Vectorization of code generation in CDMA

Nas, R.J.M.

Award date:
2003

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computer Science

MASTER'S THESIS

Vectorization of Code Generation in CDMA

by
R.J.M. Nas

Supervisor: prof. dr. C. H. van Berkel

Eindhoven, April 2003

Abstract

This master's thesis presents the design of a flexible (i.e. re-configurable) and high throughput Code Generator Unit (CGU). The specific functions we look at are Pseudo Random Noise (PRN) generation, the closely related Hadamard and Orthogonal Variable Spreading Factor codes, and Cyclic Redundancy Check (CRC). The resulting CGU consists of functional units that perform these functions and a unit that combines the output of these functional units in a flexible way. The throughput of the CGU is limited by the throughput of its PRN generator. Our specific PRN generator is implemented by a Linear Feedback Shift Register (LFSR) and its high throughput is achieved by adapting it such that multiple sequence bits are generated in parallel, while keeping the circuits latency relatively low. Conventional parallel implementations of an LFSR have a latency that scales linearly with the amount of parallelism. We use a mathematical model of a parallel LFSR and prove the existence of a factorization of the next-state function for this LFSR. This factorization is the specification of a design with a latency that scales logarithmically with the amount of parallelism in the LFSR. Implementation (down to transistor level) of this specification is straightforward; a specific instance, an LFSR of length 32 generating 16 bits in parallel every clock cycle, was implemented using standard cells and the resulting circuit runs at over 300 Mhz. in 0.18 micron technology. This equates to a throughput of approx. 5 Gb/s. The design is also fully and on-the-fly re-configurable.
The CGU's initial target application area is Third Generation Mobile Communications (3G), because of the project context in which this research took place. There are multiple other applications that can benefit from a high throughput LFSR as well, such as the Global Positioning System (GPS), CRC, cryptography (encryption/decryption) and Built-In Self-Test, both test vector generation and signature analysis.

Table of contents

Preface

This document is my Master's thesis and is the result of a graduation project to obtain the degree of Master of Science in Computer Science with a specialization in Parallel Systems. Most of the work has been performed in de Embedded Systems Architectures on Silicon (ESAS) group at Philips Research in Eindhoven.

This document describes my work on the design of the Code Generation Unit for a programmable, multi standard, Third Generation (3G) wireless communication solution Philips is working on. As such, this documents intended audience has a technical background and is interested in wireless communication.

I would like to thank: Kees van Berkel for his excellent supervision at Philips Research, Rob Takken for his help with several VHDL and synthesis related issues and Daniel Timmermans, Edwin Rijpkema and the members of the ESAS coffee club in general for their help, idea's and interesting discussions that were not necessarily all related to my work. Finally I would like to thank Henk van Tilborg from the Eindhoven University of Technology for his help in understanding, and attempt at dealing with, quaternary Pseudo Random Noise (PRN) sequences.

Rick Nas,
Eindhoven, February 2003

Glossary

3G	Third Generation
BIST	Built-In Self Test
C/A	Coarse/Acquisition
CDMA	Code Division Multiple Access
CGU	Code Generation Unit
CRC	Cyclic Redundancy Check
CVP	Co-Vector Processor
DS-CDMA	Direct Sequence CDMA
DSP	Digital Signal Processor
FDMA	Frequency Division Multiple Access
GPRS	General Packet Radio Service
GSM	Global System for Mobile Telecommunications
GPS	Global Positioning System
IC	Integrated circuit
ISA	Instruction Set Architecture
LFSR	Linear Feedback Shift Register
LUT	Look Up Table
Mbps	Mega bits per second
MHz	Mega hertz
OVSF	Orthogonal Variable Spreading Factor
PRN	Pseudo Random Noise
SA	Signature Analysis
SF	Spreading Factor
SIMD	Single Instruction Multiple Data
SW-MODEM	Software Modem
TD-SCDMA	Time Division Synchronous CDMA
TDD-CDMA	Time Division Duplex CDMA
TDMA	Time Division Multiple Access
TLU	Table Look-Up
TU/e	Technische Universiteit Eindhoven (Eindhoven University of Technology)
UMTS	Universal Mobile Telecommunications System
VLIW	Very Long Instruction Word
VLSI	Very Large-Scale Integration

1 Introduction

In this section we introduce several concepts and technologies that are relevant to the context of the assignment discussed in this document. More topics that need to be introduced will follow, but these are quite specific and we will save their introduction until they become relevant.

1.1 Scope

This document is written as a graduation report for the Eindhoven University of Technology (TU/e) and as a research report for Philips Research Laboratories Eindhoven. The context of this assignment is the Software Modem project (SW-MODEM). The aim of the project is to design a low-cost software modem that can support multiple Third Generation (3G) wireless standards as well as the evolution of these standards. More specifically, to design a fully programmable domain-specific co-processor that, together with a conventional "scalar" micro controller or digital signal processor:

- Supports all base-band processing for 3G wireless standards;
- Supports the evolution of these standards by means of software upgrades;
- Is low costs, i.e. has a competitive silicon footprint.

1.2 3G wireless communication

As the successor to Global System for Mobile Telecommunications (GSM, 2G) and General Packet Radio System (GPRS, 2.5G), the third generation mobile radio system is supposed to finally bring broadband Internet access to mobile phones around the world. Broadband, packet-based transmission of text, digitized voice, video, and multimedia at data rates up to and possibly higher than 2 Mbps will be become possible. Different 3G standards have been proposed and several have found acceptance somewhere in the world. The Universal Mobile Telecommunications System (UMTS) is expected to become the prevailing standard in Europe, while the United States seem to be going with CDMA2000 and China is developing Time Division Synchronous Code Division Multiple Access (TD-SCDMA).
While there are certainly differences between these standards, they have at least one important characteristic in common: Wideband Code Division Multiple Access (W-CDMA) has emerged as the mainstream air interface solution for the third generation networks.

"Multiple Access" indicates that the common transmission medium is shared between users of the system. There are basically three multiple access schemes, which are illustrated in figure 1.1 [1]

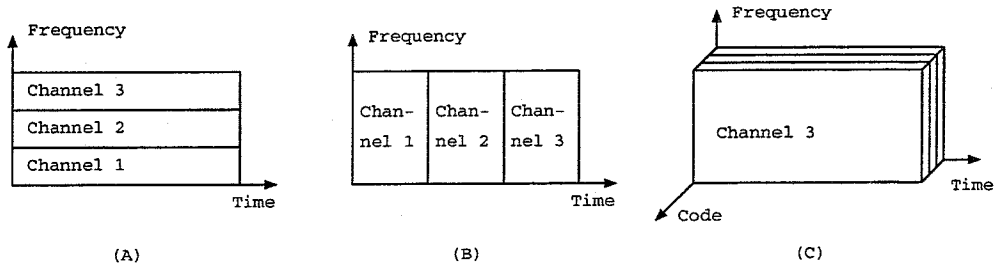


Figure 1.1 Multiple Access schemes: (A) FDMA, (B) TDMA and (C) CDMA.

There are also hybrid schemes, like TD-CDMA, where multiplexing is still achieved with CDMA, but TDMA is used to duplex the uplink and downlink of a signal on the same frequency band.

1.3 CDMA

In CDMA multiple access is achieved by assigning each user a pseudo-random sequence (codeword). Pseudo-random indicates that the codes are not truly random, but deterministically generated and reproducible with an algorithm. The pseudo-random sequence is then used to transform the user's signal into a wide band spread spectrum signal. This transformation is usually achieved in one of three different ways. In frequency hopping spread spectrum the codeword defines the transmission frequency. In time hopping spread spectrum the codeword defines the transmission moment. Finally, in Direct Sequence (DS)-CDMA the information signal is multiplied by the codeword, which results in a wideband signal. DS-CDMA is the technique used in 3G. These three different techniques are illustrated in figure 1.2 [1].

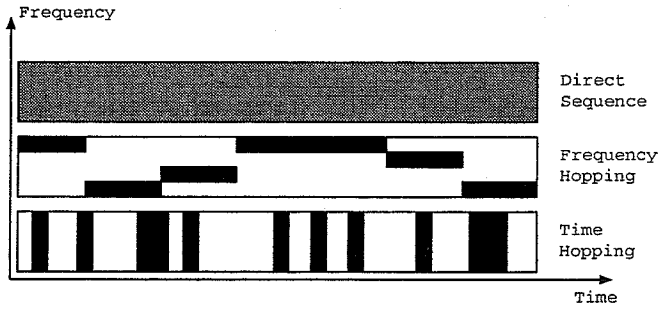


Figure 1.2 Direct sequence, frequency hopping and time hopping CDMA.

On the receiving side of the transmission the same sequence is generated. The sequence is then used to extract the user's signal from the received signal by either listening (to the right frequency) at the right time, or by correlating the sequence with the received signal and thereby lifting the user's signal above the noise. DS-CDMA is illustrated in figure 1.3 [2]

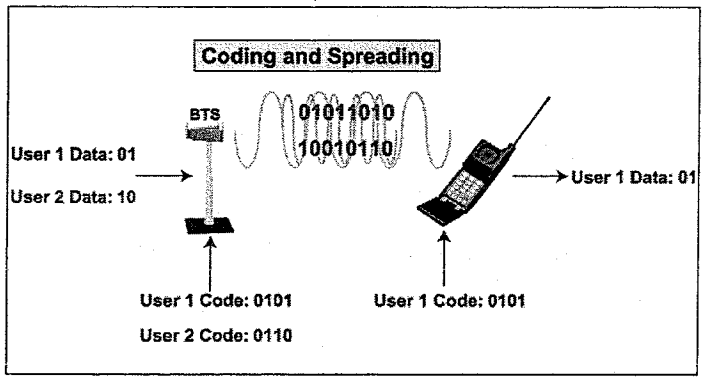


Figure 1.3 Coding en Spreading in CDMA.

CDMA has several important advantages over T(ime)DMA and F(requency)DMA, the most important of which are universal frequency reuse, power control and rake receiver. Below we give a short description of these three advantages.

In both TDMA and FDMA neighboring cells in the network cannot use the same set of frequencies because otherwise users in different cells would interfere with each other. In

CDMA users are separated by code channels, not frequency channels and therefore each cell in the network has access to the entire frequency band. Universal frequency reuse is illustrated in figure 1.4 [2].

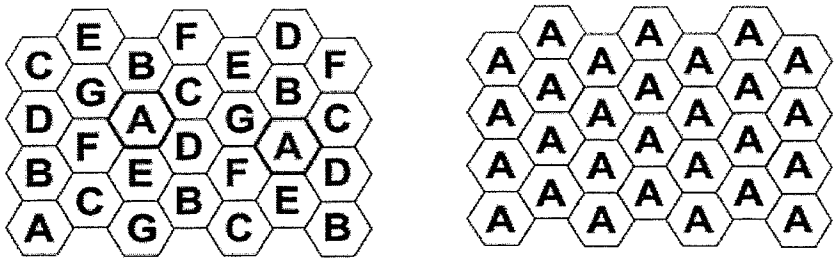


Figure 1.4 Frequency use in TDMA/FDMA versus CDMA.

Power control is a feature that enables mobiles to adjust the power at which they transmit. This ensures that the base station receives all signals at an appropriate power. The CDMA network determines, for each mobile individually, the power at which it should transmit its signal. If all mobiles transmitted at the same power level, the base station would receive unnecessarily strong signals from mobiles nearby and extremely weak signals from mobiles that are far away, which is know as the near-far problem. As a result the capacity of the system would be reduced because it is no longer possible to lift weak signals above the strong signals through correlation. Power control is a necessary technique that solves this problem and has the added benefit that a mobile never uses too much power to send a signal, thus reducing power usage. Power control is illustrated in figure 1.5 [2]. It should be noted that Power control is not unique to CDMA; GSM also has power control, but there it is not as crucial to the correctness of the technique.

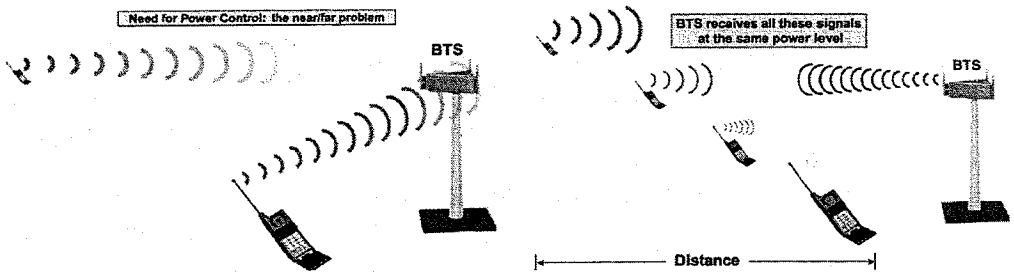


Figure 1.5 Using power control to solve the near-far problem.

A rake receiver is a CDMA feature that turns what is a problem in other technologies into an advantage for CDMA. Signals sent over the air can take a direct path to the receiver, or they can bounce off objects and then travel to the receiver. These different paths, called multi-paths, can result in the receiver getting several versions of the same signal but at slightly different times. Multi-paths can cause a loss of signal through cancellation in other multiple access schemes. CDMA's rake receiver implements multiple receivers in one. The rake receiver identifies the strongest multi-path signals and combines them to produce one very strong signal. A rake receiver therefore uses multi-path to reduce the power at which the transmitter must send. A rake receiver is illustrated in figure 1.6 [2].

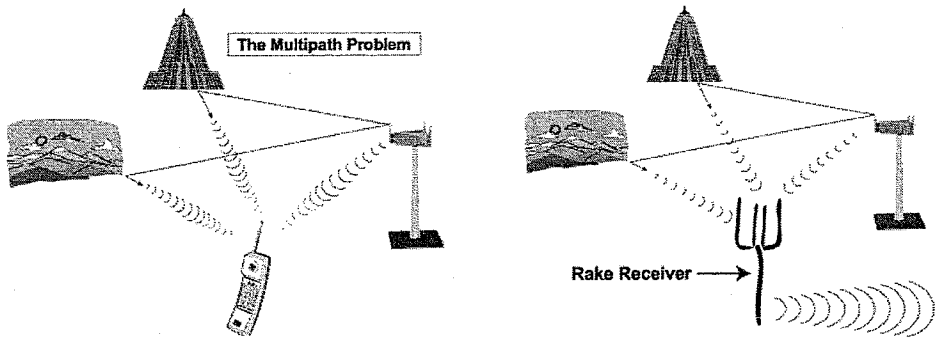


Figure 1.6 Using a rake receiver to solve the multi-path problem.

Besides these advantages there are a couple of advantages that are inherent to spread spectrum modulation. Because a signal is spread across a wide frequency band it looks like random low energy noise to anyone who is trying to listen but doesn't have the right codewords. This makes it harder for anyone to either eavesdrop on the signal or to scramble it by sending high-powered signals at certain frequencies.

All these benefits do not come for free however. Spreading a signal with DS-CDMA and extracting a specific signal from the ether on the receiving end is no easy task and all in all CDMA-reception is a very computationally intensive technique. On a conventional DSP enhanced with some specific instructions, UMTS would require between 3 and 5 billion operations per second of computing power, an order of magnitude more than any current Digital Signal Processor (DSP) can deliver. Solving this by just harnessing the power of several DSPs is not a very economical solution and using task specific hardware accelerators in combination with a conventional DSP makes for a very standard specific architecture. What we are looking for is an architecture that is both fast and flexible enough to handle several different standards. The SW-MODEM project is developing the Co-Vector Processor (CVP) with that goal in mind.

1.4 Co-Vector Processor

Using the terms loosely, the CVP can be described as a Very Long Instruction Word (VLIW) Single Instruction Multiple Data (SIMD) processor. The SIMD is the most obvious, because as the name says, the CVP is operating on vectors of arguments. VLIW comes from the fact that every instruction for the CVP actually consists of instructions for each of its functional units. Figure 1.7 [3] shows the CVP embedded in the Software Modem (SW-MODEM) architecture. This figure also shows that the CVP is in fact a co-processor to a conventional DSP or micro-controller.

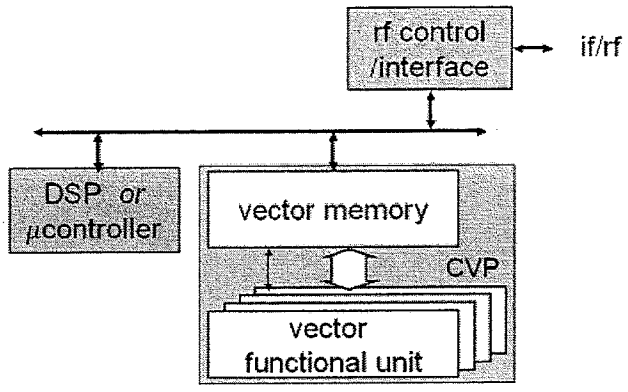


Figure 1.7 CVP in the SW-MODEM architecture.

The idea behind the SW-MODEM project is that, assuming a high enough clock speed and vector width and the possibility of vectorizing a large percentage of the algorithms that play a role in CDMA, this architecture is capable of delivering the computational power that is required by CDMA.

Inside the CVP we find a vector memory and several functional units. These functional units perform functions like shifting and shuffling vectors and inter- or intra-vector arithmetic. Besides these functional units there is the Code Generation Unit (CGU), which is responsible for generating the pseudo-random sequences we spoke of earlier and several other codes that we will come to speak of. The design of the CGU is the topic of the rest of this document.

2 Problem Description

2.1 Assignment

The initial goal of this project is the design of a high speed, flexible (i.e. multi standard) CDMA code generation unit. This description consists of several sub goals.

First we need to investigate several speed aspects of the CGU. Because of the vector nature of the CVP, the CGU will need to be capable of generating an entire vector full of code-sequence elements (chips) every clock cycle, in order to use the full width of the data path. In the current design of the CVP this means that we need to generate 16 chips every clock cycle. Second, because of the high load some of the CDMA kernels have on the CVP, the clock rate of the chip is ideally around 300 MHz (in 0.12 micron CMOS technology). This speed is most likely unattainable without some clever design decisions. Finally, when specifying an Instruction Set Architecture (ISA) for the CGU, instruction latencies, introduced by for example pipeline stages, should be kept to a minimum, to avoid latency penalties after a state change or re-configuration instruction.

A second, more implicit, sub goal of the project is to list applications and application standards that use the codes like the ones in CDMA and see if the CGU is relevant to them. A subset of these applications is then selected and a specification of the CGU's minimum functionality is then derived from their standards. Of principle interest are the three prevalent 3G standards: UMTS, CDMA2000 and TD-SCDMA. Besides these we will look at the Global Positioning System (GPS) and a seemingly completely unrelated application: Cyclic Redundancy Check (CRC) and Signature Analysis (SA).

Besides being a vector processor, a second important characteristic of the CVP is its programmability. This programmability significantly enlarges the application domain of the CVP. Obviously the programmability of the CVP is limited by the flexibility of its functional units. In the CGU we will try to achieve high flexibility by supporting, in hardware, a set of functions that is more or less shared among the different applications in the target application domain. Some of these functions will be configurable in order to compensate for small differences between applications. The functions we will look at include: Linear Feedback Shift Registers and Hadamard Code generation. With such an architecture we hope to support large parts of different standards and applications in hardware and solve remaining differences in software.

2.2 Previous work

The design described in this document is not the first attempt at designing the CGU. Previously, Tom Geelen, a student from the TU/e, designed a CGU as his internship assignment at Philips Research in 2001 [4,5]. This version of the CGU implemented some of the required features like vector generation of codewords and some of the required configuration possibilities, but it was to slow. Tom Geelen estimated that at a vector width of 8 his design could be made to run at around 100 MHz (in 0.18 micron technology) and this figure would drop dramatically when going to the preferred vector width of 16. Also, the approach he took at vectorizing Pseudo Random Noise (PRN) sequence generation turned out to have been patented by IBM in 1995 [6]. It doesn't lend itself very well for high clock speeds, so it is unlikely that Tom Geelen's design can be easily adapted to run at the required clock speed of 250-300 MHz. Instead, we will have to take an entirely new approach, which is an important topic in this document.

3 CGU Overview

In this section we present the initial high-level architecture and in subsequent sections we elaborate on the implementation of its various elements. Obviously this initial design has to be based on some requirements, which we define in the next paragraph.

3.1 Requirements

We base the functional requirements of the CGU on the specification documents of several 3G standards and other applications that fit well on the CGU.

On of the most important characteristics of the CVP is that it will be programmable for several different 3G standards. However, targeting all of the 3G standards at the first try would be a bit over ambitious, so we decide to focus our main attention on UMTS.

UMTS
UMTS will initially be the most important application for the CVP and it provides ample opportunities to add and test the flexibility to the design. Code generation for UMTS is specified in [9]. This documents introduces and specifies several different codes and how these are combined with each other and the data signal.

Other 3G standards
The 2 other prevailing 3G standards are TD-SCDMA in China and CDMA2000 in the US. Ultimately, the CVP and thus the CGU will be compatible with these standards, but for now this is not a top priority. We occasionally look at parts from their respective specifications [12 and 13] to see where adjustments to the design are needed to accommodate these standards.

GPS
The Global Positioning System uses a network of earth orbiting satellites to enable a GPS-receiver to determine its position on earth. The one-way communication between the satellites and the receiver uses CDMA as its multiple access scheme, which makes it a potential target application for the CVP. Consumer electronics use the coarse mode [10,11] (C/A-mode) of GPS, which has an implementation that is almost completely compatible with UMTS.

CRC and SA
Signature analysis, as used in CRC and built-in self-tests for integrated circuits (IC) use hardware that is very similar to the hardware that we use to generate some of the codes required for UMTS. A big difference with CMDA is that CRC and SA process an input stream and only occasionally produce output. This requires that the CGU be fitted with a port for this input stream. Requirements for CRC and SA in Very Large-Scale Integration (VLSI) are presented in [7,8].

Besides these functional requirements, there are also some practical requirements that come from the operational context of the CVP. First of all, since we are designing flexible hardware, we will introduce configuration parameters that define the exact operation of a functional unit. These parameters can be used to specify a specific mode of operation or to supply the CGU with rarely changing constants, which would otherwise have to be included in, for example, the instruction-word. The ability to configure these parameters introduces requirements for both the i/o-interface of the CGU and its ISA. The CVP is a VLIW co-processor, which means that the CGU, as a separate functional unit, will have

its own ISA. Furthermore, the CVP uses multitasking to be able to handle several different tasks in parallel. Multi-tasking implies task switching, which requires the ability to save/restore the state of the entire CVP and thus also of the CGU. Saving/restoring the state of the CGU will add a couple of instructions to the ISA and it also imposes some requirements on the i/o-interface.

3.2 Detailed Specification

Initially, the CGU will support UMTS, GPS and CRC and SA. All these applications have very detailed specifications [9, 10/11 and 7/8 respectively], but for the CGU only the specifications of the required codes are important. We will now give a list of these codes, with some explanation.

Name	Specification
$C_{sig,SF,s}$	This is a Hadamard code. The length of the code, also called the spreading factor (SF) determines how many codewords there are, and the argument s specifies which of these SF codewords is needed. The argument s is part of the configuration for the unit that generates this code. The output of the functional unit should be a vector of elements from the $C_{sig,SF,s}$ code.
$C_{ch,SF,n}$	This is an Orthogonal Variable Spreading Factor (OVSF) code. It is very similar to the Hadamard code, and will be generated by the same unit.
$c_{long,1,n}$ $c_{long,2,n}$	$=(x_n(i) + y(i))(0,1 := 1,-1)$ $=(x_n(i+16777232) + y(i+16777232))(0,1 := 1,-1)$ $x_n : G(x) = X^{25}+X^3+1$ $y : G(x) = X^{25}+X^3+X^2+X+1$ These two are both the sums of two Pseudo Random Noise (PRN) sequences x_n and y, which are generated by a Linear Feedback Shift Register (LFSR). These LFSRs are specified by two generator polynomials *G(x)*, which are both part of the configuration of the unit that generates them. The code-number n specifies the initial state of the generator unit. Notice $c_{long2,n}$ is a delayed version of $c_{long,1,n}$. The amount of delay is determined by two more polynomials, which are both also part of the configuration for the unit that generates these codes. The output of the functional unit should be two vectors of elements, one for each of these two codes.
$c_{short,1,n}$ $c_{short,2,n}$	$=(a(i)+2b(i)+2d(i))(0,1,2,3 := 1,-1,-1,1)$ $=(a(i)+2b(i)+2d(i))(0,1,2,3 := 1,1,-1,-1)$ $a : G(x) = X^8+X^5+3X^3+X^2+2X+1$ //Quaternary code $b : G(x) = X^8+X^7+X^5+X+1$ $d : G(x) = X^8+X^7+X^5+X^4+1$ $c_{short,1,n}$ and $c_{short,2,n}$ are very similar to the two c_{long} codes. In this case however, we are summing not two but three PRN codes, one of which, a, is not a binary, but quaternary code. This means that every chip can have four different values instead of two. Again we see some generator polynomials, which will be part of the configuration. The code-number n specifies the initial state of the generator. Again, the output will be two vectors of sequence elements.
Z_n	$=(x(i) + y(i))(0,1 := 1,-1)$ $x : G(x) = X^{18}+X^7+1$ $y : G(x) = X^{18}+X^{10}+X^7+X^5+1$ Z_n is basically the same as the c_{long} codes, except for different configuration parameters. Note that we will also need a delayed version of Z_n, just like we saw with $c_{long,2,n}$, but this is again part of the configuration of the functional unit that will generate this code.
GPS C/A-code	$=G1(t) + G2(t + i)$ $G1 : G(x) = X^{10}+X^3+1$ $G2 : G(x) = X^{10}+X^9+X^8+X^6+X^3+X^2+1$ The GPS code that we need to support is the sum of a PRN sequence and a delayed version of a PRN code. We saw this before, only with different configuration parameters.

Besides these codes, there are several codes that further combine the codes we just introduced into complex valued sequences, which are the final output of the CGU. Note, that in some cases the above codes are not combined any further and become the final output of the CGU themselves.

Name	Specification
$S_{dl,n}(k)$	$=Z_n(k) + j\ Z_n(k + 131072)$ $S_{dl,n}$ combines a normal and a delayed version of the Z_n code we just saw.
$C_{pre,n,s}(k)$ $C_{c\text{-}acc,n,s}(k)$ $C_{c\text{-}cd,n,s}(k)$	$=c_{long,1,n}(k) * C_{sig,s}(k) * e^{j(\pi/4 + k\pi/2)}$ This combination is a bit more complex than the previous one. The multiplication is used is introduced to work more easily with complex values. To facilitate this, the binary sequences that are used are first mapped to the real values {-1,1}. The multiplication with the power of *e* makes the sequence complex valued.
$C_{long,n}(k)$ $C_{short,n}(k)$	$=c_{long,1,n}(k) * (1 + j(-1)^k * c_{long,2,n}(2 * \lfloor k/2 \rfloor))$ $=c_{short,1,n}(k) * (1 + j(-1)^k * c_{short,2,n}(2 * \lfloor (k/2 \rfloor))$ Again a combination that maps binary valued sequences onto the complex domain. Note how not all the elements from the $c_{long,2,n}$ code are actually used.

The UMTS standard defines some more codes, but these are only other names for the same codes, to be used in different circumstances. The above codes specify the CGU requirements with respect to UMTS and GPS. The third application that we decided to implement is CRC and SA. This application is different that UMTS and GPS, in that it doesn't generate a code-sequence, but instead consumes one. This requires that the functional unit that implements CRC can receive input. CRC output only occurs after a complete input code-sequence has been consumed. The output of the CRC unit will be a scalar. Internally, a CRC unit is implemented with a LFSR, which is specified by a generator polynomial that will be part of the configuration for the CRC unit.

3.3 Architecture Overview

The CGU will be a functional unit of the CVP. The CVP has two different data-paths: a vector data-path and a scalar data-path. Figure 3.1 shows a black-box model of the CGU, with its interfaces, together with an overview of what information uses which data-path.

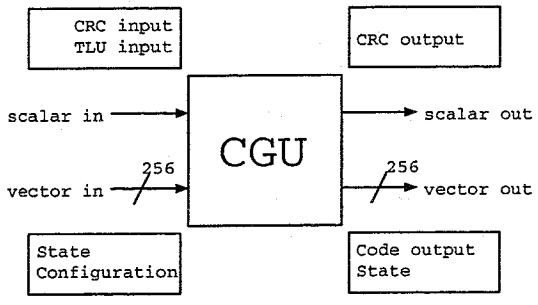


Figure 3.1 Black-box model of the CGU

The scalar path is used to supply the input sequence of which a CRC signature needs to be calculated. Once this signature is calculated, the CGU outputs it over the scalar path. We could have used the vector port for this output, but that provided no extra value. The vector input is used to read configuration parameters from the vector memory. It is also used to read in a state vector after a context switch. The difference between state and configuration is that configuration vectors are constants that change very rarely and certainly not every context switch, while state vectors are very variable and are

saved/restored every context switch. Since the configuration parameters aren't changed by the CGU, we never need to save them. The state on the other hand does change and therefore we need the vector output to save it when a context switch occurs. The actual output of the CGU, the generated codeword, is also sent over the vector output port. We could have used the scalar in and output ports to load and save state and configuration. However, the scalar path is not wide enough to send all the state or all the configuration data in one packet. So, sending this information over the scalar path would require several clock-cycles, which introduces an unnecessary latency.

The specifications of the codes that we need to support shows that the actual output of the CGU is a combination of several basic codes that may or may not be from the same type.

Looking at the specification, we see that we need a Hadamard/OVSF Code Generator to generate the so-called channelisation and synchronization codes. The detailed design of the Hadamard/OVSF Code Generator is presented in paragraph 6.1.
Looking further we see that a very important class of codes in both UMTS and GPS (but also in TD-SCDMA and CDMA2000) is the class of PRN sequences, specifically the ones generated by an LFSR. The design of these PRN Generators turned out to be by far the most challenging topic of this project and we will come to speak of them in detail in chapters 4 and 5. The parallel generation of several PRN sequences requires the inclusion of at least 2 of these PRN Generators.
Finally, for added flexibility we add Table Look-Up (TLU) functionality to the architecture, giving us the possibility to use codes for which we are not prepared to design dedicated hardware. The TLU table will be located in the Vector Memory Unit (VMU), and we decide to use the scalar input to retrieve values from this memory for use in the CGU. TLU-functionality and a situation, occurring in UMTS, that uses it is presented in paragraph 6.2. With this information we can fill in the black-box from Figure 3.1. This more detailed architecture is presented in Figure 3.2.

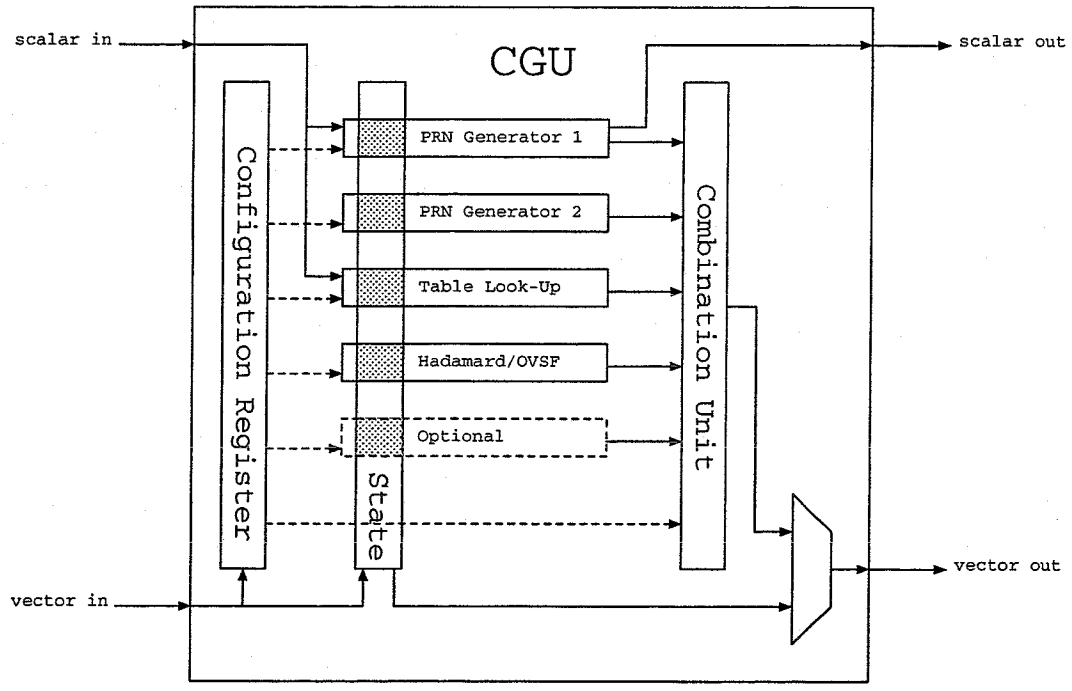


Figure 3.2 CGU Architecture

This architecture clearly shows how the TLU-functionality is implemented over the scalar path. It also shows how one of the PRN Generators can receive input from and generate output for the scalar path; this is used to implement the required CRC/SA functionality.

Both PRN Generators could be extended to provide this functionality, but we choose not to, for sake of simplicity.

All functional units, including the Code Combiner have access the configuration vector. The state vector is distributed over the functional units. Note however that the Code Combiner is stateless. The design of the Code Combiner is discussed in paragraph 6.3.

The modular nature of this design makes it easy to add new or extra code generators to the design. They can simply be added and their output needs to be connected to the Code Combiner in the required way. Looking at for example CDMA2000 it seems likely that another Hadamard Generator is needed to generate some of the required codes. How this extra generator is used in CDMA2000 can then be reflected in changes made to the Code Combiner.

In the next chapter we take a detailed look at Linear Feedback Shift Registers, the basis for the design of the PRN generators.

4 Vectorization of Linear Feedback Shift Registers

4.1 LFSR introduction

In CDMA a specific class of PRN sequences is used, namely the class of sequences generated by the recurrence relation

$$x_i = \sum_{j=1}^{N} g_{N-j}\, x_{i-j} \qquad i = N,\ N+1,\ N+2,\ \ldots \tag{4.1}$$

where, $x_i \in \{0,1\}$ are output sequence digits; $x_0..x_{N-1}$ represents an "initial state"; $g_0..g_{N-1}$ are given constants and the summation sign represents addition modulo 2. While this form of sequence generation can also be performed in other finite fields, we will assume GF(2) in the rest of this document, unless explicitly stated otherwise. A major advantage of this generation method is the simplicity of its implementation. The hardware of a generator that functions in accordance with expression (4.1) comprises only an N-bit shift register and a set of modulo-2 adders to generate feedback. The resulting circuit is called a Linear Feedback Shift Register (LFSR). There are two basic ways to implement an LFSR. The so-called Fibonacci implementation (Figure 4.1) consists of a simple shift register in which a binary-weighted modulo 2 sum of the register values (taps) is fed back to the input. The register taps are represented with a triangle and the weighing factor; a crossed circle represents addition.

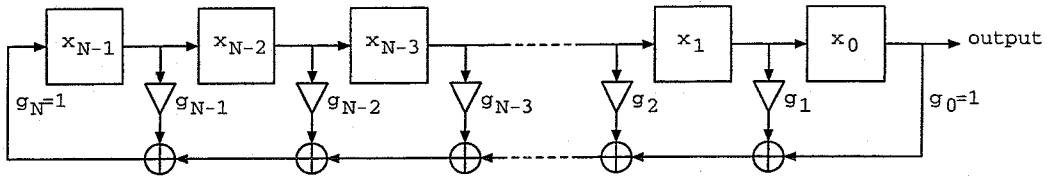


Figure 4.1 Fibonacci LFSR

The Galois implementation (Figure 4.2) consists of a shift register, the contents of which are modified at every step by a binary-weighted value of the output stage. A Galois LFSR has a lower latency than an equivalent Fibonacci LFSR because all additions can be performed in parallel, whereas in a Fibonacci LFSR, the additions can merely be balanced, resulting in a logarithmic evaluation time. Galois LFSRs are generally faster than Fibonacci LFSRs and because of this Tom Geelen used them in his design of the CGU [4,5].

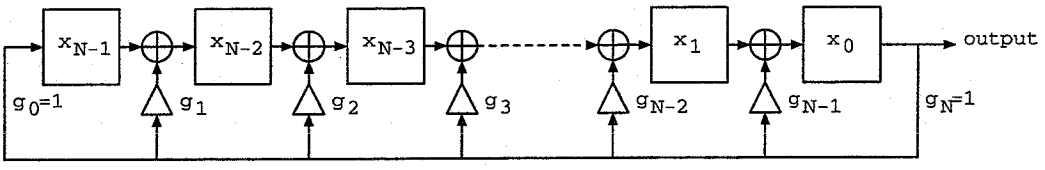


Figure 4.2 Galois LFSR

These two implementations will produce the same PRN sequence if configured with the same weights and the appropriate, but different, initial states. Because of this, an LFSR is usually identified by only its weights, as a binary string or in the form of a generator polynomial:

$$G(x) = g_n X^n + g_{n-1} X^{n-1} + g_{n-2} X^{n-2} + \ldots + g_2 X^2 + g_1 X + g_0 \tag{4.2}$$

LFSRs have a firm mathematical basis and one of their properties is that if $G(x)$ is primitive, the produced sequence will have a cycle length of $2^n - 1$, which is called an M-

sequence, or Maximum Length Sequence. In CDMA we will mainly be looking at M-sequences. Also note that a sequence generated by an n-bit LFSR will never contain a so-called "run" of n or more zero's in succession. This fact is most easily verified by looking at a Fibonacci LFSR. Because the content of a Fibonacci LFSR equals the next n bits in its output sequence, a run of n zero's implies that the register is filled with all zeros. However, it is easily verified that once the register is filled with all zeros, it can never produce even a single one anymore. There is one exception to this rule and that is when the register is initially filled with zeros, in which case the register produces only zeros. For a more thorough treatment of LFSRs, M-sequences and their properties we refer to the standard works on these subjects [7,8].

Different implementations of an LFSR can be compared on several different criteria. First of all there are two criteria that are important to any hardware design: speed and area.

As for speed, or more precisely, cycle time: the combinatorial logic in an LFSR design consists of mostly AND-gates (which implement binary multiplication) and XOR-gates (which implement binary addition). Therefore, we define the cycle time of a design to be the number of 2-input AND-gate delays measured in D_{and} and the number of 2-input XOR-gate delays, measured in D_{xor}. An AND-gate delay is not the same as a XOR-gate delay, so a real comparison can only be made when we assign weights to these two measures. We will do so at the end of paragraph 4.2, when we draw some conclusions. For area we will use the same metric as for speed; we count the number 2-input AND gates and the number of 2 input XOR-gates, measured in A_{and} and A_{xor} respectively.

There is a third criterion that is important in a hardware design: power consumption. This criterion is quite difficult to determine at a high level and a good estimate requires careful simulation. For this reason we will mention power consumption only if there is especially striking opportunity to gain an advantage in this area.

A criterion that we *will* consider is the number of configuration bits required to configure the design. Normally we only have the N configuration bits from the generator polynomial, but we will see an approach that gains an area and speed advantage at the cost of extra configuration bits.

4.2 Fibonacci LFSR Vectorization

As said before, the CGU is designed to operate on vectors of arguments. For this reason the CGU has to generate a vector full of chips every clock cycle. In the remainder of this document we will call the vector width W and the LFSR length N. An LFSR of length N generating W chips (values) every clock cycle will be called an N/W-LFSR.

There are two aspects to generating code words in vectors. First, every clock cycle, W chips have to be derived from the state of the LFSR. Second, every clock cycle, the state has to be advanced by W chips. We will look at both of these aspects.

For a Fibonacci LFSR, the first aspect is almost trivial. At any time, the state of a Fibonacci LFSR consists of the next N output chips. So, as long as $W \leq N$, we can just take the next W output chips straight from the state of the LFSR. The UMTS standard also uses sequences that are generated with an LFSR that is shorter than the preferred vector width of 16 chips, so $W > N$. This could present a problem, because in such a case we would like to output more chips than have actually been generated. However, we will deal with this problem and for now we assume that W is always smaller than N (which will turn out to be the case). So, in the case of a Fibonacci LFSR, the output, which we call Z, equals (for now) $[x_0, \ldots, x_{P-1}]^T$.

In the following we look at several methods to advance the state of the register by W. We compare these methods with the criterion introduced in the previous paragraph.

4.2.1 First solution: Direct implementation

The first method that we look at is the method used by Tom Geelen in his work on the CGU [4,5]. Tom Geelen used the combinatorial logic of multiple (W) cascaded LFSRs to advance the state of one LFSR by W (Figure 4.3).

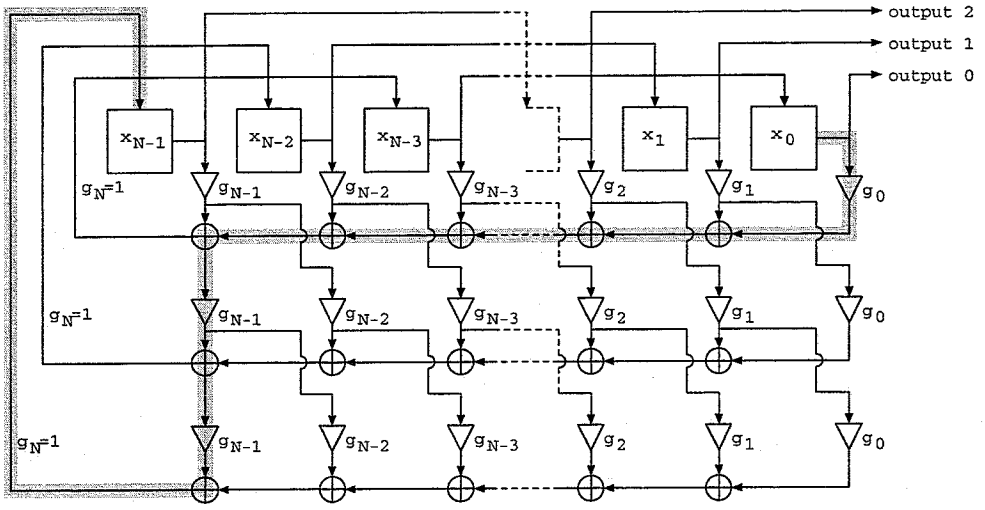


Figure 4.3 Direct implementation of a Fibonacci LFSR with step-size 3.

The main disadvantage of this solution is the cycle time of the resulting circuit. We highlighted two additions in the figure: one horizontal and one vertical. The values for the horizontal addition are all immediately available, so this addition 'tree' can be rebalanced to get a logarithmic (in N) evaluation-time. However, the vertical addition requires values that are not available yet and this addition has to be evaluated linearly. Because of this the cycle time of the design increases linearly with the step-size W. More precisely, the cycle time of this design equals: $(\lceil^2\log N\rceil + W - 1)\, D_{xor} + (W)\, D_{and}$.

The size of the design is easy to determine; it is simple equal to W times the size of a standard LFSR: $(W * N)\, A_{and} + (W * (N-1))\, A_{xor}$. Finally, no extra bits are required to configure the design, only the N bits from the generator polynomial.

4.2.2 Second solution: Offline F^W

As the start of our second approach, we first create a mathematical model of an LFSR.

The state of the register at timestamp t is denoted by the vector $X(t)$.

$$X(t) = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-2} \\ x_{N-1} \end{pmatrix} (t) \tag{4.3}$$

The values $g_0 \ldots g_{n-1}$ determine the feedback pattern of the LFSR and they depend on the generator polynomial. (g_0 is usually equal to 1.) The elements of $X(t+1)$ are all a linear combination of the elements in $X(t)$, therefore, mathematically, shifting the register is equivalent to multiplying the state with a matrix that depends on $g_0 \ldots g_{n-1}$. This matrix is called the characteristic matrix of the LFSR.

$$F = \begin{pmatrix} & 1 & & \\ & & \ddots & \\ & & & 1 \\ g_0 & g_1 & \cdots & g_{N-1} \end{pmatrix} \tag{4.4}$$

Because of the sparse nature of the matrices we will be working with, 0-elements are omitted.

With these definitions, making a single step with the LFSR can be expressed as:

$$X(t+1) = F \cdot X(t) \tag{4.5}$$

We can generalize expression (4.5) to advance $X(t)$ by W steps:

$$X(t+W) = F \cdot F \cdot \ldots \cdot F \cdot X(t) \tag{4.6}$$

And finally, because matrix multiplication is associative, we also have:

$$X(t+W) = F^W \cdot X(t) \tag{4.7}$$

A symbolic evaluation of F^W in expression (4.7) results in a matrix in which the last W rows contain expressions that depend on $g_0 \ldots g_{n-1}$. As a second solution to the problem of advancing $X(t)$ by W steps we could choose to introduce a configuration bit for each of these expressions, giving us a matrix filled with $N*W$ configuration bits. Advancing the state of the register is now a simple matter of multiplying $X(t)$ by a scalar matrix. To find the required configuration bits, we need to calculate the expressions offline for given $g_0..g_{N-1}$. This would result in a very fast solution, because we only have to hardwire a matrix-vector multiplication, which can be implemented as the parallel evaluation of dot-products. A dot product only takes 1 $D_{and} + \lceil {}^2\log N \rceil D_{xor}$, which is a lot less then the previous solution. The solution requires the same amount of hardware; one dot product takes $(N)\ A_{and} + (N-1)\ A_{xor}$, so W dot products in parallel takes $(W*N)\ A_{and} + (W*(N-1))\ A_{xor}$.
The problem with this solution is twofold: First of all, the solution is less flexible; on-the-fly re-configuring is only possible for a known set of $g_0..g_{N-1}$, for which we can store the configuration bits in a memory. This might not be a problem in practice for current standards, but it might be for future standards. Secondly, the solution requires far more configuration bits than the standard N from the generator polynomial. The required number of bits in this case is $W*N$, making it scale linearly with W, instead of being constant with respect to step-size. These extra bits increase the memory requirement and possibly the time required to configure the design, because retrieving that many bits from a memory probably requires several steps.

4.2.3 Third solution: Online F^W

In order to limit the configuration bits to just $g_0 \ldots g_{n-1}$ we have to evaluate the expressions in F^W on the fly. The symbolic evaluation of F^W for even a relatively small 8/8-LFSR shows that the expressions in F^W grow to enormous length as W increases. To illustrate this we look at one column from F^W, for a LFSR of length 8, as W increases:

$$F^1[4] = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ g_4 \end{pmatrix}$$

$$F^2[4] = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ g_4 \\ g_3 + g_4\, g_7 \end{pmatrix}$$

$$F^4[4] = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ g_4 \\ g_3 + g_4\, g_7 \\ g_2 + g_4\, g_6 + g_3\, g_7 + g_4\, g_7 \\ g_1 + g_4\, g_5 + g_3\, g_6 + g_2\, g_7 + g_3\, g_7 + g_4\, g_7 \end{pmatrix}$$

$$F^8[4] = \begin{pmatrix} g_4 \\ g_3 + g_4\, g_7 \\ g_2 + g_4\, g_6 + g_3\, g_7 + g_4\, g_7 \\ g_1 + g_4\, g_5 + g_3\, g_6 + g_2\, g_7 + g_3\, g_7 + g_4\, g_7 \\ g_0 + g_4 + g_3\, g_5 + g_2\, g_6 + g_4\, g_6 + g_1\, g_7 + g_2\, g_7 + g_3\, g_7 + g_4\, g_7 + g_4\, g_6\, g_7 \\ g_2\, g_5 + g_1\, g_6 + g_3\, g_6 + g_0\, g_7 + g_1\, g_7 + g_2\, g_7 + g_3\, g_7 + g_4\, g_7 + g_4\, g_5\, g_7 + g_3\, g_6\, g_7 + g_4\, g_6\, g_7 \\ g_3 + g_1\, g_5 + g_4\, g_5 + g_0\, g_6 + g_2\, g_6 + g_4\, g_6 + g_0\, g_7 + g_1\, g_7 + g_2\, g_7 + g_3\, g_7 + g_3\, g_5\, g_7 + g_2\, g_6\, g_7 + g_3\, g_6\, g_7 + g_4\, g_6\, g_7 \\ g_0\, g_5 + g_3\, g_5 + g_1\, g_6 + g_3\, g_6 + g_4\, g_5\, g_6 + g_0\, g_7 + g_1\, g_7 + g_2\, g_7 + g_3\, g_7 + g_4\, g_7 + g_2\, g_5\, g_7 + g_1\, g_6\, g_7 + g_2\, g_6\, g_7 + g_3\, g_6\, g_7 \end{pmatrix}$$

This clearly shows that the size of the expressions in F^W grows super linear as W increases. In fact, further practical tests suggest that the size of the expressions in F^W is $O(W^2)$. For UMTS we are looking at 32/16-LFSRs and in large instances like that the size and shear number of these expressions becomes a bottleneck in hardware size. If we assume that the average length of the expressions in F^W grows quadratically with W then a rough estimate for the hardware required for F^W is $O(N*W^3))\ A_{xor} + O(N*W^3))\ A_{and}$. Similarly, an estimate for the latency of this solution is $O(^2\log\ (N*W^2))\ D_{xor} + O(1)\ D_{and}$, which is based on the knowledge that the expressions in F^W can be evaluated in parallel. Finally, we only need $g_0..g_{N-1}$ to configure a design based on this solution.

4.2.4 Fourth solution: Factorized F^W

To reduce the hardware requirements of the previous solution we will try to transform the expressions in F^W into a functionally equivalent form that requires much less hardware and evaluates just as fast. In particular, we would like to identify sub-expressions that occur numerous times in $F^W X(t)$. To this end we look at an 8/8-LFSR. The operation of an 8/8-LFSR is modeled by:

$$X(t+8) = F \cdot F \cdot F \cdot F \cdot F \cdot F \cdot F \cdot F \cdot X(x) \tag{4.8}$$

We can add some parentheses to force a right-to-left evaluation:

$$X(t+8) = F \cdot (F \cdot (F \cdot (F \cdot (F \cdot (F \cdot (F \cdot X(t)))))))) \tag{4.9}$$

We will evaluate expression (4.9) and try to find a pattern as we go along. Starting with:

$$X(t) = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix}$$

we multiply with F and get:

$$X(t+1) = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ g_0 x_0 + g_1 x_1 + g_2 x_2 + g_3 x_3 + g_4 x_4 + g_5 x_5 + g_6 x_6 + g_7 x_7 \end{pmatrix}$$

If we compare $X(t+1)$ with $X(t)$ it is clear that, as expected in a shift register, the elements in $X(t)$ shifted up one location and a new expression is inserted at the bottom. Since the same thing will happen when we multiply $X(t+1)$ by F we substitute the bottom element in $X(t+1)$ with identifier s_1. We multiply the resulting state by F again and get:

$$X(t+2) = \begin{pmatrix} x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ s_1 \\ g_7 s_1 + g_0 x_1 + g_1 x_2 + g_2 x_3 + g_3 x_4 + g_4 x_5 + g_5 x_6 + g_6 x_7 \end{pmatrix}$$

This time, instead of replacing the entire last element by an identifier, we choose to only replace the part that is similar in form to s_1. The main reason for this is that replacing the entire first element by s_2 would result in a nested set of identifiers, which would force a more sequential evaluation scheme, i.e. a less balanced addition tree, increasing the evaluation latency. Multiplying the resulting state with F two more times and performing substitutions were appropriate, we get:

$$X(t+4) = \begin{pmatrix} x_4 \\ x_5 \\ x_6 \\ x_7 \\ s_1 \\ g_7 s_1 + s_2 \\ (g_6 + g_7) s_1 + g_7 s_2 + s_3 \\ (g_5 + g_7) s_1 + (g_6 + g_7) s_2 + g_7 s_3 + s_4 \end{pmatrix}$$

The bottom elements in this vector are all a linear combination of s-terms, with repeating coefficients so we can rewrite the above expression into:

$$X(t+4) = \begin{pmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & & & & 1 \\ & & & & & & 1 & g_7 \\ & & & & & 1 & g_7 & g_6 + g_7 \\ & & & & 1 & g_7 & g_6 + g_7 & g_5 + g_7 \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \\ x_6 \\ x_7 \\ s_4 \\ s_3 \\ s_2 \\ s_1 \end{pmatrix}$$

And because the identifiers s_i we introduced are a linear combination of the original state bits and the generator polynomial, we can rewrite this into:

$$X(t+4) = \begin{pmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & & & & 1 \\ & & & & & & 1 & g_7 \\ & & & & & 1 & g_7 & g_6 + g_7 \\ & & & & 1 & g_7 & g_6 + g_7 & g_5 + g_7 \end{pmatrix} \begin{pmatrix} & & & & 1 & & & \\ & & & & & 1 & & \\ & & & & & & 1 & \\ & & & & & & & 1 \\ & & & & g_0 & g_1 & g_2 & g_3 & g_4 \\ & & & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 \\ & & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & g_6 \\ & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & g_6 & g_7 \end{pmatrix} X(t)$$

In a similar fashion we can rewrite *X(t+5)* into:

$$X(t+5) = \begin{pmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & 1 & & & & & \\ & & & & & & & p_0 \\ & & & & & & p_0 & p_1 \\ & & & & & p_0 & p_1 & p_2 \\ & & & & p_0 & p_1 & p_2 & p_3 \\ & & & p_0 & p_1 & p_2 & p_3 & p_4 \end{pmatrix} \begin{pmatrix} & & & & 1 & & & \\ & & & & & & 1 & \\ & & & & & & & 1 \\ & & & & g_0 & g_1 & g_2 & g_3 \\ & & & g_0 & g_1 & g_2 & g_3 & g_4 \\ & & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 \\ & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & g_6 \\ g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & g_6 & g_7 \end{pmatrix} X(t)$$

where we introduced another set of identifiers p_i, to simplify the result even more.

The simple structure of the two matrices we introduced triggered us to propose the following theorem:

Theorem 1

$$F^W = P_W\, G_W \qquad , W \le N$$

where for given W and N

$$\begin{array}{lcll} P_{i,j} & = & \text{if} & \\ & & [] \quad (i=j) \wedge (i < N-W) & \Rightarrow \quad 1 \\ & & [] \quad (i+j \ge 2N-W-1) & \Rightarrow \quad p_{i+j-2N+W+1} \\ & & [] \quad \text{else} & \Rightarrow \quad 0 \\ & & \text{fi} & \end{array}$$

$$\begin{array}{lcll} G_{i,j} & = & \text{if} & \\ & & [] \quad (j-i=W) & \Rightarrow \quad 1 \\ & & [] \quad (i+j \ge N-1) \wedge (i \ge N-W) & \Rightarrow \quad g_{i+j-N+1} \\ & & [] \quad \text{else} & \Rightarrow \quad 0 \\ & & \text{fi} & \end{array}$$

$$\begin{aligned} p_0 &= 1 \\ p_i &= \sum_{n=0}^{i-1} p_n\, g_{N-i+n} \end{aligned}$$

or, depicted schematically

$$P_W = \begin{pmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & \\ & & & p_0 \\ & & \iddots & \vdots \\ & & p_0 \cdots & p_{W-1} \end{pmatrix} \qquad G_W = \begin{pmatrix} & & 1 & & \\ & & & \ddots & \\ & & & & 1 \\ & & g_0 & g_1 \cdots & g_{N-W} \\ & \iddots & \vdots & \vdots & \vdots \\ g_0 & \cdots & g_{W-1} & g_W \cdots & g_{N-1} \end{pmatrix}$$

Proof of theorem 1

We prove the theorem by induction.

Base case: $W = 1$

$$F^1 = P_1\, G_1$$

Substitution of $W = 1$ in the definitions of P_W and G_W shows that P_1 is the identity matrix and G_1 equals F, so for the base case the theorem holds.

Induction step: $F^{W+1} = P_{W+1}\, G_{W+1}$

Using the induction hypothesis $F^W = P_W\, G_W$ we try to prove $F^{W+1} = P_{W+1}\, G_{W+1}$.

F^{W+1}
$\equiv$ {calculus}
$F F^W$
$\equiv$ {induction hypothesis}
$F P_W\, G_W$
$\equiv$ {lemma 4.1 : $F P_W = Q_{W+1}$}
$Q_{W+1}\, G_W$
$\equiv$ {lemma 4.2 : $Q_{W+1} = P_{W+1} K_W$}
$P_{W+1} K_W\, G_W$
$\equiv$ {lemma 4.3 : $K_W\, G_W = G_{W+1}$}
$P_{W+1}\, G_{W+1}$

where for given W and N

$$Q_W = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & & & & p_0 \\ & & & & & \iddots & \vdots \\ & & & & & p_0 \cdots & p_{W-2} \\ g_0 & g_1 \cdots & g_{N-W} & p_1 & \cdots & & p_{W-1} \end{pmatrix} \qquad K_W = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ g_0 & g_1 \cdots & g_{N-W-1} & & \\ & & & 1 & \\ & & & & \ddots \\ & & & & 1 \end{pmatrix}$$

For the proof of lemma's 4.1, 4.2 and 4.3 please see the appendix.

So, what was gained with these transformations? Both the speed and hardware size of this solution are harder to quantify then in the previous solution because of the p_i-factors. These p_i-factors are generated by a simple recurrence relation, but because we are working in GF(2), a lot of optimizations can be applied to the expressions afterwards, which makes it hard to find closed formulas for there latency and size. Below we have included the first 8 optimized p_i-factors.

$p_0 = 1$
$p_1 = g_{15}$
$p_2 = g_{14} + g_{15}$
$p_3 = g_{13} + g_{15}$
$p_4 = g_{12} + g_{14} + g_{15} + g_{14}\, g_{15}$
$p_5 = g_{11} + g_{15} + g_{13}\, g_{15} + g_{14}\, g_{15}$
$p_6 = g_{10} + g_{13} + g_{14} + g_{15} + g_{12}\, g_{15} + g_{14}\, g_{15}$
$p_7 = g_9 + g_{13}\, g_{14} + g_{15} + g_{11}\, g_{15}$

Compared to the expressions in F^8 we showed earlier, these expressions are quite simple. These first factors grow approximately linear in W so, if fully balanced, they evaluate in approximately $\lceil {}^2\log W \rceil\, D_{xor}$ + O(1) D_{and}. The multiplication $G_W X(t)$ takes $\lceil {}^2\log N \rceil\, D_{xor}$ + 1 D_{and}, so in practical cases the p_i-factors can be evaluated in parallel with this first multiplication, which means we can disregard them in our latency evaluation. After the first matrix-multiplication, the multiplication with P_W takes another $\lceil {}^2\log W \rceil\, D_{xor}$ + 1 D_{and}, which puts the total for this solution at $\lceil {}^2\log N \rceil + \lceil {}^2\log W \rceil\, D_{xor}$ + 2 D_{and}, which is approximately $\lceil {}^2\log N{*}W \rceil\, D_{xor}$ + 2 D_{and}.

The size of this solution is more difficult to specify, because we can't disregard the p_i expressions here. The hardware required for the two matrix-multiplications can be easily derived from the definitions of the matrices.

Multiplication with P_W requires

$$\sum_{i=1}^{W-1} i = W\left(\frac{1}{2}W - \frac{1}{2}\right) \quad A_{and} \text{ and}$$

$$\sum_{i=0}^{W-2} i = W\left(\frac{1}{2}W - 1\frac{1}{2}\right) + 1 \quad A_{xor}$$

Multiplication with G_W requires:

$$\sum_{i=N-W+1}^{N} i = W\left(N - \frac{1}{2}W + \frac{1}{2}\right) \quad A_{and} \text{ and}$$

$$\sum_{i=N-W}^{N-1} i = W\left(N - \frac{1}{2}W - \frac{1}{2}\right) \quad A_{xor}.$$

This gives a total of $(W*N)$ A_{and} + $(W*(N-2))$ A_{xor}, exclusive the hardware in the p_i expressions. If we assume again that for practical cases p_i expressions grow linearly with W, this adds approximately $\frac{1}{2}W^2$ A_{xor} and A_{and} to the hardware requirements.

The number of configuration bits required for this solution equals N.

Finally, this solution has a third advantage that needs mentioning, namely in the area of power consumption. The p_i-factors that we introduced depend only on the generator polynomial and because this polynomial normally doesn't change during sequence generation, the values of the p_i-factors don't change either. This means that p_i-factors only need to be evaluated during the initial configuration of the generator and can then be clock-gated afterwards to save energy. This observation also reinforces our decision to exclude p_i-factors from the latency of the solution.

The final result of our transformations is a nice, concise and precise mathematical description of the hardware in an N/W-LFSR, which is both easy to instantiate for arbitrary N and W and easy to implement. A high-level schematic of this design is shown in Figure 4.4.

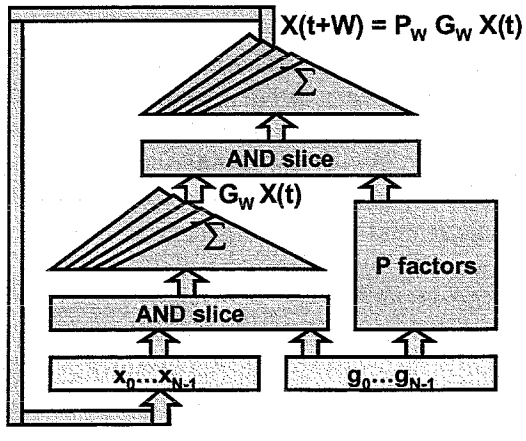


Figure 4.4. Factorized F^W schematic.

The schematic shows how the p_i expressions only depend on the generator polynomial. The rest of the circuit consists of parallel, balanced addition trees preceded by a vector AND-operation that implement vector dot-products, which in turn implement matrix multiplication.

4.2.5 p_i expressions

In the previous paragraph we introduced the p_i expressions that were defined by

$$p_0 = 1$$
$$p_i = \sum_{n=0}^{i-1} p_n g_{N-i+n}$$

The summation in the definition might suggest that the size of these expressions will grow exponentially as i increases. However, experimental data suggests a linear relation, at least for small (practical) i. Figure 4.5 shows a graphical representation of the number of additions in the p_i expressions, when calculated in GF(2), GF(4) and the field of integers. As a reference we also added $(1.5)^i$ to the graph.

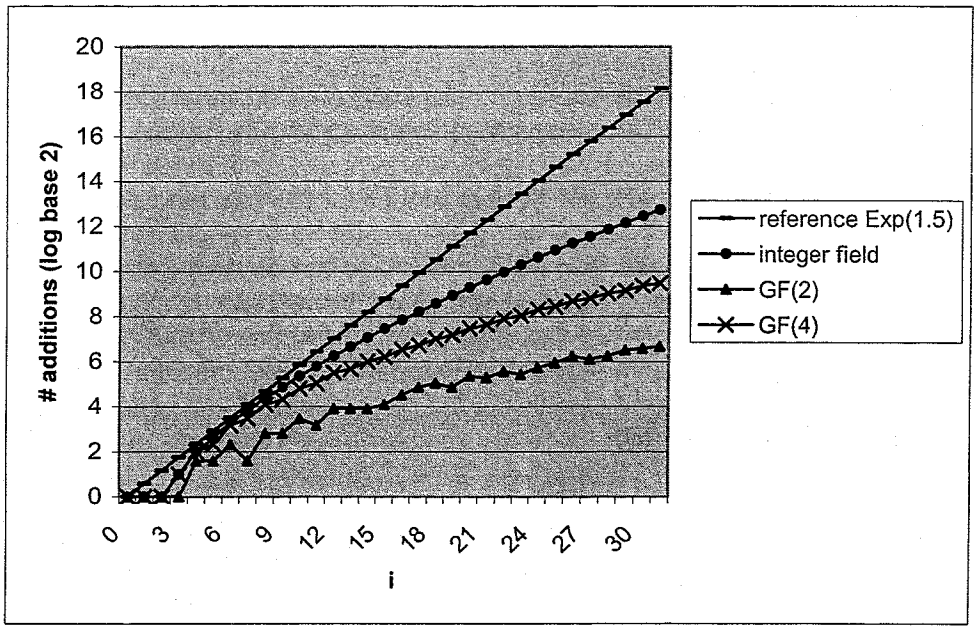


Figure 4.5 Number of additions in p_i expressions

It is difficult to conclude whether or not the size of the expressions grows exponentially, based on such limited data. What we can say is that if it grows exponentially, it does so with a very small base. In the integer field, where very few optimizations are possible, the size grows exponentially in the beginning, but then seems to start flattening out. In both GF(2) and GF(4) the size grows much slower than the reference $(1.5)^i$. This is because several very systematic optimizations can be performed on the expressions in these cases. Specifically, in GF(2), sub-expressions with an even coefficient can be removed completely, uneven coefficients are taken modulo 2 and exponents can be discarded. These optimizations are also the reason that the size of the expressions is not strictly increasing; some expressions just have more opportunity for optimization than others. There seems to be a pattern in when these decreases occur, but we don't have an explanation for this behavior.

In GF(4) there are similar types of optimizations. For example, sub-expressions with a coefficient that is a multiple of 4 can be discarded. Since completely removing a sub-expression is a very effective optimization, it is clear that binary LFSRs profit the most from these optimizations, at least in absolute terms.

If we look at the size of the expressions in GF(2) again, our assumption that it increases linearly in the beginning seems to be correct; the first 16 expressions range in size from 1

to 18 terms. After that the size starts to grow more quickly, perhaps quadratically, as the size ranges from 24 to 102 in the next 16 expressions. Still, these expressions aren't extremely complex and their latency grows slowly, so an LFSR with step-size 32 is not unthinkable.

As a final remark we can say that the number of multiplications in the p_i expressions grows approximately as fast the number of additions. However, multiplications are implemented with and-gates, which are a lot smaller than the XOR-gates that implement addition. So, the number of additions dominates the amount of hardware required to implement the expressions. Also, because all terms in a p_i expression can be evaluated in parallel, the evaluation latency of an expression is also mostly determined by the number of additions.

4.2.6 Conclusion

We have seen four different approaches to advance the state of the LFSR by W bits. But only by factorizing F^W, were we able to designed a solution that scores well on all three important criteria. The following table gives an overview of the mentioned solutions and their score on the introduced criteria.

	Costs	Latency	Config size	Remarks
Direct	$(W*N)$ A_{and} $W*(N-1)$ A_{xor}	(W) D_{and} $\lceil {}^2\log N \rceil + W - 1$ D_{xor}	N bits	
Offline F^W	$(W*N)$ A_{and} $W*(N-1)$ A_{xor}	1 D_{and} $\lceil {}^2\log N \rceil$ D_{xor}	$N*W$ bits	• high memory requirement • less flexibility
Online F^W	$O(N*W^3)$ A_{and} $O(N*W^3)$ A_{xor}	$O(1)$ D_{and} $O({}^2\log (N*W^2))$ D_{xor}	N bits	• horrible to implement
Factorized F^W	$\sim W*(N+\frac{1}{2}W)$ A_{and} $\sim W*(N+\frac{1}{2}W-2)$ A_{xor}	2 D_{and} $\sim\lceil {}^2\log N*W \rceil$ D_{xor}	N bits	• vhdl: ~2.6 ns • clock gating opportunity

The last solution is by far the best. It is only marginally more expensive than the cheapest solution, while at the same time being one of the fastest solutions. Determining a good indication of exactly how fast a design is, is best left to a synthesis tool. Assigning weights to the delay of different types of gates is difficult because these values depend greatly on factors such as the drive-strength on the inputs and the required drive-strength on the output. Ballpark figures for a D_{xor} and a D_{and} are 0.30 ns and 0.20 ns respectively. Such ballpark figures for the size of these two gates are 40 μ^2 and 20 μ^2 for a A_{xor} and a A_{and} respectively. With these figures we can calculate another table, with practical data for a 32/16-LFSR.

	Costs (μ^2)	Latency (ns)
Direct	30080	9.2
Offline F^W	30080	1.7
Online F^W	~125000000	4.1 (1.7)
Factorized F^W	37120	3.1

The "Offline F^W" approach still looks very good. However, it requires so many configuration bits that the design just isn't practical anymore. The fact that all these huge expressions need to be calculated offline isn't very intuitive and it is just a plain ugly solution.

4.3 Galois LFSR Vectorization

In this paragraph we will look at both aspects of vector generation of chips for a Galois LFSR. Unlike with a Fibonacci LFSR, the first aspect, deriving a vector of W chips from the current state of the register is not trivial. Only the very last element in the state occurs in the output sequence unaltered, all other elements change as they shift through the register. Because at every step a state element is only influenced by the elements with a lower index (i.e. closer to the output), we can derive the next W outputs from the last W state bits. We calculated the expressions that transform state bits into output bits and were able to express this transformation in a matrix-vector product again:

$$\begin{pmatrix} Z_0 \\ Z_1 \\ \vdots \\ Z_{W-2} \\ Z_{W-1} \end{pmatrix} = \begin{pmatrix} p_0 & & & & \\ p_1 & p_0 & & & \\ \vdots & \vdots & \ddots & & \\ p_{W-2} & p_{W-3} & \cdots & p_0 & \\ p_{W-1} & p_{W-2} & \cdots & p_1 & p_0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{W-2} \\ x_{W-1} \end{pmatrix}$$

That is surprising! The matrix is almost exactly P_W from the previous paragraph and with a small rewrite we find:

$$\begin{pmatrix} Z_0 \\ Z_1 \\ \vdots \\ Z_{W-2} \\ Z_{W-1} \end{pmatrix} = P_W \begin{pmatrix} x_{W-1} \\ x_{W-2} \\ \vdots \\ x_1 \\ x_0 \end{pmatrix}$$

So, we found a nice, clean way to transform the state bits into output chips. However, the fact that we need to do so might already indicate that Galois LFSRs are less well suited for vector generation of a sequence than Fibonacci LFSRs.

To advance the state of the register by W, we investigate how the solutions for a Fibonacci LFSR translate to a Galois LFSR.

In the first solution we cascaded to logic of several Fibonacci LFSRs. This direct implementation has an equivalent in the case of a Galois LFSR, which is depicted in Figure 4.5.

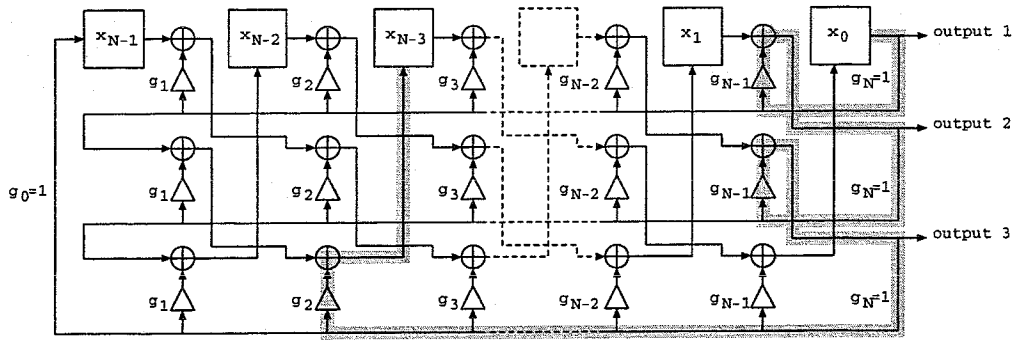


Figure 4.5 Direct implementation of a 3-step Galois LFSR.

A one-step Galois LFSR has a lower latency then a one-step Fibonacci LFSR and in a direct implementation this advantage translates to a multi-step LFSR. We highlighted the critical path of the circuit and it is clear that going to a multi-step LFSR adds approximately the same latency to a Galois LFSR as it does to a Fibonacci LFSR. This means that the absolute advantage of a Galois LFSR remains, but the relative advantage decreases. The latency of this direct implementation of a multi-step Galois LFSR is (W)

D_{xor} + (W) D_{and}. The size of the LFSR equals that of a Fibonacci LFSR: $W*(N-1)$ A_{xor} + $W*(N-1)$ A_{and}. Configuring the design requires only N bits.

A Galois equivalent for the other three solutions from the previous paragraph requires the characteristic matrix F. In case of a Galois LFSR this matrix has the following form:

$$F = \begin{pmatrix} g_{N-1} & 1 & & \\ g_{N-2} & & \ddots & \\ \vdots & & & 1 \\ g_0 & & & \end{pmatrix} \tag{4.10}$$

Advancing the state by one is again expressed by expression 4.5 from the previous paragraph:

$$X(t+1) = F \cdot X(t)$$

Expression 4.7 again expresses advancing the state by multiple steps.

$$X(t+W) = F^W \cdot X(t)$$

With these definitions there are Galois equivalents for the second and third solutions from the previous paragraph, which differ only in latency from these solutions.
A direct evaluation of F^W in case of a Galois LFSR results in a matrix in which in every row the first W elements are expressions that depend on $g_0 \ldots g_{n-1}$. Multiplying $X(t)$ with this matrix results in the parallel evaluation of N expressions each having a time complexity of O(log W). In case of a Fibonacci LFSR multiplying $X(t)$ with F^W results in the parallel evaluation of W expressions each having a time complexity of O(log N). So, assuming $W < N$, a Galois LFSR still has the potential of being faster than a Fibonacci LFSR, but this advantage theoretically decreases as W increases. As was the case with a Fibonacci LFSR, an ad hoc evaluation of F^W results in huge expressions that would require too much hardware to hardwire.

We haven't been able to find a factorization of F^W equivalent to the one we found in case of a Fibonacci LFSR. There is however a different factorization that wasn't possible for a Fibonacci LFSR because of latency issues. Because the latency of multiplying $X(t)$ with F^W in the Galois case depends on W instead of N (in the Fibonacci case), we can replace $F^W X(t)$ by $F^{½W} F^{½W} X(t)$, without immediately doubling the latency of the evaluation. This transformation has a very positive effect on the hardware requirements because the expressions in $F^{½W}$ are a lot smaller than in F^W and they only have to be implemented once. Furthermore, the hardware required for the two smaller multiplications is actually a bit less then what is required for the single big multiplication.

The latency of the operation is affected in conflicting ways by this transformation. On the one hand, the expressions in $F^{½W}$ are a lot smaller and thus evaluate faster than in F^W. On the other hand, the latency of the actual multiplications rises from 1 D_{and} + $\lceil^2\log W\rceil$ D_{xor} to 2 D_{and} + 2*$\lceil^2\log ½W\rceil$ D_{xor}. How these two effects interact is not exactly clear. We will assume that splitting F^W increases the latency of the operation, because we are forcing a specific evaluation order that might not be ideal. If the latency increases, then the splitting-transformation is a tradeoff between size and speed, something that is not uncommon in hardware designs. For the CGU, the optimum value in this tradeoff is where the CGU is not a latency bottleneck for the CVP and requires as little hardware as possible. We didn't investigate this tradeoff any further, because the solution doesn't seem to have any advantages over our solution for a Fibonacci LFSR. At the same time it

suffers from several annoying disadvantages, such as difficult instantiation for arbitrary W and N and extremely error prone implementation.

Concluding we can say that we weren't able to find a vectorized solution for a Galois LFSR that is as nice as the one we found for the Fibonacci LFSR. This is unfortunate, because theoretically a Galois LFSR still has a lower latency. In future work it might be a good idea to reinvestigate this issue.

5 PRN Generator

Pseudo Random Noise sequences are sequences of values that, for many intents and purposes, appear completely random, but are deterministically generated and reproducible with an algorithm. Two pseudo random sequences have the property that the similarity between them is very low, and not much higher than what would be statistically expected from two completely random sequences. The similarity between two sequences is also called the cross-correlation. Another property of a pseudo random sequence is that two instances of the same sequence are only similar if there is no phase difference between them, as would be expected from a completely random sequence. The similarity of two instances of the same sequence is also called auto-correlation. The good cross- and auto-correlation properties of PRN sequences are what make them well suited for DS-CDMA, because they make it possible to extract a specific user's signal from the received signal, while ignoring the rest.
The LFSR that we designed in the previous chapter forms the basis of our PRN generator. In this chapter we will enhance the functionality of this design so that it is flexible enough to implement all the requirements from paragraph 3.2.

5.1 Delayed Sequence Generation

It is a well-known property of LFSRs that the value of a linear combination of the register values in an LFSR cycles through a delayed version of the sequence generated by the LFSR. This property can be used to generate in parallel an m-sequence and a cyclically shifted (i.e. delayed) version of the same sequence. A basic Fibonacci LFSR that implements this property is depicted in Figure 5.1.

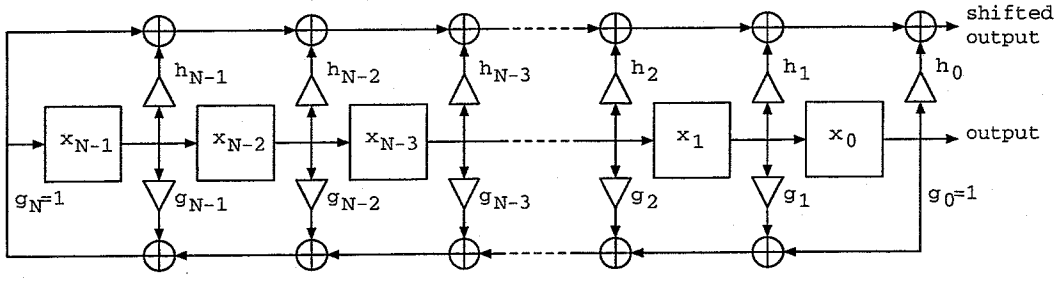


Figure 5.1 Fibonacci LFSR generating a delayed output bit

This extension adds control bits $h_0..h_{N-1}$ that specify exactly how the linear combination of register values should be generated.
The parallel generation of a sequence and a delayed version is used in both UMTS and GPS and no doubt in other applications as well. Again, we need to vectorize this mode of operation, because otherwise generating the delayed output becomes a bottleneck on the throughput of our LFSR.

The register can be seen as a window on the sequence that is being generated. This is especially true in the case of a Fibonacci LFSR where the bits in the register no longer change as they shift through it. The first delayed output bit depends on all the bits in the window and the next delayed output already depends on sequences bits that are outside the window. Generalizing, the next W delayed output bits depend on a window that is W-1 bits longer that the size of our register. We can easily increase the size of our window on the sequence by buffering output bits before we actually output them. We need to buffer W-1 bits on the output side of the LFSR, but we choose to buffer W because this way we can take the entire vector of output bits from the buffer. Adding the buffer can be

seen as increasing the size of our LFSR, but because the buffer doesn't generate feedback for the generation of new bits nothing changes to the output sequence of the LFSR. Every clock cycle, W output bits can be taken from the buffer, W bits can be copied from the original LFSR into the buffer and W new bits are generated with the feedback logic.
With the new buffer we have a window of $N+W$ bits on the sequence and it is now easy to extend the generation of the delayed output to vectors. Figure 5.2 shows a Fibonacci LFSR that generates in parallel W normal and W delayed output bits.

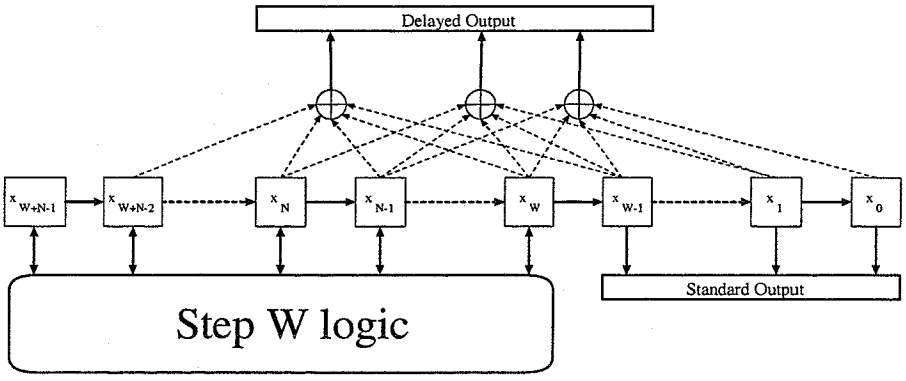


Figure 5.2 Vectorized Fibonacci LFSR for normal and delayed output

Copying W values from the actual register to the buffer can trivially be incorporated in the F^W matrix for both the Fibonacci and Galois LFSR, but in this case these mathematics don't add anything to our understanding of the subject. However, because we will be using the construct in one of the following paragraphs we will introduce a new matrix H_W that mathematically represents the linear combinations that generate the delayed output. To this end we first have to redefine the state of the LFSR to include the new buffer. We will call the state including the buffer $X'(t)$ and it is defined as:

$$X'(t) = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{W-1} \\ x_W \\ \vdots \\ x_{N+W-2} \\ x_{N+W-1} \end{pmatrix}(t) \tag{5.1}$$

where $(x_0 \ldots x_{W-1})^T$ is the buffer and the rest the original state. Now, generating the delayed output bits, which we will call Z', is an operation on $X'(t)$ that can be expressed as a matrix-vector product:

$$Z' = H_W \cdot X'(t) \tag{5.2}$$

where H_W equals:

$$H_W = \begin{pmatrix} h_0 & \cdots & h_{W-1} & \cdots & h_{N-1} & & 0 \\ & \ddots & \vdots & & \vdots & \ddots & \vdots \\ & & h_0 & \cdots & h_{N-W} & \cdots & h_{N-1} \; 0 \end{pmatrix} \tag{5.3}$$

Please note that the last column in this matrix could have been left out completely if we added only the W-1 required buffer bits.

The H matrix is the same for both Fibonacci and Galois LFSRs, but the values of $h_0 \ldots h_{N-1}$ will differ between these two architectures. The hardware cost for implementing delayed output is quite significant; calculating the linear combinations of state bits requires $(N*W)$ A_{and} + $(W*(N-1))$ A_{xor} which is roughly equivalent to the amount of hardware needed to implement basic vectorized LFSR operation. From a cycle time point of view we can safely say that delayed output will not present a bottleneck. The time complexity of calculating $H_W X(t)$ equals approximately 1 $D_{and} + \lceil^2 \log N \rceil D_{xor}$.

5.2 LFSR Resizing

Previously, we decided to assume that the step-size W would always be less than or equal to the length N of our LFSR. However, if we inspect some of the 3G standards, we see that they use LFSRs of length 24 and 18, but also 10 and 8, which is smaller than our intended vector width (step-size) of 16. On the other hand, since we decided to only implement an LFSR of length 32, $N = 32$ regardless of which sequence we are generating and thus $W \leq N$. This does mean that we have to find a way to map the generator polynomials of these different length LFSRs onto our single LFSR of length 32.

In the previous paragraph we added a buffer of register locations to our LFSR, in order to generate delayed output. These extra locations raised the length of our LFSR, but they didn't change the register's output sequence. After adding the 16 buffer locations, we have a length 48 LFSR that produces the output of a length 32 LFSR. Clearly, this is because the extra locations don't participate in the feedback logic. In essence, without looking at it in that way at the time, we did exactly what we are trying to do now; configure a length N LFSR to produce the sequence of a length $M \leq N$ LFSR.
Looking at the previous paragraph, we see that the least significant generator polynomial bits are set to 0 to prevent them from taking part in the feedback. So, to generalize this behavior, we have to map generator polynomials of degree M onto a generator polynomial of degree N with its least significant coefficients set to 0. An easy way to do this, which is also easily verified to be correct, is to multiply the polynomial by its variable X (not to be confused with the state of the register) until it has degree N. The resulting polynomial is no longer primitive and will therefore no longer produce an m-sequence, but that is exactly what we want. An LFSR of length N configured with this polynomial will produce the exact same sequence as a shorter LFSR configured with the original polynomial of degree M.

This is a nice result, but there is still one problem. A shorter LFSR is initialized with a shorter initial state that our LFSR of length 32. We can pad this initial state with, for example zeros, but these extra bits have to be ignored and we can only start producing real output when enough new sequence bits have been generated. A consequence of this is that configuring the LFSR has a long and variable latency, which makes scheduling program code for the CVP more difficult.
We can avoid this latency if we make the location from which we take the output from the register variable. This way we only need one cycle to configure the LFSR and another one to produce the first W outputs, which can then be taken (somewhere) from the register in the next clock cycle. In case of a (32+16)/16-LFSR this means that the location of the first output bit can vary between x_0 and x_{30} and the location of the 16^{th} output bit between x_{15} and x_{45}, in case of a length 32 or length 2 LFSR that is mapped onto the design respectively. This selection of output locations is implemented with a so-called barrel shifter. Because (in case of a Fibonacci LFSR) there was no logic yet between the output stage and the pipeline register that holds the output, we can safely implement this barrel shifter without raising the latency of the design. A barrel shifter that can shift by 30 locations can be configured with only 5 configuration bits. This barrel shifter can again be

seen a function that operates on the adapted state *X'(t)*. The output of the register, previously called *Z* now becomes:

$$Z = O_W \cdot X'(t) \tag{5.4}$$

with

$$O_W = \begin{pmatrix} o_0 & \cdots & o_{W-1} & \cdots & o_{N-2} & & 0 & 0 \\ & \ddots & \vdots & & \vdots & \ddots & \vdots & \vdots \\ & & o_0 & \cdots & o_{N-W-1} & \cdots & o_{N-2} & 0 & 0 \end{pmatrix} \tag{5.5}$$

where exactly one bit out of $o_0..o_{N-2}$ is one and the rest zero. This one-hot configuration can be encoded in the 5 configuration bits we mentioned. Please note that this is just a specification of the output logic, and in this case a matrix-vector multiplication is not the most efficient way to implement this. A barrel shifter uses layers of multiplexers to direct the right value to the right output element. Shifting by a maximum of *M* locations using a barrel shifter requires $\lceil^2 \log M\rceil$ layers of multiplexers.

Also note that it is still possible to generate the delayed output we introduced in the previous paragraph. If we map an LFSR of length $M \leq N$ onto our LFSR of length *N*, and use one cycle to generate *W* new sequence bits, we have a window of *M+W* bits on the sequence and that is exactly what we need to generate a vector of delayed output. The only thing that changes is that for shorter polynomials some of the configuration bits $h_0..h_{N-1}$ are not used and must be set to 0. It is not difficult to see that just like with the generator polynomial, it will be the least significant bits that shouldn't participate. So, if for an LFSR of length *M* we have $h_0...h_{M-1}$, these are mapped onto $h_{N-M}...h_{N-1}$ and $h_0...h_{N-M-1}$ are set to 0.

Now, finally, all the work from the previous paragraphs in this chapter can be combined into one figure (Figure 5.3) that represents the complete design of a *(N+W)/W*-LFSR of the Fibonacci variety. The result is a design that is both fast and flexible and produces normal and delayed output, in vectors, for a whole range of LFSRs of different lengths.

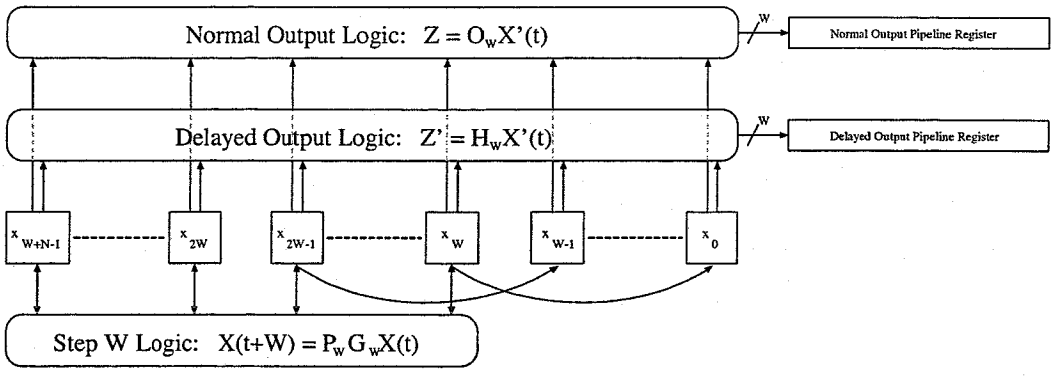


Figure 5.3 Multi-step LFSR with normal and delayed output

5.3 CRC and Signature Analysis

CRC is a technique used to detect errors in transmitted binary data. The sender of a message calculates, from the message, a (much shorter) binary value, which is a rather (but not completely) unique representation of the message. This binary value, or signature as it is also called, is then appended to the original message and the result is sent to the receiver. The receiver splits the message from the signature and calculates a new signature from the message, using the same method as the sender. If the new signature

and the received signature do not match then a transmission error has occurred. Since the message is much longer than the signature, this error has most likely occurred in the message, which consequently has to be resent. If the new signature and received signature match then it can be concluded with a very high degree of probability that the message has been received correctly.
Built-In Self Test (BIST) of hardware uses the same technique, where it is usually referred to as Signature Analysis. In BIST random binary sequences are written to, for example, memories. A signature of the random sequence is compared to the signature of the read-back data and if the two don't match the memory contains faulty bits. The underlying operation performed in both CRC and Signature Analysis is polynomial division, where the message (interpreted as a polynomial) is divided by the generator polynomial and the signature is the remainder of this division. Since a Fibonacci LFSR does not always produce the correct remainder, CRC is usually implemented with Galois LFSRs. A hardware implementation of CRC consists of a LFSR that is adapted such that it consumes an input stream on the side opposite to the output. Figure 5.4 shows a Fibonacci LFSR adapted for Signature Analysis.

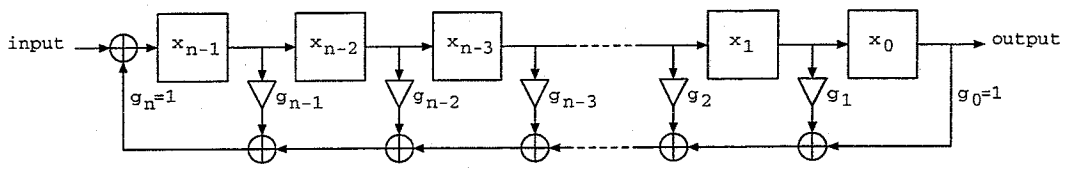


Figure 5.4 Fibonacci LFSR with Signature Analysis

Obviously, this isn't good enough for us; CVP relies on vector processing, and we want our LFSRs to consume a vector full of input bits every clock cycle. We use some more mathematics to determine what is needed to adapt our (Fibonacci) design to accept input vectors.

First we will go back to the basics; paragraph 4.2 showed us a mathematical representation of an LFSR doing a single step. We can easily adapt this representation to include the consumption of one input bit. A vector consisting of all zero's and one input bit is simply added to the new state:

$$X(t+1) = F\,X(t) + Y(0) \tag{5.6}$$

where

$$Y(0) = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ y_0 \end{pmatrix} \tag{5.7}$$

If we repeat this process and keep consuming an input bit every clock cycle we arrive at the following:

$$X(t+W) = F(F\cdots(F\,X(t) + Y(0)) \cdots + Y(W-2)) + Y(W-1) \tag{4.18}$$

which is equal to:

$$X(t+W) = F^W\,X(t) + F^{W-1}\,Y(0) + \cdots + F\,Y(W-2) + Y(W-1) \tag{4.19}$$

Using the results from paragraph 4.2 this formula turned out to reduce very nicely to a much more manageable form, which we capture in theorem 2.

Theorem 2

$$X(t+W) = P_W\left[G_W\, X(t) + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ y_{W-1} \\ \vdots \\ y_0 \end{pmatrix}\right]$$

Proof of Theorem 2

The proof is not very difficult:

$$\begin{aligned}
&F(F\cdots(F\,X(t)+Y(0))\cdots+Y(W-2))+Y(W-1)\\
&\equiv \{\text{math}\}\\
&F^W\,X(t)+F^{W-1}\,Y(0)+\cdots+F\,Y(W-2)+Y(W-1)\\
&\equiv \{Y(j)=(0\mid y_j)^T \text{ therefore, lemma 5.1}: F^i\,Y(j)=P_{i+1}\,Y(j)\}\\
&F^W\,X(t)+P_W\,Y(0)+\cdots+P_2\,Y(W-2)+P_1\,Y(W-1)\\
&\equiv \{\text{lemma 5.2}: P_W\,Y(0)+\cdots+P_1\,Y(W-1)=P_W\cdot(0\mid y_{W-1},\cdots,y_0)^T\}\\
&F^W\,X(t)+P_W\cdot(0\mid y_{W-1},\cdots,y_0)^T\\
&\equiv \{\text{theorem 1}\}\\
&P_W\,G_W\,X(t)+P_W\cdot(0\mid y_{W-1},\cdots,y_0)^T\\
&\equiv \{\text{math}\}\\
&P_W\cdot\left(G_W\,X(t)+(0\mid y_{W-1},\cdots,y_0)^T\right)
\end{aligned}$$

For the proof of lemma's 5.1 and 5.2, please see the appendix.

The result is indeed manageable, as it hardly changes anything to our previous results. All that is added is an unconditional addition of a vector to an intermediate result in the calculation of $X(t+W)$. The hardware needed to implement this is negligible, which gives us vector generation of CRC signatures almost for free! In the previous chapter we showed that it is possible to map any generator polynomial of length $\leq W$ onto our LFSR design. So, when implementing CRC, we can abstract from the actual generator polynomials that are used. All we need is the small change to the design that we derived in this paragraph and a configuration bit that instructs the LFSR to process input from the scalar path.

5.4 Sequential Multi-Step LFSR

Different CDMA standards (and other LFSR applications) usually require different length LFSRs. Our goal of supporting several of these different standards and applications therefore requires that that we have a means of dealing with LFSRs of different lengths.

One observation that we made was that it is possible to map a shorter LFSR onto a longer LFSR. In principle, this observation allows us to support a range of LFSRs that is limited by the length of the longest LFSR that we choose to physically implement. The main drawback of this approach is the amount of hardware that is wasted. The shorter the LFSR that is mapped onto the large LFSR becomes, the more hardware is wasted. Currently, we have to physically implement the longest LFSR that we want to support (or longer) and map all other lengths to this LFSR. For UMTS, this has resulted in a length 32 LFSR that is often used as a length 8 LFSR, in which case a lot of hardware is wasted. CDMA2000 uses a length 42 LFSR; if we were to implement this, we would be wasting hardware practically all the time! This observation led us to explore the

possibilities of using a shorter LFSR to emulate the operation of a longer LFSR. Effectively, we are trying to design a software-LFSR for our specialized hardware. If successful, this approach increases the combinations of LFSRs that can be mapped onto our architecture, while decreasing the waste of hardware in some cases. Also, in such a design, the lengths of LFSRs that we can support is not limited by the length of the longest physical LFSR, but by the number clock cycles that we are willing to spend on emulating such a long LFSR. This last remark is also the drawback of the approach; the sequential steps in this emulation cause a higher latency and thus a lower maximum throughput.

We will show how we can emulate the operation of a $(m*N)/W$-LFSR on an N/W-LFSR, for arbitrary m. Because we are working with the mathematical representation of LFSRs of different lengths, we adapt our notation of P and G matrices to include N, in the following way: $P_{N/W}$ and $G_{N/W}$. If, in the following calculations, the dimensions of a matrix are not clear from their contents, they are defined by their context.

From Theorem 1 we know that the next state $X(t+W)$ of a $(m*N)/W$-LFSR equals:

$$P_{m \cdot N/W}\ G_{m \cdot N/W}\ X(t)$$

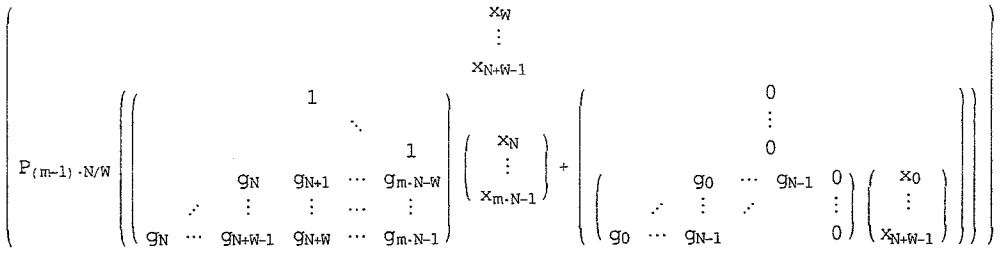
Both $P_{m.N/W}$ and $G_{m.N/W}$ contain a diagonal of one's, performing a shift and copy operation on the state respectively. We can decrease the dimensions of both $P_{m.N/W}$ and $G_{m.N/W}$ by performing a part of this shift-copy operation now, and move the remaining part of the multiplication into the result vector. This results in the following state vector:

$$\begin{pmatrix} \begin{pmatrix} x_W \\ \vdots \\ x_{N+W-1} \end{pmatrix} \\ \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & & p_0 & \\ & & & \vdots & \ddots \\ & & & p_{W-1} & \cdots & p_0 \end{pmatrix} \begin{pmatrix} & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \\ g_0 & \cdots & g_{W-1} & \cdots & g_{N+W} & \cdots & g_{m \cdot N-1} \end{pmatrix} \begin{pmatrix} & & & & \\ & & & & \\ & & & & \\ g_0 & \cdots & g_{N+1} & \cdots & g_{m \cdot N-W} \\ \vdots & \ddots & \vdots & \cdots & \vdots \end{pmatrix} X(t) \end{pmatrix}$$

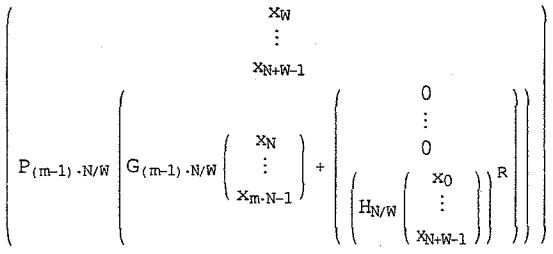
Because both dimensions of the P matrix decreased, the result is still a P matrix. The G matrix on the other hand only changed in height, and the result is no longer a G matrix. Because addition distributes over matrix multiplication, we can split the new matrix into two parts, which gives us an opportunity to regain a proper G matrix:

$$\begin{pmatrix} P_{(m-1) \cdot N/W} \begin{pmatrix} \begin{pmatrix} x_W \\ \vdots \\ x_{N+W-1} \end{pmatrix} \\ \begin{pmatrix} & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \\ 0 & \cdots & 0 & g_N & \cdots & g_{N+W-1} & g_{N+W} & \cdots & g_{m \cdot N-1} \end{pmatrix} \begin{pmatrix} & & & & \\ & & & & \\ & & & & \\ g_N & g_{N+1} & \cdots & g_{m \cdot N-W} \\ \vdots & \vdots & \cdots & \vdots \end{pmatrix} X(t) + \begin{pmatrix} & & & & 0 \\ & & & & \vdots \\ & & & & 0 \\ g_0 & \cdots & g_{N-1} & & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ g_0 & \cdots & g_{N-1} & & 0 \cdots 0 \end{pmatrix} X(t) \end{pmatrix} \end{pmatrix}$$

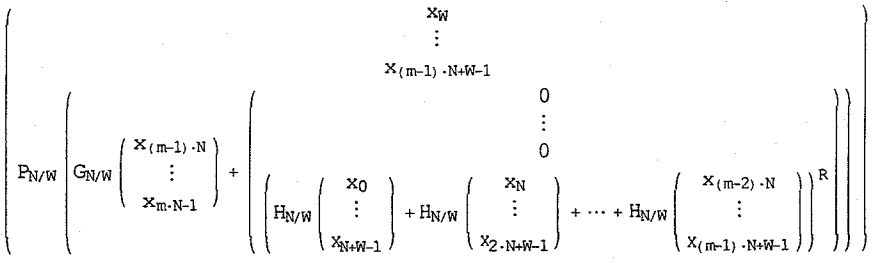
These two new matrices both have columns that contain only zeros. We can remove these columns, if we reduce the size of the state vector that they operate on. The rightmost matrix also has rows that contain only zeros; these rows result in zeros in the result of the multiplication with the state. If we make these zeros in the resulting state explicit, we can remove these rows from the matrix. This results in the following state:



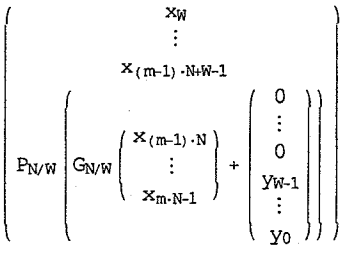
This simplified things a lot and we can clearly recognize a matrix that we have seen before; on the left we have a *G* matrix, with reduced dimensions. On the right we see a matrix that looks a lot like the *H* matrix that we introduced previously for the generation of shifted output bits. The only difference is that all the columns in the matrix have been reversed such that the last row has become the first row and vice versa and the same goes for all other rows. The consequence of this is that we can use the *H* matrix here, but we have to reverse the vector that results after the multiplication. In the rest of this derivation we will use *H* and indicate that the resulting vector has to be reversed with an "^{R}":



This result implements our initial LFSR of length *m.N* with the hardware of a length *(m-1).N* LFSR. We can repeat this step until we arrive (quite trivially) at the following state, which uses only hardware of a length *N* LFSR:



where, of course, the different *H* matrices contain different parts of the original generator polynomial. We can simplify this result a little more, by assuming that we evaluated all the *H* matrix multiplications and added their results, which requires several sequential steps. Also, we will assume that after these additions we reverse the resulting vector. This would result in the following state:



which, surprisingly, is exactly the hardware we derived in the previous paragraph for CRC and SA!

The result of all these transformations is that with some shifting of vectors (i.e. getting the right part of the original state and generator polynomial in the right place at the right time) and some vector additions, which the CVP is good at, we can now emulate (in several sequential steps) an arbitrarily long LFSR on the hardware of a much shorter LFSR.

5.5 Quaternary Codes

One of the main goals of the SW-MODEM project is to support in software a range of next generation mobile application standards, using flexible hardware. This is the opposite of using dedicated hardware for every different standard, which is the strategy employed in most current solutions. Designing flexible hardware that supports everything a standard asks for, or will ask for in the unknown future is not possible, and in some cases a tradeoff has to be made between supporting a feature and keeping the design as flexible as possible. In the mapping of the UMTS standard to our design we encountered such a situation. UMTS uses a quaternary (four-valued) code in an intermediate step to produce the so-called "short codes". This quaternary code is produced with a quaternary LFSR of length 8, where each register location can have 4 values. This LFSR behaves exactly the same as a binary LFSR except that all calculations are done in GF(4) instead of GF(2).

One obvious way to support these quaternary codes on the CVP is to add a quaternary LFSR to the design. There are two different design decisions that can be made in that case; first we could choose to add a full featured configurable quaternary LFSR to the design, hoping that there will be more standards that make use of the feature. This is a rather expensive solution, especially considering the observation that quaternary codes seem to be very uncommon. There is a good chance that the flexible quaternary LFSR that would be added in this case is only used in one specific configuration in UMTS.

With this in mind we could remove all the configuration possibilities from the quaternary LFSR and add hardware that is specific for the quaternary code in UMTS. This solution would be a lot cheaper; UMTS uses only one quaternary LFSR, with one specific generator polynomial. Since this generator polynomial is known, the transition matrix F that we introduced previously is fully specified now and we can computer the F^W, without suffering from an explosion of symbolic expressions. The result of this matrix exponentiation can now be implemented directly in hardware and it would only cost a handful of gates compared to a fully configurable solution. But still, this isn't what we want; the CVP shouldn't contain standard specific hardware. If we do this now, next time a 5-valued LFSR is needed for a new standard, or some strange hardware that we haven't even thought of yet. Will we add specific hardware for those standards too then? If so, we might as well replace the entire CGU by UTMS specific hardware now, because it will probably be cheaper, in the short term that is. So, it seems we need to take a different approach.

An approach that wouldn't violate the principle CVP design decisions, and would in fact be a great witness of its flexibility, is to find a way to map this very specific requirement in UMTS onto the flexible hardware that is already there. Our first (naïve) attempt to accomplish this is to interpret every quaternary value in the generator polynomial and state of the quaternary LFSR as a pair of binary values and configure a binary LFSR with these. This obviously doesn't work because calculations in GF(4) are fundamentally different from calculations in GF(2); mathematically there is no isomorphism between Z/4Z and (Z/2Z * Z/2Z) and practically our binary calculations didn't take the carry from the least significant bit to the most significant bit into account. Adapting our binary LFSR

to (conditionally) compensate for these differences is as bad as (if not worse then) adding specific hardware, so again, we need a different approach.

Henk van Tilborg from the TU/e introduced us to an algorithm by Berlekamp and Massey [14] that calculates for a given sequence the shortest LFSR that generates it. Using this algorithm, we discovered that, for every initial state, the binary interpretation of the sequence produced by the quaternary LFSR could be generated with one binary LFSR of length 72. We were able to improve on this number by splitting the binary sequence into two sequences; one with all the least significant bits and one with all the most significant bits of the binary interpretation of the quaternary values. This way, the sequence of least significant bits could be generated with a binary LFSR of length 8 and the sequence of most significant bits could be generated with a binary LFSR of length 36. Our architecture already uses 2 binary LFSRs, so both sequences could be generated in parallel. However, the LFSR of length 36 is longer that the LFSRs of length 32 that we chose to implement. We could of course solve this by increasing the length of our LFSRs or by using the sequential LFSR approach we introduced in the previous paragraph, but there is also a problem that we can't solve. An LFSR of length 36 needs to be initialized with 36 state bits, but the UMTS standard only provides 16, of which only 8 are most significant. So, in order to use the LFSR of length 36 we first need to have another means of finding the 28 missing initialization bits. This is basically begging the question, as an efficient solution to this problem would be a solution to the original problem. At this point, the most viable approach for generating the quaternary sequences seems to do so on the micro controller that is controlling the CVP. One aspect of the quaternary code in UMTS that supports this decision is the fact that the quaternary code is cyclic with a cycle length of only 256 values. This means that the micro controller only needs to generate 256 elements (= 512 bits) of a codeword, after which they can be stored and reused until a new codeword is required. This observation results in both a low load on the micro controller and small power advantage.

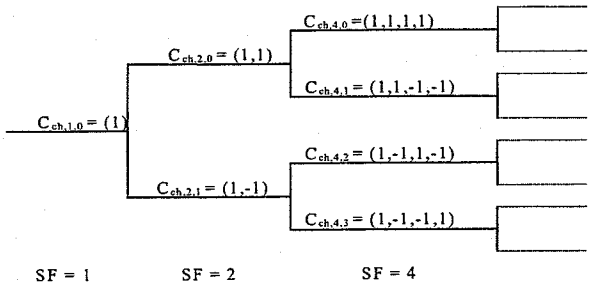
These quaternary codes show that there will always be exotic requirements that test the limits of the CVP's flexibility and it is only reasonable to assume that other standard will have requirements that we didn't and couldn't prepare for. It seems that this is something we just have to accept...

6 CGU Architecture

In this chapter we design the remaining functional units that we saw in the architecture overview in Chapter 3. At the end of this chapter we introduce the CGU's programming model, which consists of an Instruction Set Architecture (ISA) and a configuration and state vector definition. The PRN generators are obviously an important part of the CGU architecture, but they have already been the main topic of two chapters, so we won't mention them here anymore.

6.1 Hadamard/OVSF Code Generator

There is a second type of codes that is used to separate the individual user signals in the broad frequency band used in CDMA. These are the Orthogonal Variable Spreading Factor (OVSF) code group and the Hadamard code group. Both of these types are orthogonal codes, which means that the dot product of two different code words from a group equals 0 (after mapping the binary values to {1,-1}). Or, alternatively, when interpreted as vectors, two code words are perpendicular to each other. The two code groups contain exactly the same code words, but due to different generation methods the order of the code words in the groups are different. Figures 6.1 and 6.2 show representations of these two different methods together with the 8 code words of length (called Spreading Factor (SF)) 8 that are generated by these methods.



Code Number	Bit Value							
	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1	1
2	0	0	1	1	0	0	1	1
3	0	0	1	1	1	1	0	0
4	0	1	0	1	0	1	0	1
5	0	1	0	1	1	0	1	0
6	0	1	1	0	0	1	1	0
7	0	1	1	0	1	0	0	1

Figure 6.1 OVSF Code with SF = 8

$$H_0 = (1)$$
$$H_k = \begin{pmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{pmatrix}, \quad k \geq 1$$

Code Number	Bit Value							
	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0	1
2	0	0	1	1	0	0	1	1
3	0	1	1	0	0	1	1	0
4	0	0	0	0	1	1	1	1
5	0	1	0	1	1	0	1	0
6	0	0	1	1	1	1	0	0
7	0	1	1	0	1	0	0	1

Figure 6.2 Hadamard Code with SF = 8

Note that due to these generation methods, the SF for both the Hadamard code and OVSF code is always a power of 2.

The mapping of code numbers between Hadamard and OVSF appears quite complicated at first. A closer inspection reveals however that the mapping is in fact very simple; in code groups with equal SF, mapping a Hadamard code number to a OVSF code number or the other way around comes down to simply reversing the binary representation (of

length ^{2}log SF) of the code number. So, as a first simplification of the design, we can decide to implement either a Hadamard generator or an OVSF generator instead of both. Also, instead of implementing a generator for every different SF, we decide to implement one only for the largest SF we want to support, which is 512 at the moment. Due to this decision we need to find a way to map the generation of code words with smaller SF's onto this generator.

We are now left with a single design decision to make, namely the decision of whether to implement a Hadamard generator or an OVSF generator. Based on the previous two decisions there are 2 criteria on which we can base this last decision.

Assuming the availability of a type A generator with SF = SF_MAX:

1. Ease of generating type A code words with SF < SF_MAX
2. Ease of generating type B code words with SF ≤ SF_MAX

We will evaluate these criteria from both perspectives.

If we implement a Hadamard generator, the first criterion becomes trivial because every Hadamard codeword with a smaller SF is a prefix of the same Hadamard codeword with a bigger SF.
We already saw that mapping OVSF code numbers to Hadamard code numbers means reversing the binary representation of the code number. However, this statement is ambiguous now, because due to leading zeros the binary representation of a number is not uniquely defined and these leading zeros suddenly become very important when we are reversing the string. Some inspection reveals that when mapping an OVSF code with $SF_{OVSF} \leq$ SF_MAX we need to interpret the code number as a binary sequence of length (^{2}log SF_{OVSF}) and then reverse it. So, with a Hadamard generator, the difficulty in the second criterion is in the need to reverse bit strings of several different lengths, based on the allowed SF's.

Alternatively, when we implement an OVSF generator, the first criterion does require an actual mapping of code numbers. The mapping is easy though; OVSF code x with $SF_x \leq$ SF_MAX equals a prefix of OVSF code y = (SF_MAX/SF_x)*x with SF_y = SF_MAX.
Because Hadamard code words are the same in any SF (except for their length), we can always interpret a code number as a binary string of length (^{2}log SF_MAX) and reverse it to find the corresponding OVSF code number. This solution to the second criterion is a little bit easier than in the case of a Hadamard generator.

All in all, there doesn't seem to be a very compelling reason to go with either a Hadamard or an OVSF generator. Maybe small performance or usability advantages come up when programming for the CVP, but we won't go into that here.

All that remains now is to find a vectorized implementation of a Hadamard (or OVSF) generator. In his internship report [4] Tom Geelen showed a very nice implementation of a single step Hadamard generator (Figure 6.3) and a vectorized version thereof (Figure 6.4), which we won't and didn't need to improve upon.

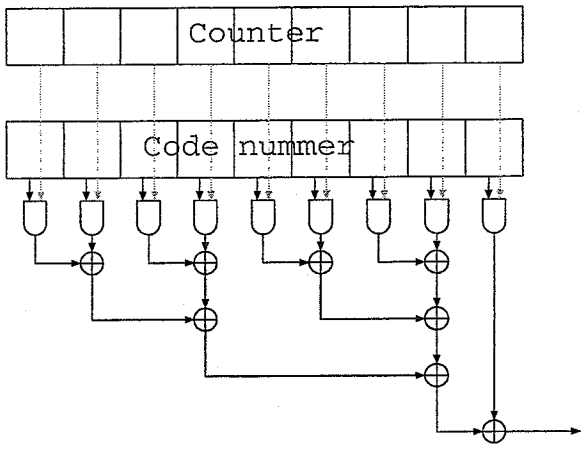


Figure 6.3 Hadamard generator, single step

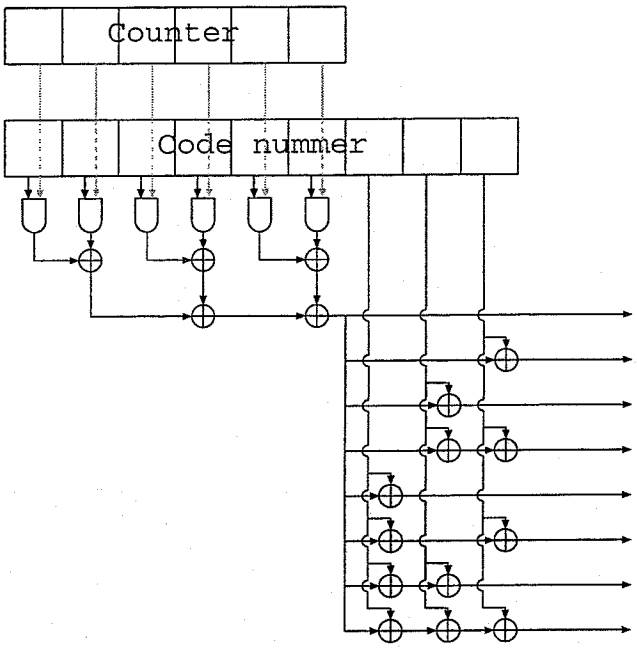


Figure 6.4 Hadamard generator, 8 step

Using the observation that an OVSF generator only requires the reversal of the code number, it would be very easy to derive an OVSF generator from this Hadamard generator. The hardware complexity in this design doesn't compare to our PRN generators and there is therefore no doubt that this design will reach the CVP's target clock speed. Also note that this design is easily extended to generate a code with a higher SF or to generate more bits per clock cycle, all with very little hardware and increase in complexity.

6.2 Table Look-Up functionality

Table Look-Up (TLU) functionality should be added to the design if only for its enormous flexibility. TLU can be used to "generate" any code imaginable and thus gives maximum flexibility. This seems too good to be true and, if fact, it is. Obviously, getting a value from a table is not the same as generating a value on the fly. It is only from the perspective of the Code Combiner that a value from a Look-Up Table (LUT) is indistinguishable from an actually generated code. In the case of UMTS we have already found one application were TLU will play an important role and that is in supplying the Code Combiner with quaternary codes. We found that generating these codes on the CVP

has drawbacks in any hardware implementation that we could think of. The most viable solution seems to be to have the controlling DSP generate the codes in software and have a LUT supply them to the Code Combiner. We will elaborate on the problems we faced with quaternary codes in paragraph 5.4. Also, a quick inspection of the CDMA2000 and TD-SCDMA standards shows that these require codes that can very likely not be generated with either LFSRs or a Hadamard generator, so in the design as it is now, these might need to be handled with TLU as well. The implementation of TLU functionality in the CGU is very simple. There is a register in the CGU that holds a value and supplies it to the Code Combiner. This register is filled with a value from the vector memory, using the scalar interface. There is more that can be said about this, but these implementation issues are not the topic of this report.

6.3 Code Combiner

The actual output of the CGU is usually not just a PRN sequence or a Hadamard/OVSF code. The output is a combination of these codes and there are several of these combinations, which are specified in the application standards. For this assignment we decided to look at combinations for UMTS and GPS and we design the code generator to be as flexible as possible under the requirements for these two standards. We start with the overview of the codes required by UMTS and GPS from Chapter 3.

Initial Combination Specification

$C_{sig,s}$ = Hadamard(s)

$C_{ch,SF,n}$ = Hadamard(f(n_{OVSF})) // f : $n_{OVSF} \rightarrow n_{Hadamard}$

$c_{long,1,n}(i)$ = $(x_n(i) + y(i))(0,1 := 1,-1)$
$c_{long,2,n}(i)$ = $(x_n(i+16777232) + y(i+16777232))(0,1 := 1,-1)$
$x_n : G(x) = X^{25}+X^3+1$
$y : G(x) = X^{25}+X^3+X^2+X+1$

$c_{short,1,n}$ = $(a(i)+2b(i)+2d(i))(0,1,2,3 := 1,-1,-1,1)$
$c_{short,2,n}$ = $(a(i)+2b(i)+2d(i))(0,1,2,3 := 1,1,-1,-1)$
$a : G(x) = X^8+X^5+3X^3+X^2+2X+1$ //Quaternary code
$b : G(x) = X^8+X^7+X^5+X+1$
$d : G(x) = X^8+X^7+X^5+X^4+1$

$C_{pre,n,s}(k)$
$C_{c\text{-}acc,n,s}(k)$ = $c_{long,1,n}(k) * C_{sig,s}(k) * e^{j(\pi/4 + k\pi/2)}$
$C_{c\text{-}cd,n,s}(k)$

$C_{long,n}(k)$ = $c_{long,1,n}(k) * (1 + j(-1)^k * c_{long,2,n}(2 * \lfloor k/2 \rfloor))$
$C_{short,n}(k)$ = $c_{short,1,n}(k) * (1 + j(-1)^k * c_{short,2,n}(2 * \lfloor (k/2 \rfloor))$

$S_{dpch,n}$
$S_{c\text{-}msg,n}$ = $C_{long,n}$ or $C_{short,n}$
$S_{r\text{-}msg,n}$

Z_n = $(x(i+n) + y(i))(0,1 := 1,-1)$
$x : G(x) = X^{18}+X^7+1$
$y : G(x) = X^{18}+X^{10}+X^7+X^5+1$

$S_{dl,n}(k)$ = $Z_n(k) + j\, Z_n(k + 131072)$

GPS C/A-code = G1(t) + G2(t + i)
$G1 : G(x) = X^{10}+X^3+1$
$G2 : G(x) = X^{10}+X^9+X^8+X^6+X^3+X^2+1$

This overview has a lot of detail that is irrelevant for the Code Combiner. First of all, the generator polynomials of the different PRN sequences are only relevant for the LFSR that generates them. Second, the exact amount by which a PRN sequence is delayed is not important; the only thing that the Code Combiner needs to know is that a particular codeword comes from the delayed output port of an LFSR. With the same argument, we also remove all code numbers from the specification. Next, we remove all unnecessary renamed codes from the specification, most notably S_{dpch}, $S_{c\text{-}msg}$ and $S_{r\text{-}msg}$. Finally, the specification mixes codes from the binary domain and codes from the complex domain. A mapping of binary/quaternary values to real values is used to introduce the multiplication that is needed for complex values. Note that there is an isomorphism between binary addition and multiplication in the set {-1,1}; the mapping doesn't change the specification, it only allows the use of the usual notation for complex values. To simplify hardware implementation, we rewrite the specification into a completely binary equivalent, using two bits to represent one complex value where needed. This results in the following simplified specification.

Simplified Combination Specification

$C_{sig} = H_1(i)$

$Z1 = LFSR_1(i) + LFSR_2(i) + H_1(i)$
$Z2 = SLFSR_1(i) + SLFSR_2(i) + H_1(i)$

GPS C/A-code $= LFSR_1(t) + SLFSR_2(t)$

$c_{long,1}(i) = LFSR_1(i) + LFSR_2(i)$
$c_{long,2}(i) = SLFSR_1(i) + SLFSR_2(i)$

$c_{short,1} = LFSR_1(i) + LFSR_2(i) + LUT_1(2i) + LUT_1(2i+1)$
$c_{short,2} = LFSR_1(i) + LFSR_2(i) + LUT_1(2i)$

$C_{pre}(k)$ = real: $\alpha + c_{long,1}(k) + C_{sig}(k)$
$C_{c\text{-}acc}(k)$ complex: $\beta + c_{long,1}(k) + C_{sig}(k)$
$C_{c\text{-}cd}(k)$ $(\alpha,\beta) \in \{(0,0),(1,0),(1,1),(0,1)\}^*$

$C_{long}(k)$ = real: $c_{long,1}(k)$
complex: $\alpha + c_{long,1}(k) + c_{long,2}(2 * \lfloor k/2 \rfloor)$
$\alpha \in \{0,1\}^*$

$C_{short}(k)$ = real: $c_{short,1}(k)$
complex: $\alpha + c_{short,1}(k) + c_{short,2}(2 * \lfloor (k/2 \rfloor)$
$\alpha \in \{0,1\}^*$

$S_{dl}(k)$ = real: $Z1(k)$
complex: $Z2(k)$

We replaced all PRN sequences by a reference to the LFSR that generates them, and all delayed PRN sequences by a reference to the delayed output port of the LFSR that generates them. In the same way, we replaced the Hadamard and OVSF sequences by a reference to the Hadamard functional unit. We removed the C_{ch} code from the specification, because we assume that the step required to translate an OVSF code number into a Hadamard code number occurs outside the CGU. The S_{dl} code shows that we need two versions of the Z code, a normal and a delayed one. In the simplified specification these are represented by Z1 and Z2. Also note that we add a Hadamard code to both Z1 and Z2, even though this wasn't part of the original specification. This is

because in the official specification, this addition is performed in a separate step, but to decrease the latency of code generation we merged this extra step into the Code Combiner. In this simplified specification it becomes clear that the quaternary code will be supplied with Table Look-Up. Every element of a quaternary codeword consists of 2 bits. To make the binary calculations easier, we separated the 16 least-significant bits from the most significant bits and supply them separately to the Code Combiner, via LUT(2i+1) and LUT(2i) respectively. There is a small simplification that we performed on $c_{short,2}$; because of the mapping that is used, we only need the most significant bit of every quaternary element. Because we rewrote the specification into binary form, we had to get rid of the power of *e*. As we mentioned before, this power of *e* is a rotation in the complex plane. We replace this rotation by an addition of a bit pair (α,β) that cycles through a vector of 4 elements. α is added to the real part and β to the imaginary part. This simplification is not completely correct, because the rotation in the complex plane is over a circle, and in the binary form we rotate over a square. We are missing a weighing factor here, but we decided to solve that problem outside of the CGU and keep the Code Combiner completely binary. In the C_{long} and C_{short} codes we see that the imaginary part of the value is multiplied with a value that toggles between -1 and 1. In binary form this implemented with an addition of a value that toggles between 1 and 0. The simplified specifications of C/A-code, S_{dl} code and the c_{long} codes are self-evident.

The simplified specification still uses the names of the different codes, but the Code Combiner has no notion of these names, it only has input and output ports. In the final rewrite of the specification we will generalize the formulas, add more detail and replace code names by in/output ports.

Generalized Combination Specification

For C_{long}

C_1: $(\forall i : 0 \le i < 16 : C_1(i) = LFSR_1(i) + LFSR_2(i))$

C_2: $(\forall i : 0 \le i < 16 : C_2(i) = SLFSR_1(i) + SLFSR_2(i))$

For S_{dl}, C_{pre}, $C_{c\text{-}acc}$ and $C_{c\text{-}cd}$

C_1: $(\forall i : 0 \le i < 16 : C_1(i) = LFSR_1(i) + LFSR_2(i) + H_1(i))$

C_2: $(\forall i : 0 \le i < 16 : C_2(i) = SLFSR_1(i) + SLFSR_2(i) + H_1(i))$

For C_{short}

C_1: $(\forall i : 0 \le i < 16 : C_1(i) = LFSR_1(i) + LFSR_2(i) + LUT_1(2i) + LUT_1(2i+1))$

C_2: $(\forall i : 0 \le i < 16 : C_2(i) = LFSR_1(i) + LFSR_2(i) + LUT_1(2i))$

For C/A (GPS)

C1: $(\forall i : 0 \le i < 16 : C1(i) = LFSR_1(i) + SLFSR_2(i))$

C2: $(\forall i : 0 \le i < 16 : C2(i) = LFSR_2(i) + SLFSR_1(i))$

C_{long} and C_{short}

OUT: $(\forall i : 0 \le i < 8 :$

$OUT(4i) = 0 + C_1(2i) + 0 * C_2(2i)$

$OUT(4i+1) = 0 + C_1(2i) + 1 * C_2(2i)$

$OUT(4i+2) = 0 + C_1(2i+1) + 0 * C_2(2i)$

$OUT(4i+3) = 1 + C_1(2i+1) + 1 * C_2(2i))$

C_{pre}, $C_{c\text{-}acc}$ and $C_{c\text{-}cd}$

OUT: $(\forall i : 0 \le i < 8 :$

$OUT(4i) = \alpha + C_1(2i)$

$OUT(4i+1) = \beta + C_1(2i)$

$OUT(4i+2) = \gamma + C_1(2i+1)$

$OUT(4i+3) = \delta + C_1(2i+1))$

$(\alpha,\beta,\gamma,\delta) \in \{(0,0,1,0),(1,1,0,1)\}^*$

S_{dl}

OUT: $(\forall i : 0 \le i < 8 :$

$OUT(4i) = 1 * C_1(2i) + 0 * C_2(2i)$

$OUT(4i+1) = 0 * C_1(2i) + 1 * C_2(2i)$

$OUT(4i+2) = 1 * C_1(2i+1) + 0 * C_2(2i+1)$

$OUT(4i+3) = 0 * C_1(2i+1) + 1 * C_2(2i+1))$

C/A (GPS)

OUT: $(\forall i : 0 \le i < 8 :$

$OUT(4i) = C_1(2i)$

$OUT(4i+1) = C_1(2i)$

$OUT(4i+2) = C_1(2i+1)$

$OUT(4i+3) = C_1(2i+1))$

This specification clearly divides the Code Combiner into two stages. In the first stage a selection of the input ports is used to generate 2 intermediate binary code-vectors of 16 elements. The next stage combines these two binary vectors and some constant vectors into one complex vector of 16 elements, which requires 16*2=32 bits. This means that in a 32 bit vector every even bit is the real part of a vector element and every odd bit is the imaginary part of a vector element. Also note that between the previous and this version of the specification some extra terms have been added, such as in the case of S_{dl}. This is part of the generalization. If we added terms that weren't there before, they are just multiplied with 0 to make sure they don't influence the code. The result of the generalization is that the outputs for the different codes are now specified in a very uniform way. Every codeword is now built-up from an optional constant vector, added to an optional masked C_1 codeword and an optional masked C_2 codeword.

The specification now has enough detail to be implemented in hardware. The hardware implementation uses more stages than the two we just mentioned. These stages are

suggested by the generalized specification and include among others a stage that gives the opportunity to apply a constant mask to the intermediate code words and a stage that allows the addition of a constant vector to the final output. These extra stages increase the flexibility of the Code Combiner. A schematic representation of this hardware implementation is depicted in Figure 6.5.

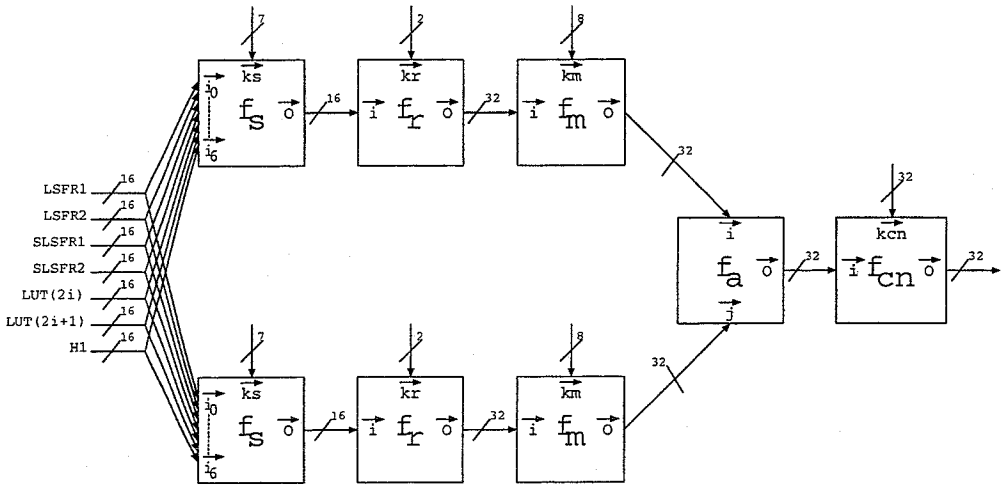


Figure 6.5 Code Combiner

The inputs to the Code Combiner shown in Figure 6.5 are the outputs of the functional units of the CGU. LFSR1 and SLFSR1 are the normal and delayed output of the first PRN generator respectively. LFSR2 and SLFSR2 are the outputs of the second PRN generator. The TLU functionality of the CGU offers a 32 bit wide register, which is interpreted as 16 elements of 2 bits. To have separate access to the first and second bit of each element we assume that the 32 bits value is sent to the Code Combiner over two 16 bits ports LUT(2i) and LUT(2i+1) transporting the first and second bit of every element respectively. H1 is the output of the Hadamard/OVSF generator. Finally, the Code Combiner has access to the configuration register of the CGU, which provides the constants kx. Note that the two instantiations of f_s, f_r and f_m receive different constants ks, kr and km. The specification of the functional units in this implementation is as follows:

Combination Function Specification

f_s: $(\forall n : 0 \le n < 16 : o_n = (\Sigma m : 0 \le m < 7 : ks_m * i_m[n]))$
//This function selects a subset of the input vectors and calculates a bit-wise addition.

f_r: $(\forall n : 0 \le n < 8 \wedge (kr_0, kr_1) = (0, 0) : (o_{4n}, o_{4n+1}, o_{4n+2}, o_{4n+3}) = (i_{2n}, i_{2n}, i_{2n}, i_{2n}))$
$(\forall n : 0 \le n < 8 \wedge (kr_0, kr_1) = (0, 1) : (o_{4n}, o_{4n+1}, o_{4n+2}, o_{4n+3}) = (i_{2n}, i_{2n}, i_{2n+1}, i_{2n+1}))$
$(\forall n : 0 \le n < 8 \wedge (kr_0, kr_1) = (1, 0) : (o_{4n}, o_{4n+1}, o_{4n+2}, o_{4n+3}) = (i_{2n+1}, i_{2n+1}, i_{2n}, i_{2n}))$
$(\forall n : 0 \le n < 8 \wedge (kr_0, kr_1) = (1, 1) : (o_{4n}, o_{4n+1}, o_{4n+2}, o_{4n+3}) = (i_{2n+1}, i_{2n+1}, i_{2n+1}, i_{2n+1}))$
//This function implements a doubling scheme to go from 16 to 32-bit vectors

f_m: $(\forall n : 0 \le n < 32 : o_n = i_n * km_{(n \bmod 8)})$
//This is a mask

f_a: $(\forall n : 0 \le n < 32 : o_n = i_n + j_n)$
//This function adds two 32-bit vectors together, using bit-wise addition

f_{cn}: $(\forall n : 0 \le n < 32 : o_n = i_n + kcn_n)$
//This function adds a constant 32-bit vector to the code vector, using bit-wise addition.

Now that we have defined all functional units and introduced all required configuration parameters, we can define the programming model.

6.4 Programming Model

In a programmable and re-configurable architecture, the programming model consists of the Instruction Set Architecture (ISA), the configuration parameters and a state definition. This paragraph gives a brief overview of these concepts. More detailed information can be found in The CVP Instruction Set Architecture Reference Manual [15].

A CGU instruction consists of three sub-instructions, which can be executed in parallel.

1. a vector sub-operation;
2. a scalar sub-operation;
3. a scalar-receive sub-operation.

The grammar for a CGU instruction is as follows:

```
CGU_cmd   =   (vopc, sopc, srcv)
vopc      =   NOP | CONFIG | RCV_STATE | SND_STATE | LEAP
sopc      =   NOP | SSND
srcv      =   NONE | VMU
```

Operation	Explanation
NOP	The standard No Operation.
NONE	Also a NOP, but in the scalar receive sub-operation the operation defines from which functional unit to receive, in which case NONE means that no scalar at all has to be received, which is effectively a NOP.
CONFIG	The operation that loads a configuration vector into the CGU.
RCV_STATE	The operation that loads a state vector.
SND_STATE	The operation that sends the current state.
LEAP	The operation that generates a code vector. This precise effect of this operation is steered by the configuration vector.
SSND	The operation that sends a value over the scalar data-path.
VMU	The operation that receives a scalar from the Vector Memory Unit.

A 256-bit configuration vector is defined as follows:

Name	# bits	Explanation
poly_g1	32	Generator polynomial for $LFSR_1$
poly_h1	32	Delay offset polynomial for $LFSR_1$
input_en1	1	CRC input enable for $LFSR_1$
unused1	5	Unused PRN generator 1 bits (32 – unused1) = actual polynomial length
poly_g1	32	Generator polynomial for $LFSR_2$
poly_h1	32	Delay offset polynomial for $LFSR_2$
input_en1	1	CRC input enable for $LFSR_2$ (reserved)
unused1	5	Unused PRN generator 1 bits (32 – unused1) = actual polynomial length
code_nr	9	Hadamard code number
ks1	7	Selected inputs for combiner branch 1
ks2	7	Selected inputs for combiner branch 2
kr1	2	Doubling scheme for combiner branch 1
kr2	2	Doubling scheme for combiner branch 2
km1	8	Masking pattern for combiner branch 1

km2	8	Masking pattern for combiner branch 2
kcn	32	Conditional negate pattern

Finally, a state vector is defined as follows:

Name	**# bits**	**Explanation**
lfsr1	32(+16)	State of $LFSR_1$
lfsr2	32(+16)	State of $LFSR_2$
counter	(5)	Hadamard generator counter state

Values between parentheses are optional. The extra 16 state bits of a PRN generator aren't actually needed to be able to reconstruct the state after a context switch, but that would require one extra operation. Also, it seems that a context switch will never occur in the middle of a Hadamard code, but if this should change we can decide to save the counter state as well.

7 Conclusions

7.1 Accomplishments

We designed a modular, configurable Code Generator that currently implements the code-requirements for UMTS and GPS. The two main code-generating functional units, a Hadamard/OVSF generator and a PRN generator, are basic requirements for other 3G standards as well. The implementation of these units in our design is highly configurable and reusing them for standards such as CDMA2000 and TD-SCDMA will probably require only minor adjustments. The modular design of the CGU allows the use of more of these functional units if they are required by other standards. The Code Combiner, which combines the output of the code-generating functional units into the final output of the CGU, is designed to be as flexible as possible, but is still the most specific unit in the CGU. Adding code-generating units to the CGU will require some adjustments to the Code Combiner.

Our work on the PRN generator has resulted in a new way of generating multiple LFSR sequence elements in parallel. The new method allows for a higher throughput of sequence elements than in previously known solutions, while remaining completely re-configurable. The resulting LFSR was subsequently adapted to generate a delayed output sequence in parallel with the normal output sequence, which is a well-known capability of standard LFSRs. The second adaptation that was performed allows for the mapping of lower degree generator-polynomials onto our LFSR of predefined length N. This adaptation greatly increases the flexibility of the LFSR, and enables us to use a single LFSR to implement all the LFSRs, of varying length (up to and including N), required by the different 3G standards. Finally, we added the ability to process input vectors to the LFSR design. This adds the capability to generate CRC signatures of an input stream.
All these additions to the basic high-throughput LFSR design didn't influence the designs latency and with practical dimensions (length 32, step-size 16), the LFSR can run at over 300 Mhz., resulting in a throughput of approx. 5 Gb/s. With a pipeline register between the code-generating functional units and the Code Combiner, the entire CGU will be able to run at 300 Mhz., the target clock-speed of the CVP. To work around the limitation of generator polynomials of degree up to and including N, we discovered that it is possible to emulate the functionality of a longer LFSR with several sequential steps of a shorter LFSR. However, implementing this feature into a future version of the CGU requires some more detailed research.

7.2 Future work

One of the most important goals of the CVP is its programmability and with this, its support for multiple 3G standards. The CGU is responsible for generating codes for these different standards. Currently, the CGU only fully supports code generation for UMTS and GPS. The design of the CGU was kept as flexible as possible under the requirements for these standards, but not much research went into determining whether or not this flexibility is enough to implement standards such as CDMA2000 and TD-SCDMA. Preliminary indications suggest that this is not the case and that some additions will have to be made to the CGU to accommodate these standards.

While our work on LFSRs for the PRN generator have resulted in a design that is sufficiently fast and flexible for the CGU, there are still some topics left that deserve a closer look. Our results for Galois LFSR in particular were less than satisfying. A better design for a multi-step Galois LFSR would also benefit CRC, because a Fibonacci LFSR

doesn't always produce the expected signature. Another topic that might deserve some research would be to compare LFSRs with different step-sizes and see if an efficient solution can be found that supports several different step-sizes. Finally, we only briefly looked at using an LFSR to generate the sequence of a longer LFSR in several sequential steps. It would be interesting to see if minor adjustments to our basic LFSR make this sequential approach easier. A related idea uses several "pipelined" short LFSRs to speedup the sequential LFSR approach.

Bibliography

[1] Tero Ojanperä and Ramjee Prasad. *Wideband CDMA For Third Generation Mobile Communications*. Artech House Publishers, 1998.

[2] *Qualcomm Introduction to CDMA*. http://www.qualcomm.com/ProdTech/cdma/training/cdma25/intro/modules.html

[3] *SW-MODEM Homepage*. http://pww.research.philips.com/natlab/sw-modem (Philps internal only)

[4] T.J.H. Geelen and R.H.M. Wubben. *Flexible UMTS code generator for the SCUBA Co-Vector Processor*. Technical Note 2000/409, Philips Research.

[5] R.H.M. Wubben. *The code generator re-visited*. Philips Research.

[6] Gruodis et al. *US Patent 5.412.665 Parallel operation linear feedback shift register*.

[7] Paul H. Bardell et al. *Built-in test for VLSI: Pseudo Random Techniques*. Wiley, New York, 1987.

[8] V.N. Yarmolik and S.N. Demidenko. *Generation and Application of Pseudorandom Sequences for Random Testing*. Wiley, Chichester, 1988.

[9] 3rd Generation Partnership Project. *Technical Specification Group Radio Access Network. Spreading and modulation (FDD) (Release 5)*. (3GPP TS 25.213 v5.1.0)

[10] *GPS Standard Positioning System Signal Specification 2nd Edition*.

[11] Elliot D. Kaplan. *Understanding GPS Principles and Applications*. Artech House Publishers, 1996.

[12] Chinese Wireless Communication Standard (CWTS). *Working Group 1 (WG1). Spreading and modulation*. (TS C104 v3.3.0)

[13] *Physical Layer Standard For CDMA2000 Spread Spectrum Systems Release C*. http://www.3gpp2.org/Public_html/specs/C.S0002-C_v1.0.pdf

[14] *Algorithmic Coding Theory (Chapter 7)*. New York McGraw-Hill, 1968

[15] P.P.E. Meuwissen and C.H. van Berkel. *The CVP Instruction Set Architecture version 1 Reference Manual*. Technical Note 2003/000, Philips Research.

Appendix

Lemma 4.1 $FP_W = Q_{W+1}$

In this step we will define Q_W such that $Q_{W+1} = F\,P_W$. Because the content of P depends on W we cannot use simple matrix multiplication to derive Q_W. Instead we will use the special form of F and the definition of P_W to derive a similar definition for Q_W. The first observation we can make is that the sub diagonal of 1's in F causes the bottom N-1 rows from P_W to be copied to the first N-1 rows in the result. So, a first approximation to a definition for Q_W is:

$$
\begin{array}{lll}
Q_{i,j} = & \text{if} & (j-i=1) \wedge (i < N-W) \Rightarrow 1 \\
& [] & (i+j \geq 2N-W-1) \wedge (i < N-1) \Rightarrow p_{i+j-2N+W+1} \\
& [] & \text{else} \Rightarrow 0 \\
& \text{fi} &
\end{array}
$$

This definition does not specify the last row of Q_W correctly yet. The second observation we can make is that the truncated diagonal in P_W copies the first part of the last row of F to the last row of Q_W. The final observation is that the remaining positions in the last row of Q_W are the dot products of the last row of F and the last columns of P_W. It is a bit tricky to see, but these dot products follow exactly the definition of p_i, to form new p-terms in the last row of Q_W. These last two observations result in the following definition of Q_W:

$$
\begin{array}{lll}
Q_{i,j} = & \text{if} & (j-i=1) \wedge (i < N-W) \Rightarrow 1 \\
& [] & (i+j \geq 2N-W-1) \wedge ((i < N-1) \vee (j > N-W)) \Rightarrow p_{i+j-2N+W+1} \\
& [] & (i=N-1) \wedge (j \leq N-W) \Rightarrow g_j \\
& [] & \text{else} \Rightarrow 0 \\
& \text{fi} &
\end{array}
$$

Lemma 4.2 $Q_{W+1} = P_{W+1} K_W$

In this step we will define a matrix K_W such that $P_{W+1}\,K_W = Q_{W+1}$. Because K_W is split of Q_{W+1} on the right side we can apply it to G_W in the next step. First observe that the last W-1 columns in P_{W+1} are the same as the ones in Q_{W+1}, this equality is preserved if there is a truncated diagonal of ones in the lower right corner of K_W. Next, the top $N-W$ rows in P_{W+1} are shifted one to the right in Q_{W+1}, which we accomplish with a truncated sub-diagonal of ones, one of center, in K_W. Finally, the last row in Q_{W+1} has g_i's in it that are not present in P_{W+1}. Luckily, the $(N-W)^{\text{th}}$ element in the last row of P_{W+1} is p_0, which equals 1, and we can use this element to copy values from the $(N-W)^{\text{th}}$ row of K_W to the last row of Q_{W+1}. This results in the following definition of K_W.

$$
\begin{array}{lll}
K_{i,j} = & \text{if} & (i=j) \wedge (i > N-W-1) \Rightarrow 1 \\
& [] & (j-i=1) \wedge (i < N-W-1) \Rightarrow 1 \\
& [] & (i=N-W-1) \wedge (j < N-W) \Rightarrow g_j \\
& [] & \text{else} \Rightarrow 0 \\
& \text{fi} &
\end{array}
$$

Lemma 4.3 $K_W\, G_W = G_{W+1}$

Here we only have to show that $G_{W+1} = K_W\, G_W$. A few simple observations will show this. First, the truncated diagonal of ones in K_W copies the last rows in G_W to G_{W+1}, which are supposed to be equal. The sub diagonal of ones in K_W moves the sub diagonal of ones in G_W up by one, which is also what we want. Finally, the sub diagonal of ones in G_W copies the g_i's in the $(N-W)^{\text{th}}$ row of K_W to the right place in the $(N-W)^{\text{th}}$ row of G_{W+1}.

Lemma 5.1 $F^i\, Y(j) = P_{i+1}\, Y(j)$

Because only the last element in $Y(t)$ is non-zero, all we need to show is that the last column of F^i equals the last column of P_{i+1}. Theorem 1 tells us that $F^i = P_i\, G_i$ and thus the last column of F^i consists of the dot products of the rows of P_i and the last column of G_i. As we concluded in a similar situation in the proof of theorem 1, these dot products follow exactly the definition of the p-terms in P_W. If this last observation is clear it is only a small step to conclude that this column of p-terms is equal to the last column in P_{i+1}.

Lemma 5.2 $P_W\, Y(0) + \cdots + P_1\, Y(W-1) = P_W \cdot (0 \mid y_{W-1}, \cdots, y_0)^T$

This lemma is actually fairly straightforward if you have a good feel for the structure of P_i.

$P_W\, Y(0) + \cdots + P_1\, Y(W-1)$
$\equiv$ {math, definition of P_i and $Y(j)$}
$(0 \mid P_0\, y_0, \cdots, P_{W-1}\, y_0)^T + (0 \mid P_0\, y_1, \cdots, P_{W-2}\, y_1)^T + \cdots + (0 \mid P_0\, y_{W-1})^T$
$\equiv$ {vector addition}
$(0 \mid P_0\, y_0,\ P_1\, y_0 + P_0\, y_1, \cdots, P_{W-1}\, y_0 + P_{W-2}\, y_1 + \cdots + P_0\, y_{W-1})^T$
$\equiv$ {math, definition of P_i}
$P_W \cdot (0 \mid y_{W-1}, \cdots, y_0)^T$