 3 0 3 4 3 0 2 4 2 6 3 4
1 0 6 2 6 6 4 2 5 3 2 4 1 6 6 7 2 0 4 3 4 2 0 3 3 3 6 6 3 2 1 1 7 4 6 2 1 6 7 7
0 3 1 3 2 5 1 2 5 0 2 6 2 4 6 6 1 2 3 0 0 4 0 4 5 4 0 3 5 3 7 4 5 2 0 7 3 7 7 5
3 6 4 3 5 2 7 7 1 3 0 5 4 5 3 0 4 4 4 7 7 4 3 2 2 1 5 7 3 2 7 1 1 0 4 2 4 3 6 2
3 6 0 3 7 3 5 7 7 2 3 4 0 0 5 0 5 6 5 0 4 6 4 1 5 6 3 0 1 4 1 1 6 4 7 5 4 6 3 1
1 2 4 0 6 5 6 4 0 5 5 5 1 1 5 4 3 3 2 6 1 4 3 1 2 2 0 5 3 5 4 7 3 4 7 0 4 1 4 6
1 1 **3 4 5** 0 0 6 0 6 7 6 0 5 7 5 2 6 7 4 0 2 5 2 2 7 5 1 6 5 7 4 2 2 3 5 0 7 6 7
5 0 6 6 6 2 2 6 5 4 4 3 7 2 5 4 2 3 3 0 6 4 6 5 1 4 5 1 0 5 2 5 7 2 2 4 5 6 0 0
7 0 7 1 7 0 6 1 6 3 7 1 5 0 3 6 3 3 1 6 2 7 6 1 5 3 3 4 6 0 1 7 1 6 0 7 7 7 3 3
7 6 5 5 4 1 3 6 5 3 4 4 0 7 5 7 6 2 5 6 2 0 6 3 6 1 3 3 5 6 7
length = 511

The following table shows several experimental results where the milestone period p=30.

PRBS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
102	0	4	12	108
213	0	4	24	96
345	2	3	28	362
467	0	3	21	69
523	0	3	6	84

Table 3-26. Experimental results for GF(8) and a 3rd degree polynomial

3.5.6 Finite field GF(9)

The finite field *GF(9)* has the order $q = p^m = 3^2 = 9$ where the prime order $p = 3$, and the positive integer $m = 2$, and it is generated by the primitive polynomial $h(x)=x^2+2x+2$ with coefficients from *GF(3)*.

The PRDEC coprocessor was synthesized to accommodate:

- a 8-value LFSR described by a generator polynomial of maximum 4th degree, and
- a milestone database composed of (4 clusters) x (6 words) x (16 bits);

The generic parameters for the chosen implementation are shown in Figure 3-55, while the synthesizing results for this implementation are shown in Figure 3-56.

```
ADDRESS_SIZE:           INTEGER:= 2;   -- width of the address bus
DATA_SIZE:              INTEGER:= 8;   -- width of the data bus
GF_WIDTH:               INTEGER:= 4;   -- width of GF elements
DOUBLE_GF_WIDTH:        INTEGER:= 8;   -- width of entry address for
                                       -- GF operation tables = 2*GF_WIDTH
MAX_POLYNOM_DEGREE:     INTEGER:= 4;   -- maximum degree of the
                                       -- irreducible polynomial
POLYNOM_DEGREE_WIDTH:INTEGER:= 3;      -- width of MAX_POLYNOM_DEGREE equal
                                       -- to LOG2(MAX_POLYNOM_DEGREE) + 1
                                       -- rounded up to the near integer
MILESTONE_SIZE:         INTEGER:= 16;  -- the size (in binary bits) of a
                                       -- milestone word saved in FIFO equal
                                       -- to MAX_POLYNOM_DEGREE * GF_WIDTH
MAX_NUMWORDS:           INTEGER:= 6;   -- maximum number of milestones
                                       -- saved in FIFO equal to
                                       -- MAX_POLYNOM_DEGREE + 2 (latency)
MILESTONE_CLUSTERS:     INTEGER:= 4;   -- number of milestone clusters
MILESTONE_CLUSTER_WIDTH:INTEGER:= 2    -- milestone cluster width
                                       -- equal to LOG2(MILESTONE_CLUSTERS)
```

Figure 3-55. Implementation parameters for GF(9)

```
** DEVICE SUMMARY **

Chip/                          Input Output Bidir  Memory  Memory          LCs
POF       Device               Pins  Pins   Pins   Bits    %Utilized  LCs  %Utilized

coprocessor
      EPF10K20TC144-3          6     10     8      2048      16 %     868     75 %

User Pins:                     6     10     8
```

Figure 3-56. Synthesizing results for GF(9) implementation

Generator Polynomial of 2nd degree

The generator polynomial $h(x) = x^2+5x+6$ of degree $m = 2$ has the feedback equation represented by $a_{k+2} = a_{k+1} + 2a_k$ and generates a PRMVS of length $n = q^m-1 = 9^2-1=80$.

0 1 1 8 **3 4** 8 2 8 5 0 6 6 5 8 1 5 7 5 2 0 3 3 2 **5 6** 2 4 **2** 7 0 8 8 7 2 3 7 1 7 4
0 5 5 4 **7 8** 4 6 4 1 0 2 2 1 4 5 1 3 1 6 0 7 7 6 **1 2** 6 8 6 3 0 4 4 3 6 7 3 5 3 8
length = 80

The following table shows several experimental results where the milestone period p = 7.

PRBS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
12	2	2	6	64
34	0	1	3	4
56	1	0	4	24
78	1	3	5	44
27	1	0	0	28

Table 3-27. Experimental results for GF(9) and a 2nd degree polynomial

The tested PRMVS windows shown in the above experimental results table are marked in bold face while their beginning are marked by underscore.

Generator Polynomial of 3rd degree

The generator polynomial $h(x) = x^3+5x+6$ of degree $m = 3$ has the feedback equation represented by $a_{k+3} = a_{k+1}+2a_k$ and generates a PRMVS of length $n = q^m-1 = 9^3-1=728$.

0 0 1 0 1 2 1 6 4 0 1 5 1 **7 3 4** 5 8 1 1 5 8 7 1 2 7 6 0 1 7 1 4 7 5 8 2 1 8 4 3
6 6 7 5 3 2 8 5 5 0 4 6 4 4 1 3 0 1 4 1 5 0 7 6 7 1 3 7 1 1 7 8 4 4 6 3 4 0 8 5
8 0 1 1 1 8 8 3 7 4 1 0 0 2 0 2 3 2 7 5 0 2 6 2 8 4 5 6 1 2 2 6 1 8 2 3 8 7 0 2
8 2 5 8 6 1 3 2 1 5 4 7 7 8 6 4 3 1 6 6 0 5 7 5 5 2 4 0 2 5 2 6 0 8 7 8 2 4 8 2
2 8 1 5 5 7 4 5 0 1 6 1 0 2 2 2 1 1 4 8 5 2 0 0 3 0 3 4 3 8 6 0 3 7 3 1 5 6 7 2
3 3 7 2 1 3 4 1 8 0 3 1 3 6 1 7 2 4 3 2 6 5 8 8 1 7 5 4 2 7 7 0 6 8 6 6 3 5 0 3
6 3 7 0 1 8 1 3 5 1 3 3 1 2 6 6 8 5 6 **0 2** 7 2 0 3 3 3 2 2 5 1 6 3 0 0 4 0 4 5 4
1 7 0 4 8 4 2 6 7 8 3 4 4 8 3 2 4 5 2 1 0 4 2 4 7 2 8 3 5 4 3 7 6 1 1 2 8 6 5 3

8 8 0 7 1 7 7 4 6 0 4 7 4 8 0 2 1 2 4 6 2 4 4 2 3 7 7 1 6 7 0 3 8 3 0 4 4 4 3 3
6 2 7 4 0 0 5 0 5 6 5 2 8 0 5 1 5 3 7 8 1 4 5 5 1 4 **3 5 6** 3 2 0 5 3 5 8 3 1 4 6
5 4 8 7 2 2 3 1 7 6 4 1 1 0 8 2 8 8 5 7 0 5 8 5 1 0 3 2 3 5 7 3 5 5 3 4 8 8 **2** 7
8 0 4 1 4 0 5 5 5 4 4 7 3 8 5 0 0 6 0 6 7 6 3 1 0 6 2 6 4 **8 1 2** 5 6 6 2 5 4 6 7
4 3 0 6 4 6 1 4 2 5 7 6 5 1 8 3 3 4 2 8 7 5 2 2 0 1 3 1 1 6 8 0 6 1 6 2 0 4 3 4
6 8 4 6 6 4 5 1 1 3 8 1 0 5 2 5 0 6 6 6 5 5 8 4 1 6 0 0 7 0 7 8 7 4 2 0 7 3 7 5
1 2 3 6 7 7 3 6 5 7 8 5 4 0 7 5 7 2 5 3 6 8 7 6 2 1 4 4 5 3 1 8 6 3 3 0 2 4 2 2
7 1 0 7 2 7 3 0 5 4 5 7 1 5 7 7 5 6 2 2 4 1 2 0 6 3 6 0 7 7 7 6 6 1 5 2 7 0 0 8
0 8 1 8 5 3 0 8 4 8 6 2 3 4 7 8 8 4 7 6 8 1 6 5 0 8 6 8 3 6 4 7 1 8 7 3 2 5 5 6
4 2 1 7 4 4 0 3 5 3 3 8 2 0 8 3 8 4 0 6 5 6 8 2 6 8 8 6 7 3 3 5 2 3 0 7 4 7 0 8
8 8 7 7 2 6 3 8
length = 728

The following table shows several experimental results where the milestone period p=40.

PRBS Window	Milestone			Computed Window Index Recovery
	Cluster	Index	Offset	
812	2	2	11	469
734	0	1	27	13
356	2	0	14	386
278	2	1	2	438
027	1	2	7	273

Table 3-28. Experimental results for GF(9) and a 3rd degree polynomial

3.6 Performance Evaluation

As explained in the Control Unit design section (subchapter 3.1.1), the pipelined processing architecture of the PRDEC coprocessor requires the following hardware resources:

- *m*-cells for the irreducible polynomial coefficients;
- *1*-cell for the multiplication table;
- *1*-cell for the addition table;
- *(m+1)*-cells for the shift register (*1* cell used to implement a delay cell);
- symbol generator cycle is *m+2* clocks.

It can be seen that the PRDEC coprocessor requires $m+2$ clock cycles to generate a new pseudo-random multi-valued symbol, where m represents the degree of the irreducible polynomial used to implement the LFSR.

If p represents the PRS milestone period then it takes at most $p \cdot (m+2)$ clock cycles for the PRDEC coprocessor to decode a m-width PRS window. This represents the most unfavorable case when the PRS window to be decoded is just one symbol after the origin of a PRS milestone and the PRS generator has to generate p symbols in order to get to the next milestone.

If $n=q^m-1$ is the length of a PRS generated by a m-degree irreducible polynomial defined over the finite field $GF(q)$ then the PRS milestone period has to be in the range $1 \leq p \leq n$. It should be noted that $p=n$ represents a serial-type code conversion (described in subchapter 2.2.1) while $p=1$ represents a parallel-type code conversion (described in subchapter 2.2.2). Based on this statement it can be said that actually both serial- and parallel-type code conversions represent particular cases of the serial-parallel-type code conversion at the two ends of the PRS milestone period range.

What the above analysis means for a real implementation?

Consider the particular case of the pseudo-random decoding application described in section 3.4.1 defined by the PRS generated by the generator polynomial $h(x)=x^3+x^2+x+2$ over the finite field $GF(4)$. Assigning the Altera FLEX10K family for PRDEC coprocessor synthesis the MAX+PLUS II compiler chose the device EPF10K10LC84-3, which is the smallest and the slowest device in the family accommodating this implementation.

Checking the implementation for Timing Performance a maximum clock frequency of 36.23MHz was found. It can be seen from the Figure 3-57 that the clock speed of the implementation is limited actually by the access time to the on-chip RAM that stores the algebra operation tables.

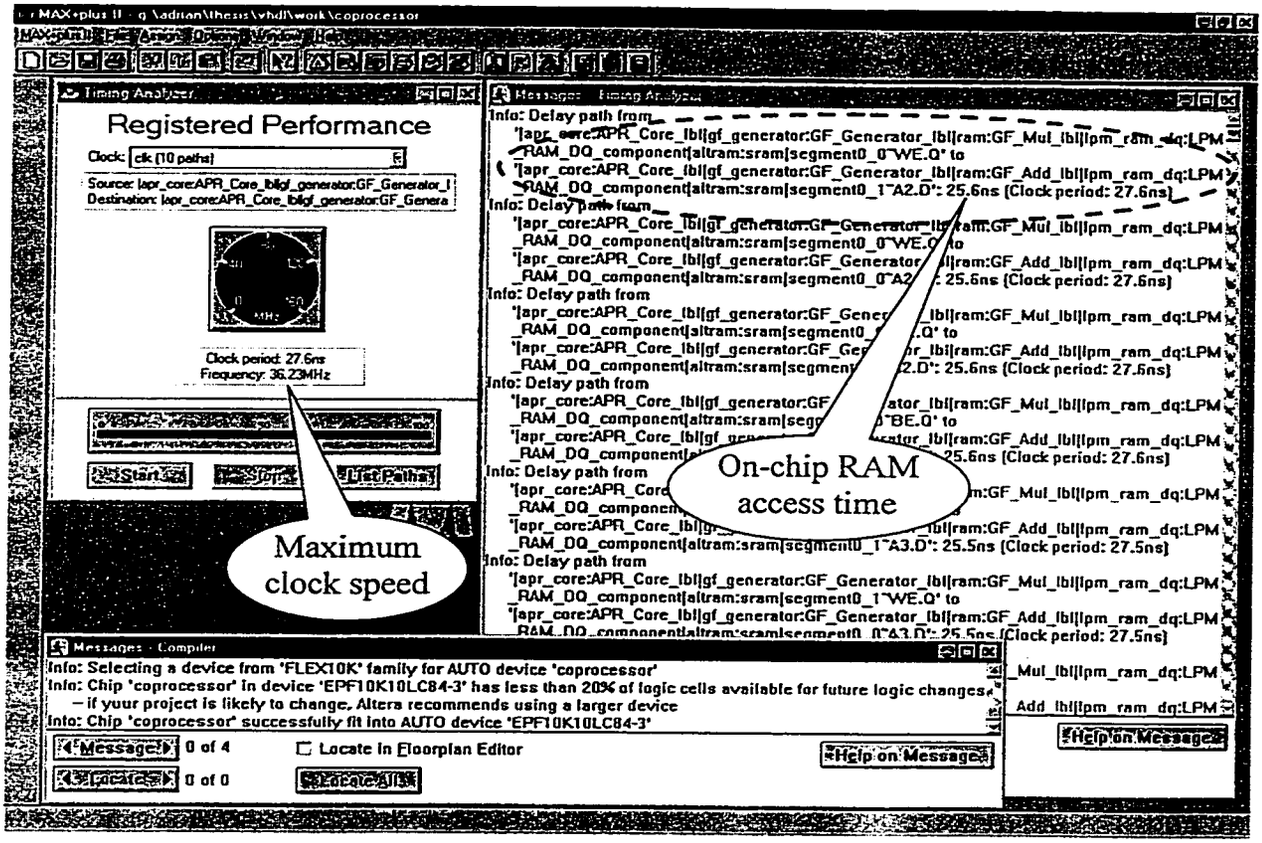


Figure 3-57. Compiler and timing analyzer results

When programming the synthesized PRDEC coprocessor with the algebra defined over *GF(4)*, a generator polynomial of degree $m=3$ and the milestone period $p=5$ (see section 3.4.1 and the simulation waveform diagram in Figure 3-39), the most unfavorable PRS window decoding case requires $p \cdot (m+2)=25$ clock cycles. Based on this result it can be safely assumed that on average there are necessary 12.5 clock cycles to decode a 3-symbol PRS window by the synthesized PRDEC coprocessor programmed with the above mentioned parameters.

Exercising the synthesized PRDEC coprocessor at a clock frequency of 33MHz, which is a standard microprocessor frequency, the maximum decoding rate that can be achieved by this set-up is equal to 33MHz / (12.5 clock cycles) = 2.64 million decodings per second. Of course this is a theoretical value. The real value for the decoding throughput rate is calculated by substracting from the theoretical value the time necessary to load the

coprocessor with the PRS window to be decoded and also the time required by the μP to read the decoding results from the coprocessor.

The throughput decoding rate achieved even by this modest implementation of the PRDEC coprocessor is by far superior in terms of processing speed and cost to any pseudo-random-to-natural code conversion algorithm employing a purely software solution. For applications requiring a very high encoding resolution the hardware implementation of the pseudo-random-to-natural code conversion presents an even higher relative processing speed and it is more cost effective compared to a software solution.

If there is a requirement to increase the PRDEC coprocessor operating speed, it can be done by applying one or many of the following methods:

- Implement the on-chip RAM as flip-flops, which are inherently faster, rather than native memory bits. This may result in changing the FLEX10K family chosen for the implementation with a memoryless family.
- Optimize the logic synthesis for speed rather than area. This may result in a larger device to be used for implementation, which inherently has a higher price.
- Choose a device with a faster speed grade, which will have a higher price.
- Manually fine tune the slowest parts of the design, process which is time consuming.

Chapter 4. CONCLUSIONS AND FUTURE RESEARCH

The objective of this work was to study a generalization of the pseudo-random multi-valued encoding and code conversion algorithm and to develop a specialized hardware platform for this generalized decoding algorithm. A PRDEC coprocessor was developed and synthesized in FPGA technology. This design has been documented in VHDL and its functionality has been verified by extensive VHDL simulations.

The coprocessor represents a practical PRMVS-to-natural decoding solution that delivers real-time decoding performance needed for the high resolution encoding meaningful for industrial applications.

Figure 4-1 presents the block diagram of the coprocessor with a single PRMVS processing unit. The PRMVS processing unit is represented by the PRDEC core, which is detailed in Figure 3-4.

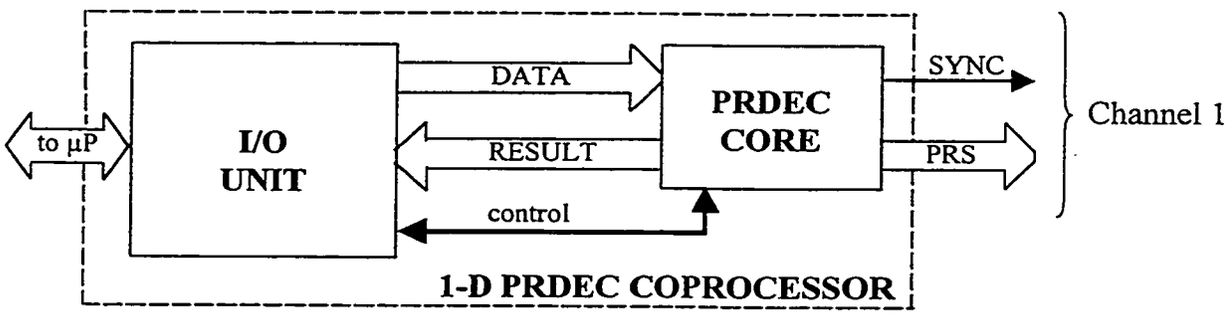


Figure 4-1. Block diagram of the 1-D PRDEC coprocessor

For applications that require processing of multiple PRMVS simultaneously, multiple PRDEC cores (all served by the same I/O unit) can be synthesized into one physical chip. Figure 4-2 shows the block diagram of a 3-D PRDEC coprocessor that has the capability of processing three PRMVSs simultaneously.

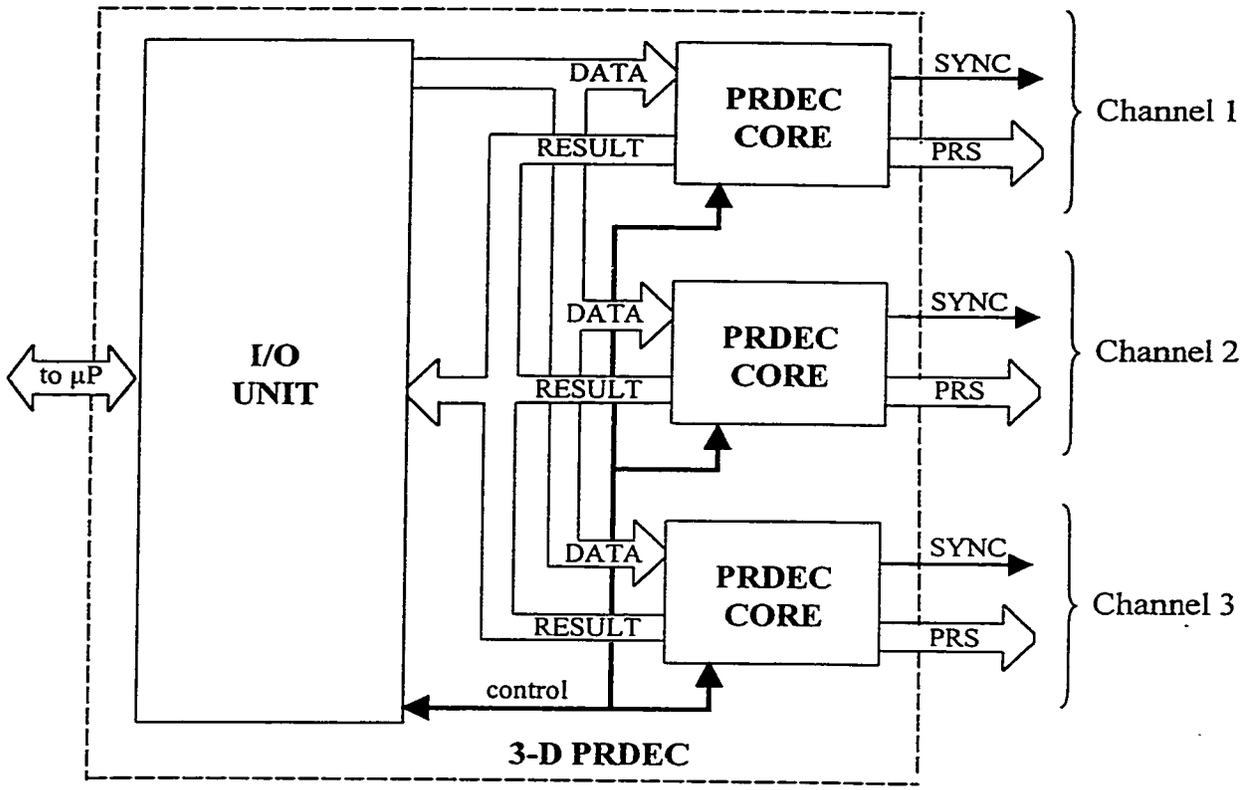


Figure 4-2. Block diagram of the 3-D PRDEC coprocessor

For specific applications the algebra and the milestone table can be hardwired into the chip in order to reduce the silicon resources and to speed up the processing. Being hardwired, these implementations are very fast and consume few silicon resources but they lack the flexibility offered by a generic microprogrammed implementation.

For a generalized type of applications the algebra downloaded into the coprocessor represents the GF employed by the application, the coefficients of the irreducible polynomial, and the PRS milestone period. Such a generic (programmable) PRDEC coprocessor inherently will use a larger amount of silicon resources then a specialized one. It will also work at a lower speed. However the generic PRDEC coprocessor could be used for decoding a multitude of PRMVSs while the specialized one is restricted to only one PRMVS.

In the same line of work future research can be done in order to increase the operating speed of the generic PRDEC coprocessor by implementing the on-chip RAM as flip-flops rather than native memory bits, which are synthesized as latches.

The current implementation of the coprocessor is based on the finite field operations mapped as look-up tables into the FPGA's on-chip memory. The size of these look-up table is *qxq*, where *q* represents the number of elements in a finite field *GF(q)*. It can be seen that the size of the two operation tables, addition and multiplication, increases exponentially with the size of the finite field, resulting in slower and more expensive decoding devices. In order to overcome this, a promising research area can be explored by implementing the finite field operations as Zech's logarithms [MaS77]. In this scheme only the polar representation (i.e. as a power of a primitive element t) of the field elements is used. The element multiplication is done easy by using a special "polar representation adder" while the addition is carried out by using Zech's logarithms, which imply using a single look-up table of size *qx1* that makes the translation from Zech's logarithms to corresponding finite field elements.

In conclusion, using Zech's logarithms to do the arithmetic in a finite field *GF(q)* the hardware resources required are represented by some sort of special adder and a look-up table of size *qx1* as opposed to two look-up tables of size *qxq* used by the PRDEC coprocessor implementation chosen for this work.

BIBLIOGRAPHY

[Alt99] *****, *Device Data Book*, Altera Corporation, May 1999.

[Ara84] B. Arazi, *Position Recovery using Binary Sequences,* IEE Electronic Letters, Vol.20, pp.61-62, 1984.

[Cha97] K.C. Chang, *Digital Design and Modeling with VHDL and Synthesis,* IEEE Computer Society Press, 1997

[Gol67] S.W. Golomb, *Shift Register Sequences,* Holden Day, Inc., 1967

[GrT74] D.H. Green, I.S. Taylor, *Irreducible Polynomials over composite Galois Fields and their Applications in Coding Techniques,* Proc. IEE, Vol.121, No.9, pp.935-939, September 1974.

[MaS77] F.J. MacWilliams, N.J.A. Sloane, *The Theory of Error-Correcting Codes*, North-Holland Mathematical Library, Elsevier Science Publishers B.V., 1977.

[MaS76] F.J. MacWilliams, N.J.A. Sloane, *Pseudo-Random Sequences and Arrays*, Proceedings of the IEEE, Vol.64, No.12, pp.1715-1729, 1976.

[Mor97] T. Moreau, *Pseudo-Random Generators, a High-Level Survey-in-Progress,* CONNOTECH Experts-conseils Inc., 1997, On-line. Available: http://www.connotech.com/RNG.HTM

[Mot91] *****, *M68HC11 Reference Manual*, Motorola, Inc., revision 3, 1991,

[Paa96] C. Paar, *A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields*, IEEE Transactions on Computers, Vol.45, No.7, pp.856-861, 1996.

[PaP98] C. Paar, P. Roelse, *Efficient Multiplier Architectures for Galois Fields* $GF(2^{4n})$, IEEE Transactions on Computers, Vol.47, No.2, pp.162-169, 1998.

[Pel95] D. Pellerin, *Digital Design and Modelling with VHDL and Synthesis,* Accolade Design Automation, Inc., 1995, On-line.
Available: http://www.acc-eda.com/h_intro.htm

[Pe&al00] E.M. Petriu, Z. Sakr, H.J.W. Spoelder, A. Moica, *Object Recognition Using Pseudo-random Color Encoded Structured Light,* IEEE Instrumentation & Measurement Magazine, 2000.

[Pe98] E.M. Petriu, *Absolute Position Measurement using Pseudo-Random Binary Encoding,* IEEE Instrumentation & Measurement Magazine, Vol. 1, No.3, pp.19-23, September 1998.

[Pe&al96] E.M. Petriu, D. Ionescu, D.C. Petriu, S.K. Yeung, Ph. Lavoie, N. Trif, *Absolute Position Measurement Applications of Pseudo-Random Encoding*, Proceedings ETIM'96 IEEE International Workshop on Emergent Technologies for Instrumentation Measurement, pp.119-126, Como, Italy, 1996.

[Pe89] E.M. Petriu, *Reconsidering Natural Binary Encoding for Absolute Position Measurement Application,* IEEE Transactions on Instrumentation Measurement, Vol.38, No.5, pp. 1014-1016, 1989.

[Pe88-1] E.M. Petriu, *New Pseudo-Random/Natural Code Conversion Method,* IEE Electronic Letters, Vol.24, No. 22, pp.1358-1359, 1988.

[Pe88-2] E.M. Petriu, *Absolute Position Recovery for Automated Path-Guided Vehicles,* Proceedings ROBOTS 12 and VISIONS '88 Conference Exhibition, Detroit, MI, June 5-9, 1988.

[Pe87] E.M. Petriu, *Absolute-type position transducers using a pseudo-random encoding,* IEEE Transactions on Instrumentation Measurement, Vol.36, No.4, pp.950-955, 1987.

[Rom92] S. Roman, *Coding and Information Theory*, Springer-Verlag, 1992

[Rot90] J. Rotman, *Galois Theory*, Springer-Verlag, 1990

[Sch97] M.R. Schroeder, *Number Theory in Science and Communication (With Applications in Cryptography, Physics, Digital Information, Computing, and Self-Similarity)*, Springer-Verlag, 1997

[Sm97] M. Smith, *Application Specific Integrated Circuits*, Addison-Wesley, 1997.

[Smi96] D.J. Smith, *HDL Chip Design – A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs using VHDL or Verilog,* Doone Publications, 1996

[Tr93] N. Trif, *Model-Based Visual Recognition of 3-D Objects Using Pseudo-Random Grid Encoding*, Thesis, M.A.Sc. in Electrical Engineering, University of Ottawa, 1993.

Annex A. PRDEC Coprocessor Implementation in VHDL

A-1. General Purpose VHDL Libraries

Vendor Library

```
library ieee;
use ieee.std_logic_1164.all;

package VENDOR_package is

   component D_FF          -- D type Flip-Flop component
   port (
      d, clk, reset, preset: in STD_LOGIC;
      q: out STD_LOGIC);
   end component;

   component D_FFE         -- D type Flip-Flop with Enable component
   port (
      d, clk, ena, reset, preset: in STD_LOGIC;
      q: out STD_LOGIC);
   end component;

   component JK_FF         -- JK type Flip-Flop component
   port (
      j, k, clk, reset, preset: in STD_LOGIC;
      q: out STD_LOGIC);
   end component;

   component JK_FFE        -- JK type Flip-Flop with Enable component
   port (
      j, k, clk, ena, reset, preset: in STD_LOGIC;
      q: out STD_LOGIC);
   end component;

   component T_FF          -- T type Flip-Flop component
   port (
      t, clk, reset, preset: in STD_LOGIC;
      q: out STD_LOGIC);
   end component;

   component T_FFE         -- T type Flip-Flop with Enable component
   port (
      t, clk, ena, reset, preset: in STD_LOGIC;
      q: out STD_LOGIC);
   end component;

   component Generic_Binary_Counter        -- Generic Counter component
   generic (Size: integer := 3);
    port (
      clk:      in STD_LOGIC;              -- Counter Clock
      enable:   in STD_LOGIC;              -- Enable Counting
      areset:   in STD_LOGIC;              -- Counter Asynchronous Reset
      sreset:   in STD_LOGIC;              -- Counter Synchronous Reset
      count:    out STD_LOGIC_VECTOR (Size-1 downto 0)-- Counter Output
    );
   end component;
```

```
    component Generic_Register                -- Generic Register component
    generic (RegSize: integer := 8);
    port (
       data:     in STD_LOGIC_VECTOR (RegSize-1 downto 0);-- Input Data signals
       clk:      in STD_LOGIC;                -- Register Clock
       enable:   in STD_LOGIC;                -- Enable Data Buffering
       reset:    in STD_LOGIC;                -- Register Reset
       preset:   in STD_LOGIC;                -- Register Preset
       reg:   out STD_LOGIC_VECTOR (RegSize-1 downto 0)-- Buffered Data signals
    );
    end component;

    component ram
    generic (
       ADDRESS_SIZE: INTEGER := 4;
       DATA_SIZE: INTEGER := 2);
    port (
       address    : in STD_LOGIC_VECTOR (ADDRESS_SIZE-1 downto 0);
       we         : in STD_LOGIC ;
       inclock    : in STD_LOGIC ;
       data       : in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
       q          : out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0));
    end component;

    component rom
    generic (
       ROM_ADDRESS_SIZE: INTEGER := 4;
       DATA_SIZE: INTEGER := 2;
       DATA_FILE: STRING := "rom.mif");
    port (
       address    : in STD_LOGIC_VECTOR (ROM_ADDRESS_SIZE-1 downto 0);
       q          : out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0));
    end component;

end VENDOR_package;
```

Vendor Library Components

```
--------------------------------------------------------------------------------
-- D type Flip-Flop

library ieee;
use ieee.std_logic_1164.all;

entity D_FF is          -- D type Flip-Flop component
   port (
      d, clk, reset, preset: in STD_LOGIC;
      q: out STD_LOGIC);
end D_FF;

architecture arch of D_FF is

COMPONENT DFF
PORT (
   d, clk, clrn, prn    : IN STD_LOGIC;
   q                    : OUT STD_LOGIC);
END COMPONENT;

   signal reset_l, preset_l: STD_LOGIC;
```

```
begin
    reset_l <= not reset;
    preset_l <= not preset;

    dff_lbl: DFF port map (
       d => d, clk => clk, clrn => reset_l, prn => preset_l, q => q);
end arch;

--------------------------------------------------------------------------------
-- D type Flip-Flop with Enable

library ieee;
use ieee.std_logic_1164.all;

entity D_FFE is          -- D type Flip-Flop with Enable component
    port (
       d, clk, ena, reset, preset: in STD_LOGIC;
       q: out STD_LOGIC
    );
end D_FFE;

architecture arch of D_FFE is

COMPONENT DFFE
PORT (
    d, clk, clrn, prn, ena  : IN STD_LOGIC;
    q                       : OUT STD_LOGIC);
END COMPONENT;

    signal reset_l, preset_l: STD_LOGIC;

begin
    reset_l <= not reset;
    preset_l <= not preset;

    dffe_lbl: DFFE port map (
       d => d, clk => clk, clrn => reset_l, prn => preset_l, ena => ena, q =>q);
end arch;

--------------------------------------------------------------------------------
-- JK type Flip-Flop

library ieee;
use ieee.std_logic_1164.all;

entity JK_FF is          -- JK type Flip-Flop component
    port (
       j, k, clk, reset, preset: in STD_LOGIC;
       q: out STD_LOGIC
    );
end JK_FF;

architecture arch of JK_FF is

COMPONENT JKFF
PORT (
    j, k, clk, clrn, prn    : IN STD_LOGIC;
    q                       : OUT STD_LOGIC);
END COMPONENT;

    signal reset_l, preset_l: STD_LOGIC;

begin
```

```
    reset_l <= not reset;
    preset_l <= not preset;

    jkff_lbl: JKFF port map (
       j => j, k => k, clk => clk, clrn => reset_l, prn => preset_l, q => q);
end arch;

-------------------------------------------------------------------------------
-- JK type Flip-Flop with Enable

library ieee;
use ieee.std_logic_1164.all;

entity JK_FFE is        -- JK type Flip-Flop with Enable component
   port (
      j, k, clk, ena, reset, preset: in STD_LOGIC;
      q: out STD_LOGIC
    );
end JK_FFE;

architecture arch of JK_FFE is

COMPONENT JKFFE
PORT (
   j, k, clk, clrn, prn, ena  : IN STD_LOGIC;
   q                          : OUT STD_LOGIC);
END COMPONENT;

   signal reset_l, preset_l: STD_LOGIC;

begin
   reset_l <= not reset;
   preset_l <= not preset;

   jkffe_lbl: JKFFE port map (
      j => j, k => k, clk => clk, clrn => reset_l, prn => preset_l,
      ena => ena, q => q);
end arch;

-------------------------------------------------------------------------------
-- T type Flip-Flop

library ieee;
use ieee.std_logic_1164.all;

entity T_FF is          -- T type Flip-Flop component
   port (
      t, clk, reset, preset: in STD_LOGIC;
      q: out STD_LOGIC
    );
end T_FF;

architecture arch of T_FF is

COMPONENT TFF
PORT (
   t, clk, clrn, prn    : IN STD_LOGIC;
   q                    : OUT STD_LOGIC);
END COMPONENT;

   signal reset_l, preset_l: STD_LOGIC;

begin
```

```
    reset_1 <= not reset;
    preset_1 <= not preset;

    tff_lbl: TFF port map (
       t => t, clk => clk, clrn => reset_1, prn => preset_1, q => q);
end arch;

-------------------------------------------------------------------------------
-- T type Flip-Flop with Enable

library ieee;
use ieee.std_logic_1164.all;

entity T_FFE is          -- T type Flip-Flop with Enable component
    port (
       t, clk, ena, reset, preset: in STD_LOGIC;
       q: out STD_LOGIC
    );
end T_FFE;

architecture arch of T_FFE is

COMPONENT TFFE
PORT (
    t, clk, clrn, prn, ena  : IN STD_LOGIC;
    q                       : OUT STD_LOGIC);
END COMPONENT;

    signal reset_1, preset_1: STD_LOGIC;

begin
    reset_1 <= not reset;
    preset_1 <= not preset;

    tffe_lbl: TFFE port map (
       t => t, clk => clk, clrn => reset_1, prn => preset_1, ena => ena, q =>q);
end arch;

-------------------------------------------------------------------------------
-- Generic Binary Counter

library ieee;
use ieee.std_logic_1164.all;

-- Signals ending in _1 are asserted LOW (0).
-- All the other signals are asserted HIGH (1).

entity Generic_Binary_Counter is                    -- Generic Counter component
    generic (Size: integer := 3);
    port (
       clk:      in STD_LOGIC;                      -- Counter Clock
       enable:   in STD_LOGIC;                      -- Enable Counting
       areset:   in STD_LOGIC;                      -- Counter Asynchronous Reset
       sreset:   in STD_LOGIC;                      -- Counter Synchronous Reset
       count:    out STD_LOGIC_VECTOR (Size-1 downto 0)-- Counter Output
    );
end Generic_Binary_Counter;

architecture arch of Generic_Binary_Counter is

    COMPONENT lpm_counter
    GENERIC (LPM_WIDTH : POSITIVE; LPM_DIRECTION : STRING);
    PORT (
```

```
        clk_en : IN STD_LOGIC ;
        sclr   : IN STD_LOGIC ;
        aclr   : IN STD_LOGIC ;
        clock  : IN STD_LOGIC ;
        q   : OUT STD_LOGIC_VECTOR (LPM_WIDTH-1 DOWNTO 0)
    );
    END COMPONENT;

begin
    lpm_counter_lbl: lpm_counter
    GENERIC MAP ( LPM_WIDTH => Size, LPM_DIRECTION => "UP")
    PORT MAP (clk_en => enable, sclr => sreset, aclr => areset,
       clock => clk, q => count);
end arch;

--------------------------------------------------------------------------------
-- Generic Register

library IEEE;
use IEEE.std_logic_1164.all;

entity Generic_Register is                          -- Generic Register component
   generic (RegSize: integer := 8);
    port (
       data:  in STD_LOGIC_VECTOR (RegSize-1 downto 0);-- Input Data signals
       clk:   in STD_LOGIC;                         -- Register Clock
       enable:in STD_LOGIC;                         -- Enable Data Buffering
       reset: in STD_LOGIC;                         -- Register Reset
       preset:in STD_LOGIC;                         -- Register Preset
       reg:   out STD_LOGIC_VECTOR (RegSize-1 downto 0)-- Buffered Data signals
    );
end Generic_Register;

architecture arch of Generic_Register is

   COMPONENT lpm_ff
   GENERIC (LPM_WIDTH : POSITIVE; LPM_FFTYPE : STRING);
   PORT (
      enable : IN STD_LOGIC ;
      aclr   : IN STD_LOGIC ;
      clock  : IN STD_LOGIC ;
      q      : OUT STD_LOGIC_VECTOR (LPM_WIDTH-1 DOWNTO 0);
      data   : IN STD_LOGIC_VECTOR (LPM_WIDTH-1 DOWNTO 0);
      aset   : IN STD_LOGIC
   );
   END COMPONENT;

begin
   lpm_ff_lbl: lpm_ff
   GENERIC MAP (LPM_WIDTH => RegSize, LPM_FFTYPE => "DFF")
   PORT MAP (clock => clk, enable => enable, aclr => reset, aset => preset,
      data => data, q => reg);
end arch;

--------------------------------------------------------------------------------
-- RAM

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY lpm;
USE lpm.lpm_components.all;
```

```
ENTITY ram IS
    GENERIC (
        ADDRESS_SIZE:    INTEGER := 6;
        DATA_SIZE:       INTEGER := 3);
    PORT (
        address    : IN STD_LOGIC_VECTOR (ADDRESS_SIZE-1 DOWNTO 0);
        we         : IN STD_LOGIC ;
        inclock    : IN STD_LOGIC ;
--      outclock   : IN STD_LOGIC ;
        data       : IN STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNTO 0);
        q          : OUT STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNTO 0));
END ram;

ARCHITECTURE SYN OF ram IS

    COMPONENT lpm_ram_dq
    GENERIC (
        LPM_WIDTH:              INTEGER := 3;
        LPM_TYPE:               STRING := "L_RAM_DQ";
        LPM_WIDTHAD:            INTEGER := 6;
        LPM_NUMWORDS:           STRING := "UNUSED";
        LPM_FILE:               STRING := "UNUSED";
        LPM_INDATA:             STRING := "REGISTERED";
        LPM_ADDRESS_CONTROL:    STRING := "REGISTERED";
        LPM_OUTDATA:            STRING := "REGISTERED";
        LPM_HINT:               STRING := "UNUSED");
    PORT (
        data:      IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
        address:   IN STD_LOGIC_VECTOR(LPM_WIDTHAD-1 DOWNTO 0);
        we:        IN STD_LOGIC := '1';
        inclock:   IN STD_LOGIC := '1';
        outclock:  IN STD_LOGIC := '1';
        q:         OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
    END COMPONENT;

BEGIN
    LPM_RAM_DQ_component : LPM_RAM_DQ
    GENERIC MAP (
        LPM_WIDTH     => DATA_SIZE,
        LPM_WIDTHAD   => ADDRESS_SIZE,
        LPM_OUTDATA   => "UNREGISTERED")
    PORT MAP (
        Data       => data,
        address    => address,
        we         => we,
        inclock    => inclock,
--      outclock   => outclock,
        q          => q);
END SYN;

--------------------------------------------------------------------------------
-- ROM

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY lpm;
USE lpm.lpm_components.all;

entity rom is
    generic (
        ROM_ADDRESS_SIZE:    INTEGER := 4;
        DATA_SIZE:           INTEGER := 2;
```

```
        DATA_FILE:          STRING := "rom.mif");
    port (
        address    : in STD_LOGIC_VECTOR (ROM_ADDRESS_SIZE-1 downto 0);
--      inclock    : in STD_LOGIC ;
--      outclock   : in STD_LOGIC ;
        q          : out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0));
end rom;

architecture SYN of rom is

    COMPONENT LPM_ROM
    GENERIC (
        LPM_WIDTH            : POSITIVE;
        LPM_WIDTHAD          : POSITIVE;
        LPM_ADDRESS_CONTROL  : STRING;
        LPM_INDATA           : STRING;
        LPM_OUTDATA          : STRING;
        LPM_FILE             : STRING);
    PORT (
          outclock  : IN STD_LOGIC := '1';
          address   : IN STD_LOGIC_VECTOR (LPM_WIDTHAD-1 DOWNTO 0);
          inclock   : IN STD_LOGIC := '1';
          q         : OUT STD_LOGIC_VECTOR (LPM_WIDTH-1 DOWNTO 0));
    END COMPONENT;

begin
    LPM_ROM_component : LPM_ROM
    GENERIC MAP (
        LPM_WIDTH            => DATA_SIZE,
        LPM_WIDTHAD          => ROM_ADDRESS_SIZE,
        LPM_ADDRESS_CONTROL  => "UNREGISTERED",
        LPM_INDATA           => "UNREGISTERED",
        LPM_OUTDATA          => "UNREGISTERED",
        LPM_FILE             => DATA_FILE)
    PORT MAP (
        address    => address,
--      inclock    => inclock,
--      outclock   => outclock,
        q          => q);
end SYN;
```

VHDL Library

```
library ieee;
use ieee.std_logic_1164.all;

package VHDL_package is

    component D_2FF         -- D type double Flip-Flop component
    port (
        d, clk, reset, preset: in STD_LOGIC;
        q: out STD_LOGIC);
    end component;

    component Pulse         -- Generates a Pulse on every edge of a signal
    port (
        d, clk, reset, preset: in STD_LOGIC;
        rise, fall: out STD_LOGIC);
    end component;

    component Generic_Encoder          -- Generic Encoder component
    generic (
        IN_SIZE: INTEGER := 8;         -- number of inputs
        OUT_WIDTH: INTEGER := 3);      -- width of the encoded output
     port (
        a: in STD_LOGIC_VECTOR (IN_SIZE-1 downto 0);
        y: out STD_LOGIC_VECTOR (OUT_WIDTH-1 downto 0));
    end component;

end VHDL_package;
```

VHDL Library Components

```
--------------------------------------------------------------------------------
-- D type double Flip-Flop

library ieee;
use ieee.std_logic_1164.all;

-- Signals ending in _1 are asserted LOW (0).
-- All the other signals are asserted HIGH (1).

entity D_2FF is         -- D type double Flip-Flop component
    port (
        d, clk, reset, preset: in STD_LOGIC;
        q: out STD_LOGIC);
end D_2FF;

architecture arch of D_2FF is

    component D_FF              -- D type double Flip-Flop component
    port (
        d, clk, reset, preset: in STD_LOGIC;
        q: out STD_LOGIC);
    end component;

    signal q_buffer: STD_LOGIC_VECTOR (2 downto 1);
    signal HIGH, LOW:   STD_LOGIC;

begin
    LOW <= '0';
    HIGH <= '1';
```

```
    q1_lbl: D_FF port map (clk => clk, reset => LOW, preset => LOW,
       d => d, q => q_buffer(1));
    q2_lbl: D_FF port map (clk => clk, reset => LOW, preset => LOW,
       d => q_buffer(1), q => q_buffer(2));

    q <= q_buffer(2);
end arch;

--------------------------------------------------------------------------------
-- Generates a Pulse on every edge of a signal

library ieee;
use ieee.std_logic_1164.all;

entity Pulse is          -- Generates a Pulse on every edge of a signal
    port (
       d, clk, reset, preset: in STD_LOGIC;
       rise, fall: out STD_LOGIC
    );
end Pulse;

architecture arch of Pulse is

component D_FF           -- D type Flip-Flop component
port (
    d, clk, reset, preset: in STD_LOGIC;
    q: out STD_LOGIC);
end component;

    signal q, rise_d, fall_d: STD_LOGIC;

begin

    delay_signal_lbl: D_FF port map (
       d => d, clk => clk, reset => reset, preset => preset, q => q);
    rise <= '1' when (d = '1' and q = '0') else '0';
    fall <= '1' when (d = '0' and q = '1') else '0';

-- rise_lbl: D_FF port map (
--     d => rise_d, clk => clk, reset => reset, preset => preset, q => rise);
-- fall_lbl: D_FF port map (
--     d => fall_d, clk => clk, reset => reset, preset => preset, q => fall);

end arch;

--------------------------------------------------------------------------------
-- Generic Encoder

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

-- Signals ending in _l are asserted LOW (0).
-- All the other signals are asserted HIGH (1).

entity Generic_Encoder is                -- Generic Encoder component
    generic (
       IN_SIZE: INTEGER := 8;            -- number of inputs
       OUT_WIDTH: INTEGER := 3);             -- width of the encoded output
    port (
       a: in STD_LOGIC_VECTOR (IN_SIZE-1 downto 0);
       y: out STD_LOGIC_VECTOR (OUT_WIDTH-1 downto 0));
```

```
end Generic_Encoder;

architecture arch of Generic_Encoder is

    signal HIGH, LOW: STD_LOGIC;

begin
    HIGH <= '1';
    LOW  <= '0';

    process(a)
    begin
        y <= (others => '-');
        for i in 0 to IN_SIZE-1 loop
            if a(i) = '1' then
                y <= CONV_STD_LOGIC_VECTOR(i, OUT_WIDTH);
            end if;
        end loop;
    end process;

end arch;
```

A-2. PRDEC Library

```
library ieee;
use ieee.std_logic_1164.all;

package PRDEC_package is

component GF_ShiftRegister
                        -- Data Storage Cell Chain component with Selection Taps
generic (
    GF_WIDTH:           INTEGER := 2;     -- width of GF elements
    MAX_POLYNOM_DEGREE: INTEGER: = 16);  -- maximum degree of the polynomial
port (
    serial_in:in STD_LOGIC_VECTOR (GF_WIDTH-1 downto 0);-- Serial Data input
    clk:   in STD_LOGIC;                  -- Clock
    tap:   in STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE-1 downto 0);-- Tap Select
    enable:       in STD_LOGIC;                      -- Enable Shift Operation
    reset:        in STD_LOGIC;                      -- Asynchronous Reset
    serial_out:   out STD_LOGIC_VECTOR (GF_WIDTH-1 downto 0); -- serial output
    parallel_out: out STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE*GF_WIDTH-1 downto 0)
                  -- parallel output sequence (window)
    );
end component;

component Taps              -- Tap Decoder component
generic (WIDTH: INTEGER := 8; MAX_POLYNOM_DEGREE: INTEGER := 16);
port (
    data:         in STD_LOGIC_VECTOR (WIDTH-1 downto 0);
    load_tap:     in STD_LOGIC;
    reset, clk:      in STD_LOGIC;
    tap:          out STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE-1 downto 0));
end component;

component Tap_FIFO          -- FIFO with TAPS (variable entry point) component
generic (
       WIDTH: INTEGER := 8;    -- data width
       NUMWORDS: INTEGER := 3  -- FIFO depth);
port (
    data       : in STD_LOGIC_VECTOR (WIDTH-1 downto 0);
    tap        : in STD_LOGIC_VECTOR (NUMWORDS-1 downto 0);   -- Tap Select
    wrreq      : in STD_LOGIC ;
    rdreq      : in STD_LOGIC ;
    clock      : in STD_LOGIC ;
    reset      : in STD_LOGIC ;
    q          : out STD_LOGIC_VECTOR (WIDTH-1 downto 0);
    all_q      : out STD_LOGIC_VECTOR (NUMWORDS*WIDTH-1 downto 0));
    end component;

component GF_Generator      -- Generic GF Generator component
generic (
    GF_WIDTH:               INTEGER := 2; -- width of GF elements
    DOUBLE_GF_WIDTH:        INTEGER := 4;
                  -- width of entry address for GF operation tables = 2*GF_WIDTH
    MAX_POLYNOM_DEGREE:     INTEGER := 4; -- maximum degree of the polynomial
    POLYNOM_DEGREE_WIDTH:   INTEGER := 2
                  -- width of MAX_POLYNOM_DEGREE = LOG2(MAX_POLYNOM_DEGREE)
    );
port (
    data:         in STD_LOGIC_VECTOR (GF_WIDTH-1 downto 0);-- serial data input
    row_address:  in STD_LOGIC_VECTOR (GF_WIDTH-1 downto 0);-- row address input
```

```
    col_address: in STD_LOGIC_VECTOR (GF_WIDTH-1 downto 0);-- col address input
    clk:         in STD_LOGIC;                          -- Clock
    polynom_degree: in STD_LOGIC_VECTOR (POLYNOM_DEGREE_WIDTH-1 downto 0);
                              -- The Degree of the Primitive Polynomial minus 1
    tap:   in STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE-1 downto 0);-- Tap Select
    load_mul:     in STD_LOGIC;        -- Load Multiply Table
    load_add:     in STD_LOGIC;        -- Load Addition Table
    load_coef:    in STD_LOGIC;        -- Load Polynomial Coefficients
    load_window:  in STD_LOGIC;        -- Load Window Sequence
    enable:       in STD_LOGIC;        -- Enable Operation
    reset:        in STD_LOGIC;        -- Asynchronous Reset
    symbol:       out STD_LOGIC_VECTOR (GF_WIDTH-1 downto 0);-- output symbol
    available:    out STD_LOGIC;       -- new symbol is available from Generator
    valid:        out STD_LOGIC;       -- PRS window valid
    match:        in STD_LOGIC;        -- Matched output symbol
    window:       out STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE*GF_WIDTH-1 downto 0)
                              -- output sequence
    );
    end component;

component GF_PRS_Milestones   -- Generic GF Milestone Storage Memory component
generic (
    DATA_SIZE: INTEGER := 8;          -- the data bus width
    MAX_POLYNOM_DEGREE: INTEGER := 4; -- maximum degree of the polynomial
    POLYNOM_DEGREE_WIDTH: INTEGER := 3;
                -- width of MAX_POLYNOM_DEGREE = LOG2(MAX_POLYNOM_DEGREE)
    MILESTONE_SIZE: INTEGER := 8;-- the size (in binary bits) of a milestone
                -- word saved in FIFO equal to MAX_POLYNOM_DEGREE * GF_WIDTH
    MAX_NUMWORDS: INTEGER := 6;    -- maximum number of milestones saved in FIFO
                                   -- equal to MAX_POLYNOM_DEGREE + 2 (latency)
    MILESTONE_CLUSTERS: INTEGER := 4; -- number of milestone clusters
    MILESTONE_CLUSTER_WIDTH: INTEGER := 2
                -- milestone cluster width equal to LOG2(MILESTONE_CLUSTERS)
    );
port (
    window:in STD_LOGIC_VECTOR (MILESTONE_SIZE-1 downto 0);  -- input PRS window
    clk:   in STD_LOGIC;                                     -- clock
    polynom_degree: in STD_LOGIC_VECTOR (POLYNOM_DEGREE_WIDTH-1 downto 0);
              -- The Degree of the Irreducible Polynomial minus 1
    period:in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
              -- GF milestones period minus 1
    tap:   in STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE-1 downto 0);-- tap selector
    available:    in STD_LOGIC;       -- new sequence window is available
    ready: in STD_LOGIC;              -- flag: ready for decoding,
              -- the algebra was loaded and the milestone table completed
    decode:in STD_LOGIC;              -- flag: decode the current PRS Window
    reset: in STD_LOGIC;              -- reset milestones
    milestone:out STD_LOGIC_VECTOR(MILESTONE_CLUSTERS*MILESTONE_SIZE-1 downto
0); -- output milestones window
    milestone_count: out STD_LOGIC_VECTOR (POLYNOM_DEGREE_WIDTH-1 downto 0);
                                      -- milestone index
    milestone_overflow: out STD_LOGIC);
end component;

component Control          -- Coprocessor Control Unit component
generic (
    DATA_SIZE:          INTEGER := 8;    -- width of the data bus
    MAX_POLYNOM_DEGREE: INTEGER := 16);  -- maximum degree of the polynomial
port (
    address:  in STD_LOGIC;                  -- address bus from µP
    data_in:  in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);-- data bus in from µP
    data_out: out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                                         -- data bus out to the other modules
```

```
        wr:    in STD_LOGIC;                    -- write enable
        clk:   in STD_LOGIC;     -- clock
        reset: in STD_LOGIC;     -- reset
        table_row_address:  out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                            -- row address for the current table
        table_col_address:  out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                            -- column address for the current table
        load_add: out STD_LOGIC;   -- load Addition Table into RAM
        load_mul: out STD_LOGIC;   -- load Multiplication Table into RAM
        load_coef:out STD_LOGIC;   -- load Polinomial Coefficients Table into RAM
        load_window:out STD_LOGIC; -- load PRS Window into RAM
        polynom_degree:  out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                            -- The Degree of the Polynomial minus 1
        tap: out STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE-1 downto 0);  -- Tap Select
        milestone_period:   out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
        decode:out STD_LOGIC;      -- status flag: decode the current PRS Window
        ready: out STD_LOGIC;      -- status flag: ready for decoding,
                    -- the algebra was loaded and the milestone table completed
        algebra_done:out STD_LOGIC;-- status flag: the algebra was loaded
        overflow: out STD_LOGIC;   -- status flag: Milestone Database overflowed
        match: in STD_LOGIC;    -- PRS Window decoded
        milestone_overflow: in STD_LOGIC; -- Milestone Database overflowed
        irq:   out STD_LOGIC);  -- interrupt request to µP on PRS window decoded
    end component;

component GF_PRS_Decoder                -- GF PRS Decoder Unit component
generic (
    DATA_SIZE: INTEGER := 8;            -- width of the data bus
    POLYNOM_DEGREE_WIDTH: INTEGER := 3;
                -- width of MAX_POLYNOM_DEGREE = LOG2(MAX_POLYNOM_DEGREE) + 1
                -- rounded up to the near integer
    MILESTONE_SIZE: INTEGER := 8;
                -- the size (in binary bits) of a milestone word saved in FIFO
                -- equal to MAX_POLYNOM_DEGREE * GF_WIDTH
    MILESTONE_CLUSTERS: INTEGER := 4;    -- number of milestone clusters
    MILESTONE_CLUSTER_WIDTH: INTEGER := 2
                -- milestone cluster width equal to LOG2(MILESTONE_CLUSTERS)
    );
port (
    milestone_count: in STD_LOGIC_VECTOR (POLYNOM_DEGREE_WIDTH-1 downto 0);
                -- milestone pointer (not synchronized)
    window: in STD_LOGIC_VECTOR (MILESTONE_SIZE-1 downto 0); -- PRS Window input
    milestone: in STD_LOGIC_VECTOR (MILESTONE_CLUSTERS*MILESTONE_SIZE-1 downto
0); -- Milestones Window
    symbol_available:   in STD_LOGIC; -- new symbol is available from Generator
    clk: in STD_LOGIC;                -- Clock
    reset: in STD_LOGIC;              -- Asynchronous Reset
    enable: in STD_LOGIC;             -- decoding enable
    ready: in STD_LOGIC;              -- flag: ready for decoding,
                -- the algebra was loaded and the milestone table completed
    milestone_offset:out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    milestone_pointer:  out STD_LOGIC_VECTOR (POLYNOM_DEGREE_WIDTH-1 downto 0);
                -- milestone index (synchronized)
    milestone_cluster:out STD_LOGIC_VECTOR (MILESTONE_CLUSTER_WIDTH-1 downto 0);
                -- milestone cluster (synchronized)
    match:    out STD_LOGIC);         -- PRS Window matched Milestone
end component;

component PRDEC_Core           -- Absolute Position Recovery Core component
generic (
    DATA_SIZE: INTEGER := 8;          -- width of the data bus
    GF_WIDTH: INTEGER := 2;           -- width of GF elements
    DOUBLE_GF_WIDTH: INTEGER := 4;
```

```
                -- width of entry address for GF operation tables = 2*GF_WIDTH
    MAX_POLYNOM_DEGREE: INTEGER := 4; -- maximum degree of the polynomial
    POLYNOM_DEGREE_WIDTH: INTEGER := 3;
                -- width of MAX_POLYNOM_DEGREE = LOG2(MAX_POLYNOM_DEGREE) + 1
                -- rounded up to the near integer
    MILESTONE_SIZE: INTEGER := 8;
                -- the size (in binary bits) of a milestone word saved in FIFO
                -- equal to MAX_POLYNOM_DEGREE * GF_WIDTH
    MAX_NUMWORDS: INTEGER := 6;    -- maximum number of milestones saved in FIFO
                -- equal to MAX_POLYNOM_DEGREE + 2 (latency)
    MILESTONE_CLUSTERS: INTEGER := 4;     -- number of milestone clusters
    MILESTONE_CLUSTER_WIDTH: INTEGER := 2
                -- milestone cluster width equal to LOG2(MILESTONE_CLUSTERS)
    );
 port (
    address:  in STD_LOGIC;
    data_in:  in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    wr:       in STD_LOGIC;
    irq:      out STD_LOGIC;   -- interrupt request to µP on PRS window decoded
    clk, reset:in STD_LOGIC;
    status:   out STD_LOGIC_VECTOR (3 downto 0);   -- Status Register
    gf_milestone_offset:out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    gf_milestone_pointer:out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    gf_milestone_cluster:out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    gf_symbol:    out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    gf_sync:      out STD_LOGIC);-- new symbol is available from Generator
    end component;

end PRDEC_package;
```

PRDEC Library Components

```
--------------------------------------------------------------------------------
-- Data Storage Cell Chain with Selection Taps

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

library work;
use work.VENDOR_package.all;

-- Signals ending in _l are asserted LOW (0).
-- All the other signals are asserted HIGH (1).

entity GF_ShiftRegister is
generic (
    GF_WIDTH: INTEGER := 2;                -- width of GF elements
    MAX_POLYNOM_DEGREE: INTEGER := 16);  -- maximum degree of the polynomial
port (
    serial_in:    in STD_LOGIC_VECTOR (GF_WIDTH-1 downto 0);-- Serial Data input
    clk:       in STD_LOGIC;               -- Clock
    tap:       in STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE-1 downto 0);-- Tap Select
    enable:    in STD_LOGIC;               -- Enable Shift Operation
    reset:     in STD_LOGIC;               -- Asynchronous Reset
    serial_out:out STD_LOGIC_VECTOR (GF_WIDTH-1 downto 0);-- serial output
    parallel_out: out STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE*GF_WIDTH-1 downto 0)
                                           -- parallel output sequence (window)
);
end GF_ShiftRegister;
```

```
architecture arch of GF_ShiftRegister is

    signal HIGH, LOW: STD_LOGIC;
    signal chain_d: STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE*GF_WIDTH-1 downto 0);
    signal chain_d, chain_q: STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE*GF_WIDTH-1
downto 0);

begin
    HIGH <= '1';
    LOW <= '0';

    ------------------------------------------------------------------
    -- The input of the first cell in chain (index MAX_POLYNOM_DEGREE) is
    -- always connected to the serial input
    Cell_Depth_lbl: Generic_Register generic map (GF_WIDTH)
    port map (
        data => chain_d(MAX_POLYNOM_DEGREE*GF_WIDTH-1 downto
                        (MAX_POLYNOM_DEGREE-1)*GF_WIDTH),
        clk, enable, reset, preset => LOW,
        reg => chain_q(MAX_POLYNOM_DEGREE*GF_WIDTH-1 downto
                        (MAX_POLYNOM_DEGREE-1)*GF_WIDTH));

    chain_d(MAX_POLYNOM_DEGREE*GF_WIDTH-1 downto
              (MAX_POLYNOM_DEGREE-1)*GF_WIDTH) <=
        serial_in when tap(MAX_POLYNOM_DEGREE-1) = '1' else
        CONV_STD_LOGIC_VECTOR(0, GF_WIDTH);

    ------------------------------------------------------------------
    Cell_Chain_lbl: for i in MAX_POLYNOM_DEGREE-1 downto 1 generate
        Cell_Sel_lbl: Generic_Register generic map (GF_WIDTH)
        port map (
            data => chain_d(i*GF_WIDTH-1 downto (i-1)*GF_WIDTH),
            clk, enable, reset, preset => LOW,
            reg => chain_q(i*GF_WIDTH-1 downto (i-1)*GF_WIDTH));

        chain_d(i*GF_WIDTH-1 downto (i-1)*GF_WIDTH) <=
               serial_in when tap(i-1) = '1' else
               chain_q((i+1)*GF_WIDTH-1 downto i*GF_WIDTH);
    end generate;

    ------------------------------------------------------------------
    parallel_out <= chain_q(MAX_POLYNOM_DEGREE*GF_WIDTH-1 downto 0);
    serial_out <= chain_q(GF_WIDTH-1 downto 0);

end arch;

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-- Tap Decoder

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

library work;
use work.VENDOR_package.all;
use work.PRDEC_package.all;

-- Signals ending in _l are asserted LOW (0).
-- All the other signals are asserted HIGH (1).

entity Taps is                              -- -- Tap Decoder component
    generic (WIDTH: INTEGER := 8; MAX_POLYNOM_DEGREE: INTEGER := 16);
```

```
    port (
       data:         in STD_LOGIC_VECTOR (WIDTH-1 downto 0);
       load_tap:     in STD_LOGIC;
       reset, clk:   in STD_LOGIC;
       tap:          out STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE-1 downto 0)
    );
end Taps;

architecture arch of Taps is

    signal HIGH, LOW: STD_LOGIC;
    signal encoded_tap: STD_LOGIC_VECTOR (WIDTH-1 downto 0);

begin
    HIGH <= '1';
    LOW  <= '0';

-------------------------------------------------------------------------
    taps_lbl: Generic_Register generic map (RegSize => WIDTH)
    port map (clk => clk, reset => reset, preset => LOW,
       data => data, enable => load_tap, reg => encoded_tap);

    decoder_lbl: process (encoded_tap) begin
       for i in 0 to MAX_POLYNOM_DEGREE-1 loop
          if (encoded_tap = CONV_STD_LOGIC_VECTOR(i, WIDTH)) then
             tap(i) <= '1';
          else
             tap(i) <= '0';
          end if;
       end loop;
    end process;

end arch;


-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-- FIFO with TAPS (variable entry point)

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

LIBRARY work;
USE work.VENDOR_package.ALL;

entity Tap_FIFO is            -- FIFO with TAPS (variable entry point)
component
   generic (
      WIDTH: INTEGER := 8;    -- data width
      NUMWORDS: INTEGER := 3  -- FIFO depth
   );
   port (
      data      : in STD_LOGIC_VECTOR (WIDTH-1 downto 0);
      tap       : in STD_LOGIC_VECTOR (NUMWORDS-1 downto 0);-- Tap Select
      wrreq     : in STD_LOGIC ;
      rdreq     : in STD_LOGIC ;
      clock     : in STD_LOGIC ;
      reset     : in STD_LOGIC ;
      q         : out STD_LOGIC_VECTOR (WIDTH-1 downto 0);
      all_q     : out STD_LOGIC_VECTOR (NUMWORDS*WIDTH-1 downto 0)
   );
end Tap_FIFO;
```

```
architecture arch of Tap_FIFO is

    signal HIGH, LOW: STD_LOGIC;
    signal fifo_ena: STD_LOGIC;
    signal fifo_in: STD_LOGIC_VECTOR (WIDTH-1 downto 0);
    signal reg: STD_LOGIC_VECTOR (NUMWORDS*WIDTH-1 downto 0);
    signal fifo_d: STD_LOGIC_VECTOR (NUMWORDS*WIDTH-1 downto 0);

begin
    HIGH <= '1';
    LOW  <= '0';

--------------------------------------------------------------------------------
    fifo_in <= data when wrreq = '1' else reg(WIDTH-1 downto 0);
    fifo_ena <= '1' when wrreq = '1' or rdreq = '1' else '0';

    fifo_cellNUMWORDS_lbl: Generic_Register generic map (WIDTH)
    port map (
        data => fifo_d(NUMWORDS*WIDTH-1 downto (NUMWORDS-1)*WIDTH),
        clk => clock, enable => fifo_ena, reset => reset,
        reg => reg(NUMWORDS*WIDTH-1 downto (NUMWORDS-1)*WIDTH), preset => LOW);

    fifo_d(NUMWORDS*WIDTH-1 downto (NUMWORDS-1)*WIDTH) <=
                fifo_in when tap(NUMWORDS-1) = '1' else
                CONV_STD_LOGIC_VECTOR(0, WIDTH);

    fifo_lbl: for i in NUMWORDS-1 downto 1 generate
        fifo_cell_lbl: Generic_Register generic map (WIDTH)
        port map (
            data => fifo_d(i*WIDTH-1 downto (i-1)*WIDTH),
            clk => clock, enable => fifo_ena, reset => reset,
            reg => reg(i*WIDTH-1 downto (i-1)*WIDTH), preset => LOW);

        fifo_d(i*WIDTH-1 downto (i-1)*WIDTH) <=
                fifo_in when tap(i-1) = '1' else
                reg((i+1)*WIDTH-1 downto i*WIDTH);
    end generate;

    q <= reg(WIDTH-1 downto 0);
    all_q <= reg;

end arch;

--------------------------------------------------------------------------------
```

PRDEC Core

```
-- Absolute Position Recovery Core

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

library work;
use work.VENDOR_package.all;
use work.VHDL_package.all;
use work.PRDEC_package.all;

-- Signals ending in _l are asserted LOW (0).
-- All the other signals are asserted HIGH (1).

entity PRDEC_Core is                -- Absolute Position Recovery Core component
generic (
    DATA_SIZE: INTEGER := 8;            -- width of the data bus
    GF_WIDTH: INTEGER := 2;             -- width of GF elements
    DOUBLE_GF_WIDTH: INTEGER := 4;
                -- width of entry address for GF operation tables = 2*GF_WIDTH
    MAX_POLYNOM_DEGREE: INTEGER := 4; -- maximum degree of the polynomial
    POLYNOM_DEGREE_WIDTH: INTEGER := 3;
                -- width of MAX_POLYNOM_DEGREE = LOG2(MAX_POLYNOM_DEGREE) + 1
                -- rounded up to the near integer
    MILESTONE_SIZE: INTEGER := 8;
                -- the size (in binary bits) of a milestone word saved in FIFO
                -- equal to MAX_POLYNOM_DEGREE * GF_WIDTH
    MAX_NUMWORDS: INTEGER := 6;     -- maximum number of milestones saved in FIFO
                                    -- to MAX_POLYNOM_DEGREE + 2 (latency)
    MILESTONE_CLUSTERS: INTEGER := 4;     -- number of milestone clusters
    MILESTONE_CLUSTER_WIDTH: INTEGER := 2
                -- milestone cluster width equal to LOG2(MILESTONE_CLUSTERS)
);
port (
     address:     in STD_LOGIC;
     data_in:     in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
     wr:          in STD_LOGIC;
     irq:  out STD_LOGIC;       -- interrupt request to µP on PRS window decoded
     clk, reset:  in STD_LOGIC;
    status:       out STD_LOGIC_VECTOR (3 downto 0);   -- Status Register
    gf_milestone_offset:out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    gf_milestone_pointer:out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                    -- Milestone Pointer (synchronized)
    gf_milestone_cluster:out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                    -- Milestone Cluster (synchronized)
    gf_symbol:    out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);-- output symbol
    gf_sync:      out STD_LOGIC         -- new symbol is available from Generator
);
end PRDEC_Core;

architecture arch of PRDEC_Core is

    signal HIGH, LOW: STD_LOGIC;
    signal data: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal row_address, col_address: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal milestone_period: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal load_mul, load_add, load_coef, load_tap, load_window: STD_LOGIC;
    signal symbol: STD_LOGIC_VECTOR (GF_WIDTH-1 downto 0);-- output symbol
    signal window_sequence: STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE*GF_WIDTH-1
                                              downto 0);-- output sequence
```

```
    signal algebra_done, decode, ready, overflow, symbol_available: STD_LOGIC;
    signal window_sequence_available, sequence_matched: STD_LOGIC;
    signal polynom_degree: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                           -- The Degree of the Primitive Polynomial minus 1
    signal tap: STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE-1 downto 0);   -- Tap Select
    signal reset_milestone, milestone_overflow: STD_LOGIC;
    signal milestone_count: STD_LOGIC_VECTOR (POLYNOM_DEGREE_WIDTH-1 downto 0);
    signal milestone_pointer: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal milestone_offset: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal buff_milestone_pointer: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal milestone: STD_LOGIC_VECTOR (MILESTONE_CLUSTERS*MILESTONE_SIZE-1
                                        downto 0);
    signal window_valid: STD_LOGIC;   -- PRS window valid

begin
    HIGH <= '1';
    LOW  <= '0';

-------------------------------------------------------------------------------
    -- Coprocessor Control Unit component
    Control_lbl: Control
    generic map (
       DATA_SIZE => DATA_SIZE,                    -- width of the data bus
       MAX_POLYNOM_DEGREE => MAX_POLYNOM_DEGREE)   -- maximum degree polynomial
    port map (
       address => address,                        -- address bus from µP
       data_in => data_in(DATA_SIZE-1 downto 0),  -- data bus in from µP
       data_out => data(DATA_SIZE-1 downto 0),    -- data bus out
       wr => wr,                                  -- write enable
       reset => reset,                            -- reset
       clk => clk,                                -- clock
       load_add => load_add,      -- load Addition Table into RAM
       load_mul => load_mul,      -- load Multiplication Table into RAM
       load_coef => load_coef,    -- load Polinomial Coefficients Table into RAM
       load_window => load_window,-- load PRS Window into RAM
       table_row_address => row_address(DATA_SIZE-1 downto 0),-- row address
       table_col_address => col_address(DATA_SIZE-1 downto 0),-- column address
       polynom_degree => polynom_degree(DATA_SIZE-1 downto 0),
                         -- The Degree of the Primitive Polynomial minus 1
       milestone_period => milestone_period(DATA_SIZE-1 downto 0),
                         -- The Milestone Period minus 1
       tap => tap(MAX_POLYNOM_DEGREE-1 downto 0),           -- Tap Select
       algebra_done => algebra_done,     -- status flag: the algebra was loaded
       ready => ready,           -- status flag: ready for decoding, the algebra
                                 -- was loaded and the milestone table completed
       decode => decode,         -- status flag: decode the current PRS Window
       overflow => overflow,     -- status flag: Milestone Database overflowed
       match => sequence_matched,        -- PRS Window decoded
       milestone_overflow => milestone_overflow,-- Milestone Database overflowed
       irq => irq                -- interrupt request to µP on PRS window decoded
    );

-------------------------------------------------------------------------------
    -- Generic GF Generator component
    GF_Generator_lbl: GF_Generator
    generic map (
       GF_WIDTH => GF_WIDTH,                  -- width of GF elements
       DOUBLE_GF_WIDTH => DOUBLE_GF_WIDTH,
                -- width of entry address for GF operation tables = 2*GF_WIDTH
       MAX_POLYNOM_DEGREE => MAX_POLYNOM_DEGREE,
                -- maximum degree of the primitive polynomial
       POLYNOM_DEGREE_WIDTH => POLYNOM_DEGREE_WIDTH
                -- width of MAX_POLYNOM_DEGREE = LOG2(MAX_POLYNOM_DEGREE)
```

```
    )
    port map (
        data => data(GF_WIDTH-1 downto 0),              -- serial data input
        row_address => row_address(GF_WIDTH-1 downto 0),-- row address input
        col_address => col_address(GF_WIDTH-1 downto 0),-- column address input
        clk => clk,                                     -- Clock
        reset => reset,                                 -- Asynchronous Reset
        load_mul => load_mul,                           -- Load Multiply Table
        load_add => load_add,                           -- Load Addition Table
        load_coef => load_coef,                         -- Load Polynomial Coefficients
        load_window => load_window,                     -- Load Window Sequence
        polynom_degree => polynom_degree(POLYNOM_DEGREE_WIDTH-1 downto 0),
                          -- The Degree of the Primitive Polynomial minus 1
        tap => tap(MAX_POLYNOM_DEGREE-1 downto 0),  -- Tap Select
        enable => decode,                               -- Enable Operation
        symbol => symbol(GF_WIDTH-1 downto 0),      -- output symbol
        available => symbol_available,-- new symbol is available from Generator
        valid => window_valid,          -- PRS window valid
        match => sequence_matched,      -- Matched output symbol sequence
        window => window_sequence(MAX_POLYNOM_DEGREE*GF_WIDTH-1 downto 0)
                                        -- output sequence window
    );

-------------------------------------------------------------------------------
    -- Generic GF Milestone Storage Memory component
    Milestones_lbl: GF_PRS_Milestones
    generic map (
        DATA_SIZE        => DATA_SIZE,          -- the data bus width
        MAX_POLYNOM_DEGREE => MAX_POLYNOM_DEGREE,
                -- maximum degree of the primitive polynomial
        POLYNOM_DEGREE_WIDTH => POLYNOM_DEGREE_WIDTH,
                -- width of MAX_POLYNOM_DEGREE = LOG2(MAX_POLYNOM_DEGREE)
        MILESTONE_SIZE => MILESTONE_SIZE,
                -- the size (in binary bits) of a milestone word saved in FIFO
                -- equal to MAX_POLYNOM_DEGREE * GF_WIDTH
        MAX_NUMWORDS => MAX_NUMWORDS,
                -- maximum number of milestones saved in FIFO
                -- equal to MAX_POLYNOM_DEGREE + 2 (latency)
        MILESTONE_CLUSTERS => MILESTONE_CLUSTERS, -- number of milestone clusters
        MILESTONE_CLUSTER_WIDTH => MILESTONE_CLUSTER_WIDTH
                -- milestone cluster width equal to LOG2(MILESTONE_CLUSTERS)
    )
    port map (
        window => window_sequence(MILESTONE_SIZE-1 downto 0), -- input PRS window
        clk => clk,             -- clock
        polynom_degree => polynom_degree(POLYNOM_DEGREE_WIDTH-1 downto 0),
                                -- The Degree of the Primitive Polynomial minus 1
        period => milestone_period(DATA_SIZE-1 downto 0),
                                -- GF milestones period minus 1
        tap => tap(MAX_POLYNOM_DEGREE-1 downto 0), -- tap selector
        available => symbol_available,    -- new sequence window is available
        ready => ready,                   -- flag: ready for decoding,
                    -- the algebra was loaded and the milestone table completed
        decode => window_valid,           -- flag: decode the current PRS Window
        reset => reset_milestone,         -- reset
        milestone => milestone(MILESTONE_CLUSTERS*MILESTONE_SIZE-1 downto 0),
                                          -- output milestones window
        milestone_count => milestone_count(POLYNOM_DEGREE_WIDTH-1 downto 0),
                                          -- milestone pointer (not sychronized)
        milestone_overflow => milestone_overflow
    );
    reset_milestone <= '1' when algebra_done = '0' else '0';
```

```
-------------------------------------------------------------------------
    -- GF PRS Decoder Unit component
    GF_PRS_Decoder_1b1: GF_PRS_Decoder
    generic map (
        DATA_SIZE => DATA_SIZE,                          -- width of the data bus
        POLYNOM_DEGREE_WIDTH => POLYNOM_DEGREE_WIDTH,
                  -- width of MAX_POLYNOM_DEGREE = LOG2(MAX_POLYNOM_DEGREE)
        MILESTONE_SIZE => MILESTONE_SIZE,
                  -- the size (in binary bits) of a milestone word saved in FIFO
                  -- equal to MAX_POLYNOM_DEGREE * GF_WIDTH
        MILESTONE_CLUSTERS => MILESTONE_CLUSTERS,-- number of milestone clusters
        MILESTONE_CLUSTER_WIDTH => MILESTONE_CLUSTER_WIDTH
                  -- milestone cluster width equal to LOG2(MILESTONE_CLUSTERS)
    )
    port map (
        milestone_count => milestone_count(POLYNOM_DEGREE_WIDTH-1 downto 0),
                  -- milestone pointer (not sychronized)
        window => window_sequence(MILESTONE_SIZE-1 downto 0),-- PRS Window input
        milestone => milestone(MILESTONE_CLUSTERS*MILESTONE_SIZE-1 downto 0),
                  -- Milestone Cluster Window
        symbol_available => symbol_available,
                  -- new symbol is available from Generator
        clk => clk,                          -- clock
        reset => reset,                      -- Asynchronous Reset
        enable => window_valid,              -- decode the current valid PRS Window
        ready => ready,                      -- flag: ready for decoding,
                  -- the algebra was loaded and the milestone table completed
        milestone_offset => gf_milestone_offset(DATA_SIZE-1 downto 0),
                  -- Milestone Offset
        milestone_pointer => gf_milestone_pointer(POLYNOM_DEGREE_WIDTH-1
                                 downto 0),    -- milestone pointer
(sychronized)
        milestone_cluster => gf_milestone_cluster(MILESTONE_CLUSTER_WIDTH-1
                                 downto 0),    -- milestone cluster
(sychronized)
        match => sequence_matched            -- PRS Window matched Milestone
    );

-------------------------------------------------------------------------
    -- Status register
    status <= overflow & decode & ready & algebra_done;

    -- GF symbol output
    extend_gf_symbol_1b1: for i in GF_WIDTH to DATA_SIZE-1 generate
        gf_symbol(i) <= '0';
    end generate;
    gf_symbol(GF_WIDTH-1 downto 0) <= symbol(GF_WIDTH-1 downto 0);

    -- new symbol is available from Generator
    gf_sync <= symbol_available;

    extend_milestone_pointer_1b1:
    for i in POLYNOM_DEGREE_WIDTH to DATA_SIZE-1 generate
        gf_milestone_pointer(i) <= '0';
    end generate;

    extend_milestone_cluster_1b1:
    for i in MILESTONE_CLUSTER_WIDTH to DATA_SIZE-1 generate
        gf_milestone_cluster(i) <= '0';
    end generate;

end arch;
```

Control Unit

```
-- Coprocessor Control Unit

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

library work;
use work.VENDOR_package.all;
use work.PRDEC_package.all;

-- Signals ending in _l are asserted LOW (0).
-- All the other signals are asserted HIGH (1).

entity Control is          -- Coprocessor Control Unit component
generic (
   DATA_SIZE:           INTEGER := 8;        -- width of the data bus
   MAX_POLYNOM_DEGREE: INTEGER := 16    -- maximum degree of the polynomial
);
port (
   address:  in STD_LOGIC;        -- address bus from µP
   data_in:  in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);-- data bus in from µP
   data_out: out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                                  -- data bus out to the other modules
   wr:       in STD_LOGIC;        -- write enable
   clk:      in STD_LOGIC;        -- clock
   reset:    in STD_LOGIC;        -- reset
   table_row_address: out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                      -- row address for the current table
   table_col_address: out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                      -- column address for the current table
   load_add: out STD_LOGIC;   -- load Addition Table into RAM
   load_mul: out STD_LOGIC;   -- load Multiplication Table into RAM
   load_coef:    out STD_LOGIC;
                      -- load Primitive Polinomial Coefficients Table into RAM
   load_window: out STD_LOGIC;-- load PRS Window into RAM
   polynom_degree: out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                      -- The Degree of the Primitive Polynomial minus 1
   tap: out STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE-1 downto 0);-- Tap Select
   milestone_period: out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
   decode: out STD_LOGIC;     -- status flag: decode the current PRS Window
   ready: out STD_LOGIC;      -- status flag: ready for decoding, the algebra
                              -- was loaded and the milestone table completed
   algebra_done: out STD_LOGIC;  -- status flag: the algebra was loaded
   overflow: out STD_LOGIC;   -- status flag: Milestone Database overflowed
   match: in STD_LOGIC;           -- PRS Window decoded
   milestone_overflow: in STD_LOGIC;-- Milestone Database overflowed
   irq: out STD_LOGIC         -- interrupt request to µP on PRS window decoded
);
end Control;

architecture arch of Control is

   subtype PossibleStates is STD_LOGIC_VECTOR (2 downto 0);
   constant Idle_State:                        PossibleStates := "000";
   constant Load_GF_Size_State:             PossibleStates := "001";
   constant Load_GF_Add_Table_State:        PossibleStates := "010";
   constant Load_GF_Mul_Table_State:        PossibleStates := "011";
   constant Load_GF_Polynom_Degree_State:   PossibleStates := "100";
   constant Load_GF_Coef_Table_State:       PossibleStates := "101";
   constant Load_Milestone_Period_State:    PossibleStates := "110";
```

```
    constant Load_PRS_Window_State:          PossibleStates := "111";
    signal present_state, next_state: PossibleStates;

    signal HIGH, LOW: STD_LOGIC;
    signal delay_data_in, number: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal polynom_degree_s: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal count1, count2: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal Counter_areset, end_Count1, Counter2_enable, end_Count2: STD_LOGIC;
    signal end_Op_Table, end_Coef_Table, end_PRS_Window: STD_LOGIC;
    signal delay_wr, load_tap, load_number: STD_LOGIC;
    signal ready_j, ready_k, ready_s: STD_LOGIC;
    signal algebra_done_j, algebra_done_k, algebra_done_s: STD_LOGIC;
    signal decode_j, decode_k, decode_s: STD_LOGIC;
    signal overflow_j, overflow_k: STD_LOGIC;
    signal irq_d: STD_LOGIC;

begin
    HIGH <= '1';
    LOW  <= '0';

    delay_wr_lbl: D_FF port map (
        d => wr, clk => clk, reset => reset, preset => LOW, q => delay_wr);

-------------------------------------------------------------------------------
    -- Delay 1 clk the data_in_bus in order to synchronize the process
    delay_data_in_lbl: Generic_Register generic map (RegSize => DATA_SIZE)
    port map (    clk => clk, enable => wr, reset, preset => LOW,
        data => data_in, reg => delay_data_in);

-------------------------------------------------------------------------------
    -- Control State Machine
    Make_next_state:
    process (present_state) begin

        case present_state is
            when Idle_State =>
                if wr = '1' then
                    case address is
                        when '0' =>          -- address 0
                            next_state <= Load_GF_Size_State;
                        when '1' =>          -- address 1
                            if ready_s = '1' then   -- the algebra was loaded and
                                                    -- the milestone table completed
                                next_state <= Load_PRS_Window_State;
                            else
                                next_state <= Idle_State;
                            end if;
                        when others => next_state <= Idle_State;
                    end case;
                else
                    next_state <= Idle_State;
                end if;

            ---------------------------------------------------
            when Load_GF_Size_State =>
                if wr = '1' then
                    next_state <= Load_GF_Add_Table_State;
                else
                    next_state <= Load_GF_Size_State;
                end if;

            when Load_GF_Add_Table_State =>
                if (wr = '1' and end_Op_Table = '1') then
```

```
                next_state <= Load_GF_Mul_Table_State;
            else
                next_state <= Load_GF_Add_Table_State;
            end if;

        when Load_GF_Mul_Table_State =>
            if (wr = '1' and end_Op_Table = '1') then
                next_state <= Load_GF_Polynom_Degree_State;
            else
                next_state <= Load_GF_Mul_Table_State;
            end if;

        ------------------------------------------------
        when Load_GF_Polynom_Degree_State =>
            if wr = '1' then
                next_state <= Load_GF_Coef_Table_State;
            else
                next_state <= Load_GF_Polynom_Degree_State;
            end if;

        when Load_GF_Coef_Table_State =>
            if (wr = '1' and end_Coef_Table = '1') then
                next_state <= Load_Milestone_Period_State;
            else
                next_state <= Load_GF_Coef_Table_State;
            end if;

        ------------------------------------------------
        when Load_Milestone_Period_State =>
            if wr = '1' then
                next_state <= Load_PRS_Window_State;
            else
                next_state <= Load_Milestone_Period_State;
            end if;

        ------------------------------------------------
        when Load_PRS_Window_State =>
            if (end_PRS_Window = '1') then
                next_state <= Idle_State;
            else
                next_state <= Load_PRS_Window_State;
            end if;

        ------------------------------------------------
        when others   =>
            next_state <= Idle_State;

    end case;
end process;

Clock_next_state:
process (reset, clk) begin
    if (reset = '1') then
        present_state <= Idle_State;
    elsif (clk'event and clk = '1') then
        present_state <= next_state;
    end if;
end process;

--------------------------------------------------------------------------------
    -- The register number keeps various parameters
    number_lbl: Generic_Register generic map (RegSize => DATA_SIZE)
    port map (clk => clk, enable => load_number, reset, preset => LOW,
```

```
      data => delay_data_in, reg => number);
   load_number <= '1' when
                  delay_wr = '1' and (present_state = Load_GF_Size_State or
                                      -- keeps the Number of Alphabet Symbols
                  present_state = Load_GF_Polynom_Degree_State or
                                      -- keeps the Degree of the Primitive Polynomial
                  present_state = Load_Milestone_Period_State)
                                      -- keeps the Milestone Period
                  else '0';

   -- The folowing two counters are used with the above Control State Machine
   Counter1_lbl: Generic_Binary_Counter generic map (DATA_SIZE)
   port map (clk => clk, enable => wr,
      areset => Counter_areset, sreset => end_Count1, count1);
   Counter_areset <= '1' when (present_state = Idle_State
                        or present_state = Load_GF_Size_State
                        or present_state = Load_GF_Polynom_Degree_State
                        or present_state = Load_Milestone_Period_State)
                        or reset = '1' else '0';
   end_Count1 <= '1' when count1 = number else '0';

   Counter2_lbl: Generic_Binary_Counter generic map (DATA_SIZE)
   port map (clk => clk, enable => Counter2_enable,
      areset => Counter_areset, sreset => end_Op_Table, count2);
   Counter2_enable <= '1' when end_Count1 = '1' and wr = '1' else '0';
   end_Count2 <= '1' when count2 = number else '0';

   end_Op_Table <= '1' when end_Count2 = '1' and end_count1 = '1' else '0';
   end_Coef_Table <= end_Count1;
   end_PRS_Window <= '1' when count1 = polynom_degree_s else '0';

-------------------------------------------------------------------------------
   load_tap <= '1' when present_state = Load_GF_Polynom_Degree_State
                        and delay_wr = '1' else '0';

   polynom_degree_lbl: Generic_Register generic map (RegSize => DATA_SIZE)
   port map (data => delay_data_in, clk => clk, enable => load_tap,
      reset => reset, preset => LOW, reg => polynom_degree_s);
   polynom_degree <= polynom_degree_s;

   -- Tap Decoder component
   Taps_lbl: Taps
   generic map (WIDTH => DATA_SIZE, MAX_POLYNOM_DEGREE => MAX_POLYNOM_DEGREE)
   port map (reset => reset, clk => clk,
      data => delay_data_in, load_tap => load_tap, tap => tap);

-------------------------------------------------------------------------------
   -- Status flags
   algebra_done_lbl: JK_FF port map (clk => clk, reset => reset, preset => LOW,
      j => algebra_done_j, k => algebra_done_k, q => algebra_done_s);
   algebra_done_j <= '1' when present_state = Load_Milestone_Period_State
                     else '0';
   algebra_done_k <= '1' when present_state = Load_GF_Size_State else '0';
   algebra_done <= algebra_done_s;

   ready_lbl: JK_FF port map (clk => clk, reset => reset, preset => LOW,
      j => ready_j, k => ready_k, q => ready_s);
   ready_j <= '1' when algebra_done_s = '1' and match = '1' else '0';
   ready_k <= '1' when present_state = Load_GF_Size_State else '0';
   ready <= ready_s;

   decode_lbl: JK_FF port map (clk => clk, reset => reset, preset => LOW,
      j => decode_j, k => decode_k, q => decode_s);
```

```
    decode_j <= '1' when present_state = Load_PRS_Window_State
                         and end_PRS_Window = '1' else '0';
    decode_k <= '1' when (match = '1' or milestone_overflow = '1') else '0';
    decode <= decode_s;

    overflow_lbl: JK_FF     port map (clk => clk, reset => reset, preset => LOW,
       j => overflow_j, k => overflow_k, q => overflow);
    overflow_j <= '1' when algebra_done_s = '1'
                         and milestone_overflow = '1' else '0';
    overflow_k <= '1' when present_state = Load_GF_Size_State else '0';

-------------------------------------------------------------------------------
    -- Commands to other modules
    load_add <= '1' when present_state = Load_GF_Add_Table_State
                         and delay_wr = '1' else '0';
    load_mul <= '1' when present_state = Load_GF_Mul_Table_State
                         and delay_wr = '1' else '0';
    load_coef <= '1' when present_state = Load_GF_Coef_Table_State
                         and delay_wr = '1' else '0';
    load_window <= '1' when present_state = Load_PRS_Window_State
                         and delay_wr = '1' else '0';

-------------------------------------------------------------------------------
    data_out <= delay_data_in;              -- GF data to other modules
    table_row_address <= count2;            -- GF Operation Table address
    table_col_address <= count1;            -- GF Operation Table address

    milestone_period <= number;

-------------------------------------------------------------------------------
    -- Generate an interrupt on PRS sequence matching during PRS decoding
    irq_lbl: D_FF port map (clk => clk, reset, preset => LOW,
       d => irq_d, q => irq);
    irq_d <= '1' when decode_s = '1' and
                     (match = '1' or milestone_overflow = '1') else '0';

end arch;
```

PRS Generator Unit

```
-- Generic GF Generator

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

library work;
use work.VENDOR_package.all;
use work.PRDEC_package.all;


-- Signals ending in _l are asserted LOW (0).
-- All the other signals are asserted HIGH (1).

entity GF_Generator is       -- Generic GF Generator component
   generic (
      GF_WIDTH:              INTEGER := 2; -- width of GF elements
      DOUBLE_GF_WIDTH:       INTEGER := 4; -- width of entry address for GF
                                           -- operation tables = 2*GF_WIDTH
      MAX_POLYNOM_DEGREE:    INTEGER := 4; -- maximum degree of the primitive
                                           -- polynomial
      POLYNOM_DEGREE_WIDTH: INTEGER := 2  -- width of MAX_POLYNOM_DEGREE equal
                                           -- to LOG2(MAX_POLYNOM_DEGREE)
   );
    port (
      data:             in STD_LOGIC_VECTOR (GF_WIDTH-1 downto 0);
                                -- serial data input
      row_address:      in STD_LOGIC_VECTOR (GF_WIDTH-1 downto 0);
                                -- row address input
      col_address:      in STD_LOGIC_VECTOR (GF_WIDTH-1 downto 0);
                                -- column address input
      clk:              in STD_LOGIC; -- Clock
      polynom_degree: in STD_LOGIC_VECTOR (POLYNOM_DEGREE_WIDTH-1 downto 0);
                        -- The Degree of the Primitive Polynomial minus 1
      tap:              in STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE-1 downto 0);
                                -- Tap Select
      load_mul:         in STD_LOGIC; -- Load Multiply Table
      load_add:         in STD_LOGIC; -- Load Addition Table
      load_coef:        in STD_LOGIC; -- Load Polynomial Coefficients
      load_window:      in STD_LOGIC; -- Load Window Sequence
      enable:           in STD_LOGIC; -- Enable Operation
      reset:            in STD_LOGIC; -- Asynchronous Reset
      symbol:           out STD_LOGIC_VECTOR (GF_WIDTH-1 downto 0);
                                -- output symbol
      available:        out STD_LOGIC;-- new symbol is available from Generator
      valid:            out STD_LOGIC;-- PRS window valid
      match:            in STD_LOGIC; -- Matched output symbol
      window:     out STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE*GF_WIDTH-1 downto 0)
                                -- output sequence window
   );
end GF_Generator;

architecture arch of GF_Generator is

   type PossibleStates is (Idle_State, New_Symbol_State, Shift_State,
                           Delay_Mul_State, Delay_Add_State);
   signal present_state, next_state: PossibleStates;

   signal HIGH, LOW: STD_LOGIC;
   signal GF_SR_enable, GF_Coef_enable, coef_fifo_stop: STD_LOGIC;
```

```
    signal GF_mul_q, GF_add_q, GF_Coef_q, GF_coef,
           cellDelay, GF_SR_serial_in,
           GF_SR_serial_out: STD_LOGIC_VECTOR (GF_WIDTH-1 downto 0);
    signal GF_SR_parallel_out,
           window_s: STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE*GF_WIDTH-1 downto 0);
    signal GF_add_address,
           GF_mul_address: STD_LOGIC_VECTOR (2*GF_WIDTH-1 downto 0);
    signal shiftcount: STD_LOGIC_VECTOR (POLYNOM_DEGREE_WIDTH-1 downto 0);
    signal endShift_State, available_d: STD_LOGIC;

begin
    HIGH <= '1';
    LOW  <= '0';

-------------------------------------------------------------------------------
    GF_ShiftRegister_lbl: GF_ShiftRegister
        generic map (
            GF_WIDTH => GF_WIDTH,
            MAX_POLYNOM_DEGREE => MAX_POLYNOM_DEGREE
        )
        port map (
            serial_in => GF_SR_serial_in,
            clk => clk,
            tap => tap,
            enable => GF_SR_enable,
            reset => reset,
            serial_out => GF_SR_serial_out,
            parallel_out => GF_SR_parallel_out
        );
    GF_SR_serial_in <= data when load_window = '1'
                       else GF_add_q when present_state = Delay_Add_State
                       else cellDelay;
    GF_SR_enable <= '1' when load_window = '1' or present_state /= Idle_State
                       else '0';

    CellDelay_lbl: Generic_Register
        generic map (GF_WIDTH)
        port map (
            data => GF_SR_serial_out,
            clk,
            enable => HIGH,
            reset,
            preset => LOW,
            reg => cellDelay
        );


    -------------------------------------------------------------------------------
    GF_Mul_lbl: ram
        generic map (
            ADDRESS_SIZE => DOUBLE_GF_WIDTH,
            DATA_SIZE => GF_WIDTH
        )
        port map (
            address => GF_mul_address,
            data => data,
            we => load_mul,
            inclock => clk,
            q => GF_mul_q
        );
    GF_mul_address <= (row_address & col_address) when load_mul = '1'
                      else (GF_coef & GF_SR_serial_out);
```

```
    GF_Add_lbl: ram
        generic map (
            ADDRESS_SIZE => DOUBLE_GF_WIDTH,
            DATA_SIZE => GF_WIDTH
        )
        port map (
            address => GF_add_address,
            data => data,
            we => load_add,
            inclock => clk,
            q => GF_add_q
        );
    GF_add_address <= (row_address & col_address) when load_add = '1'
              else (others => '0') when present_state = New_Symbol_State
              else (GF_mul_q & GF_add_q);

    GF_Coef_lbl: Tap_FIFO
        generic map (
            WIDTH => GF_WIDTH,
            NUMWORDS => MAX_POLYNOM_DEGREE
        )
        port map (
            data => data,
            tap => tap(MAX_POLYNOM_DEGREE-1 downto 0),
            wrreq => load_coef,
            rdreq => GF_Coef_enable,
            clock => clk,
            reset => reset,
            q => GF_Coef_q
        );
    GF_Coef_enable <= '1' when (present_state = New_Symbol_State
                                or present_state = Shift_State) else '0';
    GF_coef <= GF_Coef_q when GF_Coef_enable = '1' else (others => '0');

-------------------------------------------------------------------------------
    -- GF(x) Generator State Machine
    Make_next_state:
    process (present_state) begin

        case present_state is
            when Idle_State =>
                if (enable = '1') then
                    next_state <= New_Symbol_State;
                else
                    next_state <= Idle_State;
                end if;

            when New_Symbol_State =>
                next_state <= Shift_State;

            when Shift_State =>
                if (coef_fifo_stop = '1') then
                    next_state <= Delay_Mul_State;
                else
                    next_state <= Shift_State;
                end if;

            when Delay_Mul_State =>
                next_state <= Delay_Add_State;

            when Delay_Add_State =>
                next_state <= New_Symbol_State;
```

```
            when others   =>
                next_state <= Idle_State;
        end case;
    end process;

    Clock_next_state:
    process (reset, clk) begin
        if (reset = '1') then
            present_state <= Idle_State;
        elsif (clk'event and clk = '1') then
            if (match = '1') then
                present_state <= Idle_State;
            else
                present_state <= next_state;
            end if;
        end if;
    end process;

-------------------------------------------------------------------------------
    ShiftCounter_lbl: Generic_Binary_Counter
        generic map (POLYNOM_DEGREE_WIDTH)
        port map (
            clk,
            enable => HIGH,
            areset => reset,
            sreset => endShift_State,
            count => shiftcount
        );
    coef_fifo_stop <= '1' when shiftcount = polynom_degree else '0';

    endShift_State <= '1' when present_state = Delay_Add_State
                               or present_state = Idle_State else '0';

-------------------------------------------------------------------------------
    -- GF Generator outputs

    -- output sequence window
    window_lbl: for i in MAX_POLYNOM_DEGREE to 1 generate
        symbol_i_lbl: Generic_Register
            generic map (GF_WIDTH)
            port map (
                clk => clk,
                enable => available_d,
                reset => reset,
                preset => LOW,
                data => GF_SR_parallel_out(i*GF_WIDTH-1 downto (i-1)*GF_WIDTH),
                reg => window_s(i*GF_WIDTH-1 downto (i-1)*GF_WIDTH)
            );
    end generate;
    window <= window_s;

    -- serial symbol output
    symbol <= window_s(GF_WIDTH-1 downto 0);

    -- New symbol available from Generator
    available_lbl: D_FF
        port map (
            d => available_d,
            clk,
            reset,
            preset => LOW,
            q => available);
    available_d <= '1' when present_state = New_Symbol_State else '0';
```

```
    -- validate the PRS window by waiting for the first symbol_available pulse
    valid_lbl: process(clk, reset) begin
       if reset = '1' then
          valid <= '0';
       elsif clk'event and clk = '1' then
          if enable = '0' then
             valid <= '0';
          elsif available_d = '1' then
             valid <= '1';
          end if;
       end if;
    end process;

end arch;
```

Milestone Database Unit

```
-- Generic GF Milestone Storage Memory

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

library work;
use work.VENDOR_package.all;
use work.VHDL_package.all;
use work.PRDEC_package.all;

-- Signals ending in _l are asserted LOW (0).
-- All the other signals are asserted HIGH (1).

entity GF_PRS_Milestones is                    -- Generic GF Milestone Storage
Memory component
    generic (
        DATA_SIZE: INTEGER := 8;               -- the data bus width
        MAX_POLYNOM_DEGREE: INTEGER := 4;      -- maximum degree of the primitive
                                               -- polynomial
        POLYNOM_DEGREE_WIDTH: INTEGER := 3;
                    -- width of MAX_POLYNOM_DEGREE = LOG2(MAX_POLYNOM_DEGREE)
        MILESTONE_SIZE: INTEGER := 8;
                    -- the size (in binary bits) of a milestone word saved in
                    -- FIFO and it is equal to MAX_POLYNOM_DEGREE * GF_WIDTH
        MAX_NUMWORDS: INTEGER := 6;
                    -- maximum number of milestones saved in FIFO and it is
                    -- equal to MAX_POLYNOM_DEGREE + 2 (latency)
        MILESTONE_CLUSTERS: INTEGER := 4;      -- number of milestone clusters
        MILESTONE_CLUSTER_WIDTH: INTEGER := 2
                    -- milestone cluster width equal to LOG2MILESTONE_CLUSTERS)
    );
     port (
        window:           in STD_LOGIC_VECTOR (MILESTONE_SIZE-1 downto 0);
                    -- input sequence window
        clk:              in STD_LOGIC;        -- clock
        polynom_degree: in STD_LOGIC_VECTOR (POLYNOM_DEGREE_WIDTH-1 downto 0);
                    -- The Degree of the Primitive Polynomial minus 1
        period:           in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                    -- GF milestones period minus 1
        tap:              in STD_LOGIC_VECTOR (MAX_POLYNOM_DEGREE-1 downto 0);
                    -- tap selector
        available:        in STD_LOGIC;     -- new sequence window is available
        ready:            in STD_LOGIC;
                    -- flag: ready for decoding, the algebra was loaded and the
                    -- milestone table completed
        decode:           in STD_LOGIC;     -- flag: decode the current PRS Window
        reset:            in STD_LOGIC;     -- reset milestones
        milestone:        out STD_LOGIC_VECTOR(MILESTONE_CLUSTERS*MILESTONE_SIZE-
                          1 downto 0);      -- output milestones window
        milestone_count:out STD_LOGIC_VECTOR (POLYNOM_DEGREE_WIDTH-1 downto 0);
                    -- milestone pointer
        milestone_overflow:out STD_LOGIC
    );
end GF_PRS_Milestones;

architecture arch of GF_PRS_Milestones is

   signal HIGH, LOW: STD_LOGIC;
   signal GF_PRS_Milestones_wrreq, next_milestone, save_milestone: STD_LOGIC;
```

```
    signal counter_enable, pointer_enable, pointer_sreset: STD_LOGIC;
    signal count: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal pointer: STD_LOGIC_VECTOR (POLYNOM_DEGREE_WIDTH-1 downto 0);
    signal available_d, polynom_length, delay_polynom_length: STD_LOGIC;
    signal milestone_tap: STD_LOGIC_VECTOR (MAX_NUMWORDS-1 downto 0);
    signal store: STD_LOGIC;       -- store milestones command
    signal retrieve: STD_LOGIC;    -- retrieve milestones command
    signal cluster: STD_LOGIC_VECTOR (MILESTONE_CLUSTER_WIDTH-1 downto 0);
    signal load_cluster: STD_LOGIC_VECTOR (MILESTONE_CLUSTERS-1 downto 0);
    signal wrreq, rdreq: STD_LOGIC_VECTOR (MILESTONE_CLUSTERS-1 downto 0);
    signal cluster_full_d, cluster_full, cluster_enable: STD_LOGIC;

begin
    HIGH <= '1';
    LOW  <= '0';

    store <= '1' when decode = '1' and ready = '0' else '0';
    retrieve <= '1' when decode = '1' and ready = '1' else '0';

-------------------------------------------------------------------------
    Milestone_Clusters_lbl: for i in 0 to MILESTONE_CLUSTERS-1 generate
        GF_PRS_Milestones_lbl: Tap_FIFO
            generic map (WIDTH => MILESTONE_SIZE, NUMWORDS => MAX_NUMWORDS)
            port map (
                data => window,
                tap => milestone_tap,
                wrreq => wrreq(i),
                rdreq => rdreq(i),
                clock => clk,
                reset => reset,
                q => milestone((i+1)*MILESTONE_SIZE-1 downto i*MILESTONE_SIZE)
            );
        wrreq(i) <= '1' when GF_PRS_Milestones_wrreq = '1'
                         and load_cluster(i) = '1' else '0';
        rdreq(i) <= '1' when (GF_PRS_Milestones_wrreq = '1'
                             and load_cluster(i) = '0')
                             -- just roll the FIFO during initialization of
                             -- the other clusters
                            or retrieve = '1' else '0';    -- retrieve data from FIFO
    end generate;

    GF_PRS_Milestones_wrreq <= '1' when store = '1' and save_milestone = '1'
                                        and available = '1' else '0';
    milestone_tap <= tap & "00";      -- add two more due to pipeline latency

-------------------------------------------------------------------------
    -- Counts the Milestone Period
    Counter_lbl: Generic_Binary_Counter
        generic map (DATA_SIZE)
        port map (
            clk => clk,
            enable => counter_enable,
            areset => reset,
            sreset => next_milestone,
            count => count
        );
    counter_enable <= '1' when available = '1' or store = '0' else '0';

    -- Generate the save_milestone pulse each Milestone Period
    next_milestone <= '1' when count = period or store = '0' else '0';
    save_milestone <= '1' when count = CONV_STD_LOGIC_VECTOR(0, DATA_SIZE)
                      else '0';
```

```
-------------------------------------------------------------------------------
    -- Milestone Pointer: counts up to polynom_degree + 2
    -- (2 = latency added by the op. cells)
    pointer_lbl: Generic_Binary_Counter
        generic map (POLYNOM_DEGREE_WIDTH)
        port map (
            clk => clk,
            enable => pointer_enable,
            areset => reset,
            sreset => pointer_sreset,
            count => pointer
        );
    pointer_enable <= '1' when GF_PRS_Milestones_wrreq = '1'
                          or retrieve = '1' else '0';

    polynom_length <= '1' when pointer = polynom_degree and
              not (polynom_degree=CONV_STD_LOGIC_VECTOR(0,POLYNOM_DEGREE_WIDTH))
              else '0';
    delay_polynom_length_lbl: D_FFE
        port map (
            d => polynom_length,
            ena => pointer_enable,
            clk => clk,
            reset, LOW,
            q => delay_polynom_length
        );
    pointer_sreset_lbl: D_FFE
        port map (
            d => delay_polynom_length,
            ena => pointer_enable,
            clk => clk,
            reset, LOW,
            q => pointer_sreset
        );

    milestone_count(POLYNOM_DEGREE_WIDTH-1 downto 0) <=
                                pointer(POLYNOM_DEGREE_WIDTH-1 downto 0);

-------------------------------------------------------------------------------
    -- Milestone Cluster
    cluster_lbl: Generic_Binary_Counter
        generic map (MILESTONE_CLUSTER_WIDTH)
        port map (
            clk, enable => cluster_enable,
            areset => reset, sreset => ready,
            count => cluster
        );
    cluster_enable <= '1' when (pointer_sreset = '1' and pointer_enable = '1')
                            or ready = '1' else '0';

    cluster_full_lbl: D_FFE
        port map (
            d => cluster_full_d,
            clk, ena => pointer_enable,
            reset => LOW, preset => LOW,
            q => cluster_full
        );
    cluster_full_d <= '1' when pointer_sreset = '1' and
                            cluster = CONV_STD_LOGIC_VECTOR(MILESTONE_CLUSTERS-1,
                                        MILESTONE_CLUSTER_WIDTH)
                            else '0';

    milestone_overflow_lbl: D_FFE
```

```
        port map (
            d => cluster_full,
            clk, ena => pointer_enable,
            reset => LOW, preset => LOW,
            q => milestone_overflow
        );

    load_cluster_lbl: for i in 0 to MILESTONE_CLUSTERS-1 generate
        load_cluster(i) <= '1' when cluster = CONV_STD_LOGIC_VECTOR(i,
                                              MILESTONE_CLUSTER_WIDTH) else '0';
    end generate;

end arch;
```

Decoder Unit

```
-- GF PRS Decoder Unit

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

library work;
use work.VENDOR_package.all;
use work.VHDL_package.all;
use work.PRDEC_package.all;

-- Signals ending in _l are asserted LOW (0).
-- All the other signals are asserted HIGH (1).

entity GF_PRS_Decoder is                        -- GF PRS Decoder Unit component
   generic (
      DATA_SIZE: INTEGER := 8;                  -- width of the data bus
      POLYNOM_DEGREE_WIDTH: INTEGER := 3;
                -- width of MAX_POLYNOM_DEGREE = LOG2(MAX_POLYNOM_DEGREE) + 1
                -- rounded up to the near integer
      MILESTONE_SIZE: INTEGER := 8;
                -- the size (in binary bits) of a milestone word saved in FIFO
                -- and it is equal to MAX_POLYNOM_DEGREE * GF_WIDTH
      MILESTONE_CLUSTERS: INTEGER := 4;     -- number of milestone clusters
      MILESTONE_CLUSTER_WIDTH: INTEGER := 2
                -- milestone cluster width equal to LOG2(MILESTONE_CLUSTERS)
   );
    port (
      milestone_count: in STD_LOGIC_VECTOR (POLYNOM_DEGREE_WIDTH-1 downto 0);
                -- milestone pointer (not synchronized)
      window:          in STD_LOGIC_VECTOR (MILESTONE_SIZE-1 downto 0);
                -- PRS Window input
      milestone:       in STD_LOGIC_VECTOR (MILESTONE_CLUSTERS*MILESTONE_SIZE-
                          1 downto 0); -- Milestones Window
      symbol_available:in STD_LOGIC;-- new symbol is available from Generator
      clk:             in STD_LOGIC;        -- Clock
      reset:           in STD_LOGIC;        -- Asynchronous Reset
      enable:          in STD_LOGIC;        -- decoding enable
      ready:           in STD_LOGIC;
                -- flag: ready for decoding, the algebra was loaded and the
                -- milestone table completed
      milestone_offset:   out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                -- Milestone Offset
      milestone_pointer:out STD_LOGIC_VECTOR (POLYNOM_DEGREE_WIDTH-
                          1 downto 0);-- milestone pointer (synchronized)
      milestone_cluster:  out STD_LOGIC_VECTOR (MILESTONE_CLUSTER_WIDTH-
                          1 downto 0);     -- milestone cluster (synchronized)
       match:             out STD_LOGIC    -- PRS Window matched Milestone
   );
end GF_PRS_Decoder;

architecture arch of GF_PRS_Decoder is

   signal HIGH, LOW: STD_LOGIC;
   signal window_seed: STD_LOGIC_VECTOR(MILESTONE_SIZE-1 downto 0);
   signal ilestone_mux: STD_LOGIC_VECTOR(MILESTONE_SIZE-1 downto 0);
   signal match_d, match_q, shift_count_areset: STD_LOGIC;
   signal shift_count_enable, symbol_1, symbol_2, window_seed_ena: STD_LOGIC;
   signal match_s: STD_LOGIC_VECTOR (MILESTONE_CLUSTERS-1 downto 0);
   signal match_or: STD_LOGIC_VECTOR (MILESTONE_CLUSTERS downto 0);
```

```
    signal shift_count: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal encode_cluster: STD_LOGIC_VECTOR(MILESTONE_CLUSTER_WIDTH-1 downto 0);

begin
    HIGH <= '1';
    LOW  <= '0';

    -- first symbol_available pulse
    symbol_1_lbl: JK_FF
        port map (
            j => symbol_available, k => match_d,
            clk => clk, reset => reset, preset => LOW,
            q => symbol_1
        );

    -- second symbol_available pulse
    symbol_2_lbl: JK_FFE
        port map (
            j => symbol_1, k => match_d,
            clk => clk, ena => symbol_available,
            reset => reset, preset => LOW,
            q => symbol_2
        );

-------------------------------------------------------------------------------
    -- Save the PRS window seed on first symbol_available pulse
    window_seed_lbl: Generic_Register
        generic map (MILESTONE_SIZE)
        port map (
            data => window(MILESTONE_SIZE-1 downto 0),
            clk, enable => window_seed_ena,
            reset, preset => LOW,
            reg => window_seed(MILESTONE_SIZE-1 downto 0)
        );
    window_seed_ena <= '1' when symbol_1 = '0' and symbol_available = '1' else
'0';

-------------------------------------------------------------------------------
    -- If milestone database is not build yet (ready = 0) the current
    -- PRS window is compared with the PRS window seed
    milestone_mux <= window_seed when (ready = '0' and symbol_2 = '1')
                     else milestone(MILESTONE_SIZE-1 downto 0);

-------------------------------------------------------------------------------
    match_s(0) <= '1' when window = milestone_mux and enable = '1' else '0';

    match_s_lbl: for i in 1 to MILESTONE_CLUSTERS-1 generate
        match_s(i) <= '1' when window = milestone((i+1)*MILESTONE_SIZE-
                                                  1 downto i*MILESTONE_SIZE)
                              and enable = '1' else '0';
    end generate;

    match_or(0) <= '0';
    match_or_lbl: for i in 1 to MILESTONE_CLUSTERS generate
        match_or(i) <= match_or(i-1) or match_s(i-1);
    end generate;
    match_d <= match_or(MILESTONE_CLUSTERS);

    match_lbl: D_FF
        port map (
            d => match_d,
            clk => clk, reset => reset, preset => LOW,
            q => match_q
```

```
        );
    match <= match_q;

-------------------------------------------------------------------------------
    shift_counter_lbl: Generic_Binary_Counter
        generic map (DATA_SIZE)
        port map (
            clk, enable => shift_count_enable,
            areset => shift_count_areset, sreset => LOW,
            count => shift_count
        );
    shift_count_areset <= not enable;
    -- skip first symbol_available pulse from counting
    shift_count_enable <= '1' when symbol_1 = '1'
                                and symbol_available = '1' else '0';

    milestone_offset_lbl: Generic_Register
        generic map (RegSize => DATA_SIZE)
        port map (
            data => shift_count,
            clk => clk, enable => match_q, reset, preset => LOW,
            reg => milestone_offset
        );

    milestone_pointer_lbl: Generic_Register
        generic map (RegSize => POLYNOM_DEGREE_WIDTH)
        port map (
            data => milestone_count,
            clk => clk, enable => match_d, reset, preset => LOW,
            reg => milestone_pointer
        );

    encode_cluster_lbl: Generic_Encoder         -- Generic Encoder component
        generic map (
            IN_SIZE => MILESTONE_CLUSTERS,      -- number of inputs
            OUT_WIDTH => MILESTONE_CLUSTER_WIDTH -- width of the encoded output
        )
        port map (
            a => match_s(MILESTONE_CLUSTERS-1 downto 0),
            y => encode_cluster(MILESTONE_CLUSTER_WIDTH-1 downto 0)
        );

    milestone_cluster_lbl: Generic_Register
        generic map (RegSize => MILESTONE_CLUSTER_WIDTH)
        port map (
            data => encode_cluster,
            clk => clk, enable => match_d, reset, preset => LOW,
            reg => milestone_cluster(MILESTONE_CLUSTER_WIDTH-1 downto 0)
        );

end arch;
```

A-3. Interface Library

```
library ieee;
use ieee.std_logic_1164.all;

package Interface_package is

component Generic_Interface          -- Generic Memory Interface component
generic (
    ADDRESS_SIZE:    INTEGER := 2;       -- width of the address bus
    DATA_SIZE:          INTEGER := 8     -- width of the data bus
);
port (
    -- µP Interface
    address_bus:     in STD_LOGIC_VECTOR (ADDRESS_SIZE-1 downto 0);
    data_bus_in:     in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);-- data from µP
    data_bus_out:    out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);-- data to µP
    wr_l, rd_l, cs_l:in STD_LOGIC;
    irq_l:           out STD_LOGIC;
    reset, clk:      in STD_LOGIC;
    oe:              out STD_LOGIC;
    -- PRDEC Core Interface
    data_out: out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);-- data to PRDEC core
    write:    out STD_LOGIC;   -- write to PRDEC core pulse
    irq_ch0:  in STD_LOGIC;    -- interrupt request to µP on PRS window decoded
    status_ch0:in STD_LOGIC_VECTOR (3 downto 0);             -- Status Register
    milestone_offset:   in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    milestone_pointer:  in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0)
    milestone_cluster:  in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0)
);
end component;

end Interface_package;
```

Interface Library Components

```
-- Generic Memory Interface Unit

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

library work;
use work.VENDOR_package.all;
use work.VHDL_package.all;

-- Signals ending in _l are asserted LOW (0).
-- All the other signals are asserted HIGH (1).

entity Generic_Interface is          -- Generic Memory Interface component
generic (
    ADDRESS_SIZE:    INTEGER := 2;     -- width of the address bus
    DATA_SIZE:       INTEGER := 8      -- width of the data bus
);
port (
    -- µP Interface
    address_bus:     in STD_LOGIC_VECTOR (ADDRESS_SIZE-1 downto 0);
    data_bus_in:     in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0); -- data from µP
    data_bus_out:    out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);-- data to µP
```

```
    wr_l, rd_l, cs_l:in STD_LOGIC;
    irq_l:           out STD_LOGIC;
    reset, clk:      in STD_LOGIC;
    oe:              out STD_LOGIC;
    -- PRDEC Core Interface
     data_out: out STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);-- data to PRDEC core
     write:    out STD_LOGIC;   -- write to PRDEC core pulse
     irq_ch0:  in STD_LOGIC;    -- interrupt request to µP on PRS window decoded
    status_ch0:in STD_LOGIC_VECTOR (3 downto 0);      -- Status Register
    milestone_offset:in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    milestone_pointer:  in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    milestone_cluster:  in STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0)
);
end Generic_Interface;

architecture arch of Generic_Interface is

    type PossibleStates is (First_Read, Second_Read, Third_Read);
    signal present_state, next_state: PossibleStates;
    signal HIGH, LOW: STD_LOGIC;
    signal wr, rd, cs: STD_LOGIC;
    signal status_reg: STD_LOGIC_VECTOR (3 downto 0); -- Status Register
    signal interrupt_reg: STD_LOGIC_VECTOR (3 downto 0);-- Interrupts Register
    signal buff_wr, wr_d, rd_d, buff_rd: STD_LOGIC;
    signal sel, sel_reset, buff_addr, addr1_rd, addr1_rd_fall: STD_LOGIC;

begin
    HIGH <= '1';
    LOW  <= '0';
    wr <= not wr_l;
    rd <= not rd_l;
    cs <= not cs_l;

-------------------------------------------------------------------------------
    -- PRDEC Core Interface

    -- On the end of the write command generate write to PRDEC core pulse
    write_lbl: Pulse port map (
       d => buff_wr, clk => clk, reset => reset, preset => LOW, fall => write);
    buff_wr_lbl: D_FF port map (
       d => wr_d, clk => clk, reset => reset, preset => LOW, q => buff_wr);
    wr_d <= '1' when wr = '1' and cs = '1' else '0';

    -- data from µP to PRDEC Core
    data_out_lbl: Generic_Register generic map (RegSize => DATA_SIZE)
    port map (clk => wr, enable => cs, reset, preset => LOW,
       data => data_bus_in, reg => data_out);

-------------------------------------------------------------------------------
    -- µP Interface
    irq_l <= not irq_ch0;

    interrupt_reg <= "000" & irq_ch0;
    status_reg <= status_ch0;            -- Channel 0 Status Register

-------------------------------------------------------------------------------
   Make_next_state:
   process (present_state) begin

      case present_state is
         when First_Read =>
            if buff_addr = '1' then
               next_state <= Second_Read;
```

```
            else
               next_state <= First_Read;
            end if;
         when Second_Read =>
            if buff_addr = '1' then
               next_state <= Third_Read;
            else
               next_state <= Second_Read;
            end if;
         when Third_Read =>
            if buff_addr = '1' then
               next_state <= First_Read;
            else
               next_state <= Third_Read;
            end if;
         when others   =>
            next_state <= First_Read;
      end case;
   end process;

   Clock_next_state:
   process (reset, clk) begin
      if (reset = '1') then
         present_state <= First_Read;
      elsif (clk'event and clk = '1') then
         if buff_rd = '1' and buff_addr = '0' then   -- synchronous reset
            present_state <= First_Read;
         elsif addr1_rd_fall = '1' then              -- enable
            present_state <= next_state;
         end if;
      end if;
   end process;

-------------------------------------------------------------------------------
   buff_addr_lbl: D_FF port map (clk => clk, reset => reset, preset => LOW,
      d => address_bus(0), q => buff_addr);

   -- Detect the end of the read command and use it to
   -- toggle the selector signal of the output mux
   addr1_rd_fall_lbl: Pulse port map (reset => reset, preset => LOW,
      clk => clk, d => addr1_rd, fall => addr1_rd_fall);
   addr1_rd <= '1' when buff_rd = '1' and buff_addr = '1' else '0';

   buff_rd_lbl: D_FF port map (
      d => rd_d, clk => clk, reset => reset, preset => LOW, q => buff_rd);
   rd_d <= '1' when rd_l = '0' and cs_l = '0' else '0';

   -- Data Out Multiplexer
   data_bus_out <=
      milestone_cluster when                    -- read Milestone Cluster
         (address_bus(0) = '1' and present_state = First_Read)
      else milestone_pointer when               -- read Milestone Pointer
         (address_bus(0) = '1' and present_state = Second_Read)
      else milestone_offset when                -- read Milestone Offset
         (address_bus(0) = '1' and present_state = Third_Read)
      else (interrupt_reg & status_reg) when    -- read Interrupts and
         address_bus(0) = '0'                   -- Status Register
      else (others => '-');

-------------------------------------------------------------------------------
   oe <= rd_d;      -- tri state bus output enable

end arch;
```

A-4. Microprocessor Library

```
library ieee;
use ieee.std_logic_1164.all;

package MICROPROCESSOR_package is

component Microprocessor       -- Generic Microprocessor component
generic (ADDRESS_SIZE: INTEGER := 2; DATA_SIZE: INTEGER := 8);
port (
    data_bus:             inout STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    address_bus:          out STD_LOGIC_VECTOR (ADDRESS_SIZE-1 downto 0);
    wr_l, rd_l, cs_l:     out STD_LOGIC;
    irq_l:                in STD_LOGIC;
    reset, clk:           in STD_LOGIC;
    ------------------------------------------------------------------------
    -- Testing Interface (not implemented physically)
    -- used for simulation purposes only
    run:                  in STD_LOGIC                    -- start process
);
end component;

component M68HC11              -- M68HC11 Microprocessor Emulation component
generic (ADDRESS_SIZE: INTEGER := 2; DATA_SIZE: INTEGER := 8);
port (
    address_data_bus:     inout STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                                -- address/data multiplexed
    as:                   out STD_LOGIC;       -- address strobe
    rd_wr_l:              out STD_LOGIC;       -- read/write select
    e:                    out STD_LOGIC;       -- output clock equal to extal/4
    irq_l:                in STD_LOGIC;        -- interrupt request
    reset_l:              in STD_LOGIC;        -- reset
    extal:                in STD_LOGIC;        -- external input clock
    ------------------------------------------------------------------------
    -- Testing Interface (not implemented physically)
    -- used for simulation purposes only
    run:                  in STD_LOGIC                    -- start process
);
end component;

end MICROPROCESSOR_package;
```

Microprocessor Library Components

```
-- Generic Microprocessor component

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

library work;
use work.VENDOR_package.all;

-- Signals ending in _l are asserted LOW (0).
-- All the other signals are asserted HIGH (1).

entity Microprocessor is          -- Generic Microprocessor component
   generic (ADDRESS_SIZE: INTEGER := 2; DATA_SIZE: INTEGER := 8);
    port (
      data_bus:           inout STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
      address_bus:        out STD_LOGIC_VECTOR (ADDRESS_SIZE-1 downto 0);
```

```
        wr_1, rd_1, cs_1:    out STD_LOGIC;
        irq_1:               in STD_LOGIC;
        reset, clk:          in STD_LOGIC;
        ------------------------------------------------------------------------
        -- Testing Interface (not implemented physically)
        -- used for simulation purposes only
        run:                 in STD_LOGIC          -- start process
    );
end Microprocessor;

architecture arch of Microprocessor is

    constant ROM_ADDRESS_SIZE: INTEGER := 4;
    constant GF_OP_TABLE_SIZE: INTEGER := ROM_ADDRESS_SIZE*ROM_ADDRESS_SIZE;
    constant POLYNOM_DEGREE:   INTEGER := 2; -- Polynom Degree - 1
    constant COEF_TABLE_DEPTH: INTEGER := 3;

    constant MUL_TABLE_DATA_FILE: STRING := "rom_GF4_Mul.mif";
    constant ADD_TABLE_DATA_FILE: STRING := "rom_GF4_Add.mif";
    constant COEF_TABLE_DATA_FILE:STRING := "rom_GF4_Coef3.mif";
    constant WINDOW_DATA_FILE:    STRING := "rom_GF4_PRS3.mif";

    type PossibleStates is (Reset_State, Write_GF_Size_State,
                    Write_GF_Mul_Table_State, Write_GF_Add_Table_State,
                    Write_GF_Polynom_Degree_State, Write_GF_Coef_Table_State,
                    Idle_State, Write_Window_Data_State);
    signal present_state, next_state: PossibleStates;

    signal HIGH, LOW: STD_LOGIC;
    signal wr_d, wr, rd_d, rd, cs_d, cs: STD_LOGIC;
    signal Mul_Table, Add_Table: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal Coef_Table, Window_Table: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal data_mux_d, data_mux: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
     signal address_bus_d: STD_LOGIC_VECTOR (ADDRESS_SIZE-1 downto 0);
    signal count, PRS_count: STD_LOGIC_VECTOR (ROM_ADDRESS_SIZE-1 downto 0);
    signal Counter_sreset, Counter_enable: STD_LOGIC;
    signal PRS_Counter_sreset, PRS_Counter_enable: STD_LOGIC;
    signal end_GF_Mul_Table, end_GF_Add_Table: STD_LOGIC;
    signal end_GF_Coef_Table, end_GF_Window_Data: STD_LOGIC;

begin
    HIGH <= '1';
    LOW  <= '0';

------------------------------------------------------------------------
    GF_Mul_Table_lbl: rom
    generic map (
        ROM_ADDRESS_SIZE => ROM_ADDRESS_SIZE,
        DATA_SIZE => DATA_SIZE,
        DATA_FILE => MUL_TABLE_DATA_FILE)
    port map (address => count, q => Mul_Table);

    GF_Add_Table_lbl: rom
    generic map (
        ROM_ADDRESS_SIZE => ROM_ADDRESS_SIZE,
        DATA_SIZE => DATA_SIZE,
        DATA_FILE => ADD_TABLE_DATA_FILE)
    port map (address => count, q => Add_Table);

    GF_Coef_Table_lbl: rom
    generic map (
        ROM_ADDRESS_SIZE => ROM_ADDRESS_SIZE,
        DATA_SIZE => DATA_SIZE,
```

```
    DATA_FILE => COEF_TABLE_DATA_FILE)
  port map (address => count, q => Coef_Table);

  GF_PRS_lbl: rom
  generic map (
    ROM_ADDRESS_SIZE => ROM_ADDRESS_SIZE,
    DATA_SIZE => DATA_SIZE,
    DATA_FILE => WINDOW_DATA_FILE)
  port map (address => count, q => Window_Table);

-------------------------------------------------------------------------------
  -- Microprocessor State Machine
  Make_next_state:
  process (present_state) begin

    case present_state is
      when Reset_State =>
        next_state <= Write_GF_Size_State;

      when Write_GF_Size_State =>
        next_state <= Write_GF_Mul_Table_State;

      when Write_GF_Mul_Table_State =>
        if (end_GF_Mul_Table = '1') then
          next_state <= Write_GF_Add_Table_State;
        else
          next_state <= Write_GF_Mul_Table_State;
        end if;

      when Write_GF_Add_Table_State =>
        if (end_GF_Add_Table = '1') then
          next_state <= Write_GF_Polynom_Degree_State;
        else
          next_state <= Write_GF_Add_Table_State;
        end if;

      when Write_GF_Polynom_Degree_State =>
        next_state <= Write_GF_Coef_Table_State;

      when Write_GF_Coef_Table_State =>
        if (end_GF_Coef_Table = '1') then
          next_state <= Idle_State;
        else
          next_state <= Write_GF_Coef_Table_State;
        end if;

      when Idle_State =>
        if (run = '1') then
          next_state <= Write_Window_Data_State;
        else
          next_state <= Idle_State;
        end if;

      when Write_Window_Data_State =>
        if (end_GF_Window_Data = '1') then
          next_state <= Idle_State;
        else
          next_state <= Write_Window_Data_State;
        end if;

      when others  =>
        next_state <= Idle_State;
```

```
        end case;
    end process;

    Clock_next_state:
    process (reset, clk) begin
        if (reset = '1') then
            present_state <= Reset_State;
        elsif (clk'event and clk = '1') then
            present_state <= next_state;
        end if;
    end process;

-------------------------------------------------------------------------------
    Counter_lbl: Generic_Binary_Counter
    generic map (ROM_ADDRESS_SIZE)
    port map (clk => clk, areset => reset, sreset => Counter_sreset,
        enable => Counter_enable, count  => count);
    Counter_sreset <= '1' when
                        (present_state = Write_GF_Mul_Table_State
                            and end_GF_Mul_Table = '1')
                        or (present_state = Write_GF_Add_Table_State
                            and end_GF_Add_Table = '1')
                        or (present_state = Write_GF_Coef_Table_State
                            and end_GF_Coef_Table = '1')
                    else '0';
    Counter_enable <= '1' when present_state = Write_GF_Mul_Table_State
                        or present_state = Write_GF_Add_Table_State
                        or present_state = Write_GF_Coef_Table_State
                        or present_state = Write_Window_Data_State
                    else '0';

    end_GF_Mul_Table <= '1' when
           count = CONV_STD_LOGIC_VECTOR(GF_OP_TABLE_SIZE-1, ROM_ADDRESS_SIZE)
           else '0';
    end_GF_Add_Table <= end_GF_Mul_Table;
    end_GF_Coef_Table <= '1' when
           count = CONV_STD_LOGIC_VECTOR(COEF_TABLE_DEPTH-1, ROM_ADDRESS_SIZE)
           else '0';

    PRS_Counter_lbl: Generic_Binary_Counter
    generic map (ROM_ADDRESS_SIZE)
    port map (clk => clk, areset => reset, sreset => PRS_Counter_sreset,
        enable => PRS_Counter_enable, count => PRS_count);
    PRS_Counter_sreset <= '1' when present_state = Write_Window_Data_State
                                  and end_GF_Window_Data = '1'
                              else '0';
    PRS_Counter_enable <= '1' when present_state = Write_Window_Data_State
                              else '0';

    end_GF_Window_Data <= '1' when
        PRS_count = CONV_STD_LOGIC_VECTOR(COEF_TABLE_DEPTH-1, ROM_ADDRESS_SIZE)
        else '0';

-------------------------------------------------------------------------------
    -- Coprocessor Control signals
    wr_lbl: D_FF port map (
        d => wr_d, clk => clk, reset => reset, preset => LOW, q => wr);
    wr_d <= '1' when present_state /= Reset_State
                    and present_state /= Idle_State
                else '0';

    rd_lbl: D_FF port map (
        d => rd_d, clk => clk, reset => reset, preset => LOW, q => rd);
```

```
    rd_d <= '0';

    cs_lbl: D_FF port map (
       d => cs_d, clk => clk, reset => reset, preset => LOW, q => cs);
    cs_d <= '1' when present_state /= Reset_State
                     and present_state /= Idle_State
             else '0';

    -- Data Output Multiplexer
    data_mux_d <= CONV_STD_LOGIC_VECTOR(GF_OP_TABLE_SIZE-1, ROM_ADDRESS_SIZE)
                        when present_state = Write_GF_Size_State
                  else Mul_Table when present_state = Write_GF_Mul_Table_State
                  else Add_Table when present_state = Write_GF_Add_Table_State
                  else CONV_STD_LOGIC_VECTOR(POLYNOM_DEGREE, ROM_ADDRESS_SIZE)
                        when present_state = Write_GF_Polynom_Degree_State
                  else Coef_Table when present_state = Write_GF_Coef_Table_State
                  else Window_Table when present_state = Write_Window_Data_State
                  else (others => '0');
    data_mux_lbl: Generic_Register generic map (RegSize => DATA_SIZE)
    port map (clk => clk, enable => HIGH, reset, preset => LOW,
       data => data_mux_d, reg => data_mux);

    -- Output Signals
    wr_l <= not wr;
    rd_l <= not rd;
    cs_l <= not cs;

    data_bus <= data_mux when cs = '1' and wr = '1' else (others => 'Z');

    address_bus_lbl: Generic_Register generic map (RegSize => ADDRESS_SIZE)
    port map (clk => clk, enable => HIGH, reset, preset => LOW,
       data => address_bus_d, reg => address_bus);
    address_bus_d <= "01" when present_state = Write_Window_Data_State
                     else "00";

end arch;


--------------------------------------------------------------------------------
-- M68HC11 Microprocessor Emulation

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

library work;
use work.VENDOR_package.all;

-- Signals ending in _l are asserted LOW (0).
-- All the other signals are asserted HIGH (1).

entity M68HC11 is                       -- M68HC11 Microprocessor Emulation
component
    generic (ADDRESS_SIZE: INTEGER := 2; DATA_SIZE: INTEGER := 8);
    port (
       address_data_bus:    inout STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
                                  -- address/data multiplexed
       as:                  out STD_LOGIC;      -- address strobe
       rd_wr_l:             out STD_LOGIC;      -- read/write select
       e:                   out STD_LOGIC;      -- output clock equal to extal/4
       irq_l:               in STD_LOGIC;       -- interrupt request
       reset_l:             in STD_LOGIC;       -- reset
       extal:               in STD_LOGIC;       -- external input clock
```

```
        ------------------------------------------------------------------------
        -- Testing Interface (not implemented physically)
        -- used for simulation purposes only
        run:                in STD_LOGIC        -- start process
    );
end M68HC11;

architecture arch of M68HC11 is

    constant ROM_ADDRESS_SIZE: INTEGER := 4;
    constant GF_OP_TABLE_SIZE: INTEGER := ROM_ADDRESS_SIZE*ROM_ADDRESS_SIZE;
    constant POLYNOM_DEGREE:   INTEGER := 2; -- Polynom Degree - 1
    constant COEF_TABLE_DEPTH: INTEGER := 3;

    constant MUL_TABLE_DATA_FILE: STRING := "rom_GF4_Mul.mif";
    constant ADD_TABLE_DATA_FILE: STRING := "rom_GF4_Add.mif";
    constant COEF_TABLE_DATA_FILE:STRING := "rom_GF4_Coef3.mif";
    constant WINDOW_DATA_FILE:    STRING := "rom_GF4_PRS3.mif";

    type PossibleStates is (Reset_State, Write_GF_Size_State,
                    Write_GF_Mul_Table_State, Write_GF_Add_Table_State,
                    Write_GF_Polynom_Degree_State, Write_GF_Coef_Table_State,
                    Idle_State, Write_Window_Data_State);
    signal present_state, next_state: PossibleStates;

    signal HIGH, LOW: STD_LOGIC;
    signal reset, extal_1, clk_j, clk_k, clk, wr_d, wr, as_d, as_s: STD_LOGIC;
    signal Mul_Table, Add_Table: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal Coef_Table, Window_Table: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal data_mux_d, data_mux: STD_LOGIC_VECTOR (DATA_SIZE-1 downto 0);
    signal cycle_count: STD_LOGIC_VECTOR (1 downto 0);
    signal cycle: STD_LOGIC_VECTOR (4 downto 1);
    signal address_d, address: STD_LOGIC_VECTOR (ADDRESS_SIZE-1 downto 0);
    signal count, PRS_count: STD_LOGIC_VECTOR (ROM_ADDRESS_SIZE-1 downto 0);
    signal Counter_sreset, Counter_enable: STD_LOGIC;
    signal PRS_Counter_sreset, PRS_Counter_enable: STD_LOGIC;
    signal end_GF_Mul_Table, end_GF_Add_Table: STD_LOGIC;
    signal end_GF_Coef_Table, end_GF_Window_Data: STD_LOGIC;

begin
    HIGH <= '1';
    LOW  <= '0';

-- clk <= extal;
    extal_1 <= not extal;

    reset <= not reset_1;

------------------------------------------------------------------------------
    -- ROM containing the operations defined on a specific GF
    GF_Mul_Table_1bl: rom
    generic map (
        ROM_ADDRESS_SIZE => ROM_ADDRESS_SIZE,
        DATA_SIZE => DATA_SIZE,
        DATA_FILE => MUL_TABLE_DATA_FILE)
    port map (address => count, q => Mul_Table);

    GF_Add_Table_1bl: rom
    generic map (
        ROM_ADDRESS_SIZE => ROM_ADDRESS_SIZE,
        DATA_SIZE => DATA_SIZE,
        DATA_FILE => ADD_TABLE_DATA_FILE)
    port map (address => count, q => Add_Table);
```

```
    GF_Coef_Table_lbl: rom
    generic map (
        ROM_ADDRESS_SIZE => ROM_ADDRESS_SIZE,
        DATA_SIZE => DATA_SIZE,
        DATA_FILE => COEF_TABLE_DATA_FILE)
    port map (address => count, q => Coef_Table);

    GF_PRS_lbl: rom
    generic map (
        ROM_ADDRESS_SIZE => ROM_ADDRESS_SIZE,
        DATA_SIZE => DATA_SIZE,
        DATA_FILE => WINDOW_DATA_FILE)
    port map (address => count, q => Window_Table);

-------------------------------------------------------------------------------
    -- Microprocessor State Machine
    Make_next_state:
    process (present_state) begin

        case present_state is
            when Reset_State =>
                next_state <= Write_GF_Size_State;

            when Write_GF_Size_State =>
                next_state <= Write_GF_Mul_Table_State;

            when Write_GF_Mul_Table_State =>
                if (end_GF_Mul_Table = '1') then
                    next_state <= Write_GF_Add_Table_State;
                else
                    next_state <= Write_GF_Mul_Table_State;
                end if;

            when Write_GF_Add_Table_State =>
                if (end_GF_Add_Table = '1') then
                    next_state <= Write_GF_Polynom_Degree_State;
                else
                    next_state <= Write_GF_Add_Table_State;
                end if;

            when Write_GF_Polynom_Degree_State =>
                next_state <= Write_GF_Coef_Table_State;

            when Write_GF_Coef_Table_State =>
                if (end_GF_Coef_Table = '1') then
                    next_state <= Idle_State;
                else
                    next_state <= Write_GF_Coef_Table_State;
                end if;

            when Idle_State =>
                if (run = '1') then
                    next_state <= Write_Window_Data_State;
                else
                    next_state <= Idle_State;
                end if;

            when Write_Window_Data_State =>
                if (end_GF_Window_Data = '1') then
                    next_state <= Idle_State;
                else
                    next_state <= Write_Window_Data_State;
```

```
            end if;

         when others   =>
            next_state <= Idle_State;

      end case;
   end process;

   Clock_next_state:
   process (reset, clk) begin
      if (reset = '1') then
         present_state <= Reset_State;
      elsif (clk'event and clk = '1') then
         present_state <= next_state;
      end if;
   end process;

-------------------------------------------------------------------------------
   Counter_lbl: Generic_Binary_Counter
   generic map (ROM_ADDRESS_SIZE)
   port map (clk => clk, areset => reset, sreset => Counter_sreset,
      enable => Counter_enable, count => count);
   Counter_sreset <= '1' when (present_state = Write_GF_Mul_Table_State
                              and end_GF_Mul_Table = '1')
                          or (present_state = Write_GF_Add_Table_State
                              and end_GF_Add_Table = '1')
                          or (present_state = Write_GF_Coef_Table_State
                              and end_GF_Coef_Table = '1')
                     else '0';
   Counter_enable <= '1' when present_state = Write_GF_Mul_Table_State
                          or present_state = Write_GF_Add_Table_State
                          or present_state = Write_GF_Coef_Table_State
                          or present_state = Write_Window_Data_State
                     else '0';

   end_GF_Mul_Table <= '1' when
          count = CONV_STD_LOGIC_VECTOR(GF_OP_TABLE_SIZE-1, ROM_ADDRESS_SIZE)
          else '0';
   end_GF_Add_Table <= end_GF_Mul_Table;
   end_GF_Coef_Table <= '1' when
          count = CONV_STD_LOGIC_VECTOR(COEF_TABLE_DEPTH-1, ROM_ADDRESS_SIZE)
          else '0';

   PRS_Counter_lbl: Generic_Binary_Counter
   generic map (ROM_ADDRESS_SIZE)
   port map (clk => clk, areset => reset, sreset => PRS_Counter_sreset,
      enable => PRS_Counter_enable, count => PRS_count);
   PRS_Counter_sreset <= '1' when present_state = Write_Window_Data_State
                                 and end_GF_Window_Data = '1'
                          else '0';
   PRS_Counter_enable <= '1' when present_state = Write_Window_Data_State
                          else '0';

   end_GF_Window_Data <= '1' when
      PRS_count = CONV_STD_LOGIC_VECTOR(COEF_TABLE_DEPTH-1, ROM_ADDRESS_SIZE)
      else '0';

-------------------------------------------------------------------------------
   cycle_count_lbl: Generic_Binary_Counter generic map (2)
   port map (clk => extal_1, enable => HIGH, areset => reset, sreset => LOW,
      count => cycle_count);

   cycle_lbl: for i in 1 to 4 generate
```

```
        cycle(i) <= '1' when cycle_count = CONV_STD_LOGIC_VECTOR(i-1, 2)
                    else '0';
    end generate;

    clk_lbl: JK_FF port map (clk => extal, reset => reset, preset => LOW,
        j => clk_j, k => clk_k, q => clk);
    clk_j <= '1' when cycle_count = CONV_STD_LOGIC_VECTOR(0, 2) else '0';
    clk_k <= '1' when cycle_count = CONV_STD_LOGIC_VECTOR(2, 2) else '0';

-------------------------------------------------------------------------------
    -- Coprocessor Control signals

    -- output clock
    e <= cycle_count(1);        -- equal to extal/4

    -- read/write select
    wr_lbl: D_FF port map (
        d => wr_d, clk => clk, reset => reset, preset => LOW, q => wr);
    wr_d <= '1' when present_state = Write_GF_Size_State
                  or present_state = Write_GF_Mul_Table_State
                  or present_state = Write_GF_Add_Table_State
                  or present_state = Write_GF_Polynom_Degree_State
                  or present_state = Write_GF_Coef_Table_State
                  or present_state = Write_Window_Data_State
              else '0';
    rd_wr_l <= not wr;

    -- address strobe
    as_lbl: D_FF port map (
        d => as_d, clk => extal, reset => reset, preset => LOW, q => as_s);
    as_d <= '1' when (wr_d = '1' and cycle(1) = '1') else '0';
    as <= as_s;


    -- Data Output Multiplexer
    data_mux_d <= CONV_STD_LOGIC_VECTOR(GF_OP_TABLE_SIZE-1, ROM_ADDRESS_SIZE)
                            when present_state = Write_GF_Size_State
                  else Mul_Table when present_state = Write_GF_Mul_Table_State
                  else Add_Table when present_state = Write_GF_Add_Table_State
                  else CONV_STD_LOGIC_VECTOR(POLYNOM_DEGREE, ROM_ADDRESS_SIZE)
                            when present_state = Write_GF_Polynom_Degree_State
                  else Coef_Table when present_state = Write_GF_Coef_Table_State
                  else Window_Table when present_state = Write_Window_Data_State
                  else (others => '0');
    data_mux_lbl: Generic_Register generic map (RegSize => DATA_SIZE)
    port map (clk => clk, enable => HIGH, reset => reset, preset => LOW,
        data => data_mux_d, reg => data_mux);

    -- Output Signals
    address_data_bus <= address when as_s = '1'
                        else data_mux when wr = '1'
                        else (others => 'Z');

    address_lbl: Generic_Register generic map (RegSize => ADDRESS_SIZE)
    port map (clk => clk, enable => HIGH, reset => reset, preset => LOW,
        data => address_d, reg => address);
    address_d <= "01" when present_state = Write_Window_Data_State
                 else "00";

end arch;
```

Annex B. PRS Extractor Software from Simulation Data

```
/*****************************************************************************
    PRS_extractor.C

    This program extracts the Pseudo-Random Sequence from an
    ALTERA MAX+PLUS II simulation table file
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>


void main(int argc, char *argv[ ]) {

    FILE *src_file, *dest_file;
    char *line, *save_line, *gf_symbol, gf_sync;
    int pos, pattern, symbol, counter;

    if (argc != 3) {
        printf("\nUsage: prs_extractor source_file destination_file\n");
        exit(0);
    } else {
        printf("\n");
        printf("\nPRS Extractor from Altera MAX+PLUS II Simulation Table File.");
        printf("\nCopyright Adrian Moica, University of Ottawa, 2000.\n");
    }

    // Open for read (will fail if file "data" does not exist)
    if( (src_file  = fopen( argv[1], "rt" )) == NULL ) {
        printf( "Can not open the file %s\n", argv[1]);
        exit(0);
    }

    // Open for write
    if( (dest_file = fopen( argv[2], "wt" )) == NULL ) {
        printf( "Can not open the file %s\n", argv[2]);
        exit(0);
    }

    pattern = 0;
    gf_sync = 0;
    counter = 0;

    while(fgets( line, 100, src_file ) != NULL) {
        save_line = line;          // save the pointer

        switch (pattern) {
            case 0:
                if(*line == '%') {
                    pos = 0;
                    do {
                        line++;
                        pos++;
                    } while((*line != '%') && (*line != '\0'));
                    if ((*line == '%') && (pos != 0))
                        pattern = 1;
                }
```

```
            break;
         case 1:
            if(*line != '%')
               pattern = 2;
            break;
         case 2:
            // get to the '=' character
            while ((*line != '=') && (*line != '\0'))
               line++;

            if (*line == '=') {
               line++;       // *line = ' '
               line++;       // *line = gf_sync

               if ((*line == '0') && (gf_sync == '1')) {
                  gf_sync = *line;

                  line++;       // *line = ' '
                  line++;       // *line = gf_symbol
                  gf_symbol = line;
                  while ((*line != ' ')&&(*line != '\n')&&(*line != '\0'))
                     line++;

                  if (gf_symbol != line)
                     *line = '\0';    // end of gf_symbol string

                  // Eliminate the leading zeros in gf_symbol and
                  // save it in the destination file
                  symbol = atoi(gf_symbol);
                  _itoa(symbol, gf_symbol, 10);
                  fputs(gf_symbol, dest_file);
                  fputc(' ', dest_file);
                  counter++;

               } else
                  gf_sync = *line;
            }
            break;
      }

      line = save_line;       // redo the pointer
   }


   fputs("\nlength = ", dest_file);
   _itoa(counter, gf_symbol, 10);
   fputs(gf_symbol, dest_file);

   // Close input-output streams
   if(fclose(src_file))
      printf( "Can not close the file %s\n", argv[1]);
   if(fclose(dest_file))
      printf( "Can not close the file %s\n", argv[2]);
};
```