Data-dependent Superimposed Training Architecture for Wireless Communication Systems

por

Fernando Martín del Campo Ramírez

Tesis sometida como requisito parcial para obtener el grado de

MAESTRO EN CIENCIAS EN LA ESPECIALIDAD DE CIENCIAS COMPUTACIONALES

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica
Agosto 2008
Tonantzintla, Puebla

Supervisada por:

Dr. René Armando Cumplido Parra, INAOE

Data-dependent Superimposed Training Architecture for Wireless Communication Systems

Fernando Martín del Campo Ramírez

August 19, 2008

Abstract

In order to correctly recover the information sent through the air in a digital wireless system, it is usually essential to have the most accurate description or model of the transmission channel. The majority of the existing commercial systems accomplish this by inserting, along with the information, a series of known symbols, commonly called pilots or training sequence symbols, whose analysis is used to make an estimation of the channel, that is, to obtain the parameters that describe the distortions caused over the data. Nevertheless, a part of the available bandwidth has to be used by these symbols, like in the time division multiplexed training, where some of the time slots are used for the training sequences. Until now, no alternative solution has demonstrated to be fully satisfying for commercial use, but this could change thanks to a recently developed technique known as superimposed training (ST), which does not need extra bandwidth for its training sequences (TS), as it adds them arithmetically to the data. This work describes a hybrid software-hardware FPGA implementation of an algorithm that belongs to the ST family: the Data-dependent Superimposed Training (DDST), that adds, apart from the TS, a third sequence known as data-dependent sequence, that destroys the interference caused by the data over the TS. As DDST's computational burden is very high for the available microprocessors used in mobile systems, a different approach is proposed in order to solve the problem of the implementation of a digital receiver that uses this algorithm.

Resumen

Para recuperar correctamente la información enviada a través del aire en un sistema digital de comunicaciones inalámbrico, es necesario contar con la descripción o modelo más preciso del canal de transmisión. La mayoría de los sistemas comerciales realizan ésto insertando, junto con la información, una serie de símbolos conocidos, normalmente llamados pilotos o símbolos de entrenamiento. Su análisis permite realizar una estimación del canal, es decir, obtener los parámetros que describen las distorsiones causadas sobre los datos. Sin embargo, una parte del ancho de banda disponible tiene que ser usada por éstos símbolos, como en el caso del entrenamiento multiplexado por división de tiempo, en el que algunas de las ranuras de tiempo son usadas para las secuencias de entrenamiento. Hasta ahora, ninguna solución alternativa ha demostrado ser completamente satisfactoria para ser usada en sistemas comerciales, pero esto podría cambiar gracias a una técnica recientemente desarrollada, conocida como entrenamiento superimpuesto (ES), que no necesita de un ancho de banda extra para sus secuencias de entrenamiento (SE) debido a que éstas son sumadas aritméticamente a la información. Este trabajo describe una implementación híbrida hardware-software FPGA de un algoritmo que pertenece a la familia de ES: el Entrenamiento Superimpuesto Dependiente de Datos o DDST por sus siglas en inglés. DDST suma a la información, además de la secuencia de entrenamiento, una tercera secuencia (la secuencia dependiente de datos) que destruye la interferencia causada por los datos sobre la SE. Como la carga computacional del DDST es muy alta para los microprocesadores disponibles en los sistemas móviles, una solución alternativa es propuesta para resolver el problema de la implementación de un recepotor digital que use este algoritmo.

Dedication

To the reader, all the words in this document were written hoping that you find them usefull.

Acknowledgements

First of all, I want to thank my parents for their unconditional support all these years.

I would also like to express my gratitude to my supervisor, along with the rest of the Computer Science Department of the INAOE, for all the knowledge they shared with me. Your career reflects your nobility and your compromise with your fellows.

My gratitude to all those people that have granted me their friendship. You are a part of me and my life is at your service. Edgar, Judith, Marco, Oscar, Hugo, Claudia, Alex, thank you.

A very special thanks goes out to Mónica, you can be my cousin, but I see you as my sister.

Finally, I thank the most special person for me. Ata, you are my life, thank you for making me a better human being and for believing in me.

Contents

Chapter 1

Preface

In a broad sense, a communications link consists of three basic blocks: the transmitter, the transmition channel, and the receiver. Nevertheless, to create, improve or study a real digital communication system, a more detailed analysis of each component has to be done. Figure 1.1 shows the basic block diagram of such system.

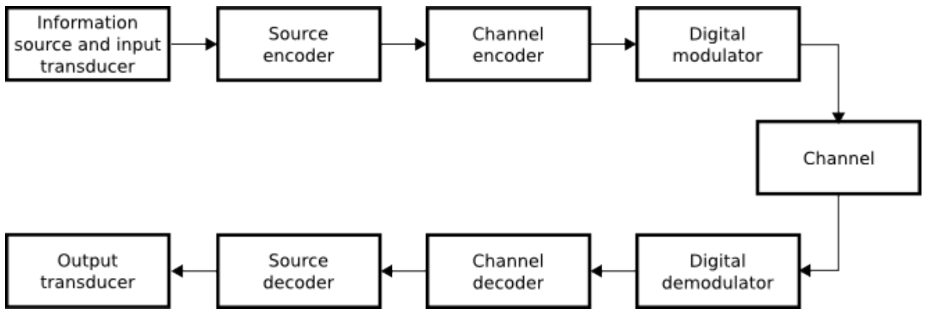


Figure 1.1: Digital communication system basic blocks.

The exact function of each block is not important at the moment, but it illustrates a powerful approach used to analyze a system: to divide it and treat it as a group of independent modules or subsystems working together.

The complexity of each block depends on the transmission method and on the characteristics of the channel (attenuation, bandwidth, noise and interference, and distortion.)

The noise and interference issues are particularly important in wireless communications systems that use the air as their transmission channel. The air is inherently noisy and its nature can contribute to the presence of different kinds of interference, like the one known as Intersymbol Interference or ISI, in which the energy of the message symbols is spread so a part of it overlaps with that of the neighboring symbols. The ISI can, in fact, make almost impossible to the detector inside the receiver to differentiate between a symbol and the spread energy of consecutive ones.

As it is not possible to avoid the influence of the air on a transmitted signal sent through it, it is necessary to find an alternative solution to the problem. Fortunately, this channel can be modeled as a linear system, whose effects can be reverted in the receiver, if its parameters are known with enough precision. Using this approach, the problem is now to have, at any time, the necessary information to estimate the channel as accurately as possible. This method is known as channel modeling, and its results can be then used in a block called equalizer which is inside the receiver. The purpose of the equalizer is to counteract any alteration of the transmitted signal caused by the channel, mainly the ISI. The process done by this block is called equalization. Having the diagram of figure 1.1 as a reference, the equalization is done inside the block named *digital demodulator*, just before the channel decoding.

To achieve the channel estimation, there are several methods and techniques. The next diagram (figure 1.2) shows a general classification of such methods.

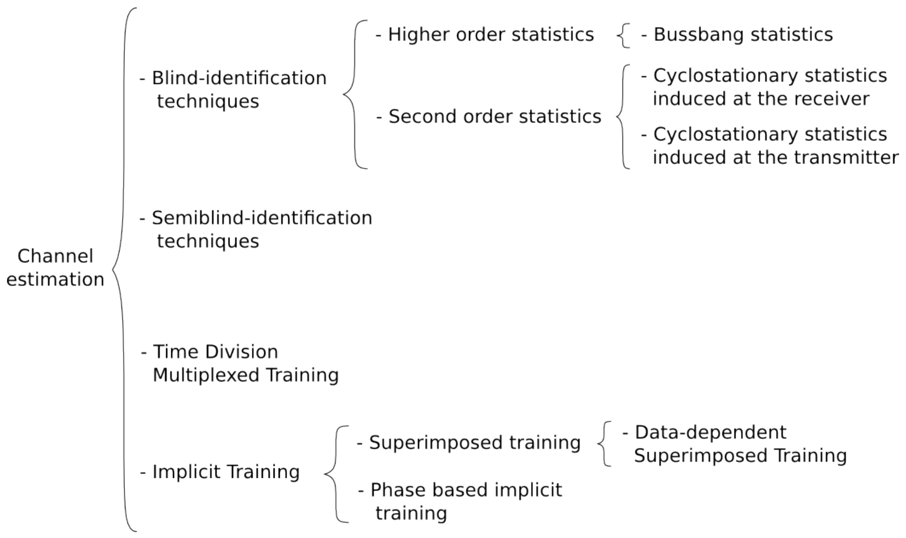

Channel estimation
- Blind-identification techniques
- Higher order statistics
- Bussbang statistics
- Second order statistics
- Cyclostationary statistics induced at the receiver
- Cyclostationary statistics induced at the transmitter
- Semiblind-identification techniques
- Time Division Multiplexed Training
- Implicit Training
- Superimposed training
- Data-dependent Superimposed Training
- Phase based implicit training

Figure 1.2: Taxonomy of channel estimation techniques

In the blind-identification techniques, there is no reference (training) sequence. In this case, the probabilistic and statistical properties of the transmitted sequence are the only information available [1]. The problem with these techniques is that their computational complexity is very high, even in the case of the second order statistics versions of this approach [9].

When talking about the semi-blind estimation, the transmitter sends several known training symbols in specific positions, and a functional is constructed that depends on both known symbols and the unknown ones. This way, the information carried out by the known part is traditionally exploited, but the use of the unknown part also enhances channel estimation performance [3].

Time division multiplexed training (TDMT), also known as common or traditional training, is the most used technique in digital wireless systems, like TDMA and CDMA. In this scheme, the pilots are inserted, usually with a regular period, into the data at certain time slots. A part of the transmitted frames in these systems is destined to the training sequences, so part of the bandwidth is used for this purpose.

An alternative to this traditional approach is the superimposed training, a branch of the implicit training. Here the training sequence is arithmetically added to the data sequence, saving the necessity of more bandwidth at the expense of a little power loss of the information signal [4]. As the training sequence cannot be seen explicitly in the transmitted signal, channel estimation must be carried out using statistical information.

A problem with the superimposed training is the interference caused by the training sequence over the information symbols. One of the alternatives recently proposed to eliminate this interference is the so called Data-dependent Superimposed Training. This technique adds a third sequence (data dependent sequence) that destroys the interference canceling the input signal DFT components in which the training sequence has energy.

1.1 Motivation

Even though simulations with the DDST method show a performance that can compete with the TDMT [16], [11], [28], the inherent computational burden of the method has made, at the moment, impossible to implement it in commercial digital communication systems, due to the time, power and space constraints imposed by the devices used in this field. Until now, the only alternative solution has been the use of a DSP architecture, that does not run above the hundreds of kHz order.

It is true that obtaining a fast and functional DDST solution is a very complex task, but the promise of a larger available bandwidth for the information is an attractive goal. Moreover, the method, seen as a set of individual steps, presents challenges for which an optimal solution has not yet been found, so the fact of, at least getting closer to them, can be of use for other open problems.

From an academic point of view, the DDST implementation is very attractive, as neither a full software nor a hardware approach seems to be a satisfying solution. On the one hand, a software alternative is, at this moment, unviable, due to the time it would require for obtaining a channel estimate and then using such estimate for the equalization. On the other hand, a hardware architecture, for example using an FPGA thinking toward the construction of an ASIC, presents several problems, caused by the enormous amount of data that has to be operated constantly, the high degree of data dependencies between stages of the process (which make very difficult to use techniques as parallelization and pipelining) and the complex control required by some of the mathematical operations that have to be performed.

1.2 Objectives

1.2.1 General Objective

To develop a functional hybrid software-hardware Data-dependent Superimposed Training digital receiver, based on an FPGA *System on a Programmable Chip*, as a *proof of concept* for future implementation in commercial mobile systems.

1.2.2 Specific Objectives

- To identify those stages of the DDST digital receiver algorithm that present the greatest challenges for a software or hardware implementation, proposing, for each of them, the best possible solution allowed by the available resources.
- The software component of the system must be easy to modify, according to possible improvements in the DDST algorithm or in the system performance, while the hardware component must be highly parameterizable, as many of the variables in the DDST receiver can modify their value according to the specific digital system in which it is used.
- To determine, from the analysis of the DDST receiver algorithm, if it is possible to reduce the computational load on the main processor by either mathematical transformations of the original equations, or by the use of hardware accelerators for the most time demanding processes.

Chapter 2

State of the Art

2.1 Training in Wireless Communication Systems

2.1.1 Time Division Multiplexed Training

Many wireless communication transmitters group the data to send through the channel in sets known as frames. These structures are divided in fields according to the purpose they serve. For example, the data itself, the control bits, the checksum, the addresses, etcetera.

The most popular way in which the channel is estimated in these systems is the *time division multiplexed training*. This scheme (used in the GSM standard, for example) intersperses its training sequence with the data blocks in the frames to send. The technique gives good channel estimates, but it needs to use part of the available bandwidth to accommodate the training sequence. For example, the training in a GSM normal burst (the way in which information is transmitted) represents almost 18% of the total load and so it will require 18% of the time destined to transmition [2]. The synchronization burst of GSM uses the 43.24% for the training sequence.

2.1.2 Implicit Training

Research on the field of implicit training is relatively new compared to other techniques used for channel estimation and equalization, and, theoretically, there are several ways to add training sequences in an implicit fashion to the information to be sent by a communication system. One example of implicit training is to add the pilot sequence to the information sequence by modifying its phase, like in (2.1). *b(k)* represents the information sequence, while *c(k)* refers to the training sequence and *s(k)* to the real data that will be transmitted.

$$s(k) = b(k) \cdot e^{j \cdot c(k)} \tag{2.1}$$

Nevertheless, the most common way in which training sequences are included with the information is to arithmetically add both of them (2.2).

$$s(k) = b(k) \; + \; c(k) \tag{2.2}$$

This technique is called *Superimposed Training* [4].

2.1.3 Superimposed Training

The subject of superimposed training was first discussed in [9]. This work proposes the use of a pilot sequence that is added to the information. The estimation of a single carrier system is considered, but the performance of such estimation is not analyzed. Although it also considers the problem of synchronization, the proposed solution is not correct [3]. The works described on [12] and [13] follow the same line, but in fact they do not do any relevant contribution, though [13] does a performance analysis of the channel estimation.

The work done by Orozco-Lugo on [3] tackles two of the main problems present on implicit training methods: training sequence synchrony (TSS) and the dc-offset. The first one is solved using fourth-order statistics while the dc-offset, which usually appears on direct conversion modern receivers, is addressed using polynomial rooting. The performance analysis covers any periodic training sequence and a family of sequences, that offers desirable properties for the channel estimation, is identified. An alternative proposal showing a better estimation of the dc-offset with a lower computational complexity is discussed on [14]. The method does its computation on the frequency domain, though the synchronization problem is tackled until [15], presented a year later. At this time, the methods published on [4] and [11] focused on a still lower complexity and a higher quality of channel estimation. In fact, both methods can be applied on practical cases due to their compromise between quality and complexity. [18] and [19] tackle, respectively, a specific case of the ST and an optimal power assignation analysis for the training and information sequences.

The remaining works have focused on the application of implicit training methods on communication networks with specific characteristics: [20] and [21] are oriented towards CDMA, while [22] covers ultra wide band networks and [23] to time-space codification along with ST. By the other side, [24] and [25] discuss the problem of ST on multiple transmitter and receiver antennas systems. Finally, [26] talks about the problems of superimposed training on ad-hoc networks.

2.1.4 Data-dependent Superimposed Training

The newest variant of ST (Data-dependent Superimposed Training) is presented in [16]. DDST cancels the undesirable effects caused by the interference between data and the training sequence. Suppressing this interference, the channel estimation performance is notoriously enhanced. In fact, without the presence of channel noise, DDST estimation is perfect.

DDST is not the only alternative to overcome the problem of the interference over the training sequence. Another one is the algorithm proposed in [17], where lineal precoding is applied to the data at the transmitter, while the training sequence is orthogonally added to the resulting sequence. At the receiver, data are processed block by block, using projections on a subspace orthogonal to the precoding matrix. The channel estimation is done at the end of this process.

DDST Estimation Procedure

Figure 2.1 shows the discrete time model of the DDST communications link. The objective of the DDST receiver is to remove all the transformations suffered by such signal.

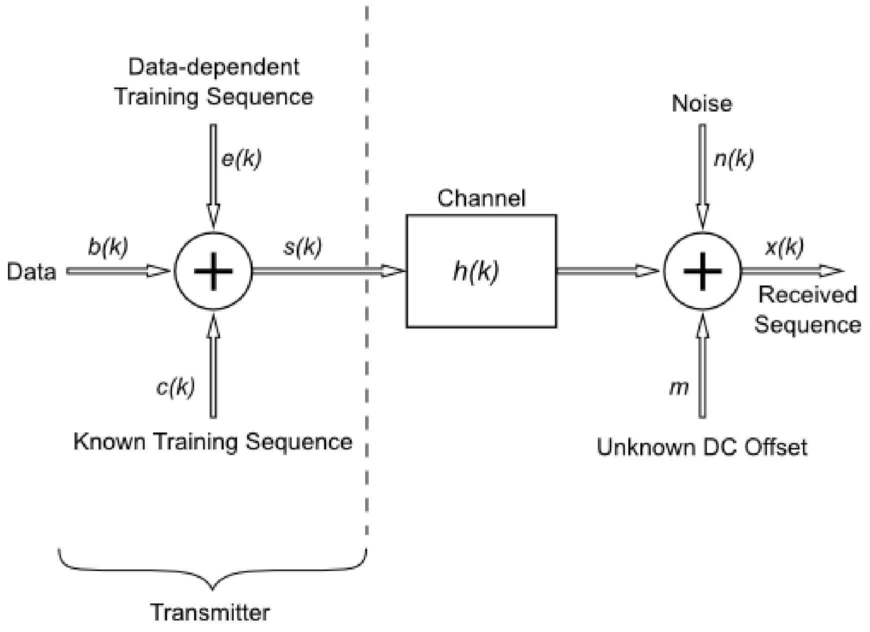


Figure 2.1: Discrete-time model for DDST transmitter and channel.

A mathematical representation of the received signal $x(k)$ for this model is shown in (2.3).

$$x(k) = \sum_{l=0}^{M-1} h(l)\left[b(k-\tau-l) + c(k-\tau-l)\right] + n(k) + m \tag{2.3}$$

- DC-Offset Estimation and Removal.

DC-offset arises from three sources:

- transistor missmatch in the signal path
- local oscillator (LO) signal leaking to the antenna because of poor reverse isolation through the mixer and RF amplifier
- large near-channel interferer leaking into the LO port of the mixer.

These effects may be reduced to a certain extent by a good circuit design, but they cannot be eliminated.

In fact, the DC-offset problem is of great importance to avoid a serious degradation of the performance in direct conversion receivers (receivers with an intermediate frequency equal to zero) to be seriously degraded. DC-offset is usually much larger than the RMS front-end noise, and may therefore significantly degrade the SNR at the detector. To remove the offset by ac coupling, the receiver will require impractically large capacitors, if the signal-bearing spectrum around dc is not to be sacrificed. However, the may be compensated with DSP-based self calibration [33].

Because of the use of first order statistics, the possible DC-offset caused by the non-ideal radio frequency receiver has to be considered and removed from the signal before the main stages of DDST channel estimation [3]. This is not the case when using second or higher order statistics, because the DC-offset can be estimated and removed in a simple way.

In the case of the DDST, DC-offset (m) can be estimated as a simple average of N samples y from the received data *x(k)* (2.4). Then, the estimated complex value can be removed by subtracting it from all the elements of *x(k)* [29].

$$y(k) = \left(\frac{1}{N_p}\right) \sum_{i=0}^{N_p-1} x(iP + j),\ j = 0, 1, \cdots, P - 1 \tag{2.4}$$

- Carry Frequency Offset Estimation and Correction

  Synchronism between the local oscillator of the receiver and the incoming signal carrier frequency (both signals having exactly the same phase) is a result that is generally difficult to achieve in practice. A phase error ($\Delta\phi$) causes a reduction in the amplitude of the output signal of $cos\Delta\phi$. Meanwhile, and error in frequency ($\Delta\omega$) shifts the signal by $cos\Delta\omega t$.

  Due to the lack of perfect oscillators, frequency drifts, Doppler shifts, and the variable distance between the transmitter and the receiver, practical pass-band systems always experiment a carry frequency offset [34] and [30]. There are two general techniques for carrier synchronization: the transmition of a pilot carrier, the use of a carrier recovery circuit.

  The work described in [30] presented three techniques for CFO estimation. All of them intrinsically exploit the periodic nature of the superimposed training sequence, but from different angles. In particular, this implementation uses the algorithm named *Maximization of Frequency Bins (MFB)*. When CFO is absent, the received sequence will be periodic. Then, the DFT of an N samples array from this sequence will only have *P* (the number of elements of the training sequence) bins with values different from zero. When a CFO is present, these bins are cyclically shifted in the frequency domain. Under these conditions it is possible to estimate the CFO as a value that maximizes a summation function.

- Training Sequence Synchronization Estimation

The training sequence C in DDST is added to the information sequence as an square matrix of dimensions $|PxP|$. This matrix also has another interesting characteristic: it is a *circulant matrix*.

A circulant matrix is an special Toeplitz matrix, based on an original vector, whose elements are rotated one position to the right on each of the matrix rows (2.5). They have an interesting property: they are diagonalized by a discrete Fourier transform, so equations containing them can be solved easily by a Fourier discrete transform.

$$\left\{\begin{array}{ccccccc} c0 & c1 & c2 & c3 & c4 & ... & cn \\ cn & c0 & c1 & c2 & c3 & ... & cn-1 \\ cn-1 & cn & c0 & c1 & c2 & ... & cn-2 \\ & & . & & & & \\ & & & . & & & \\ & & & & . & & \\ c1 & c2 & c3 & c4 & c5 & ... & c0 \end{array}\right\} \tag{2.5}$$

In the data-dependent superimposed training, the circulant matrix **C** is formed from the original training sequence, whose elements correspond to the first row of such matrix (c0, c1, c2, ..., cn). C is used several times along the algorithm, but in all cases what is operated is its inverse version C^{-1}.

When τ is known, it is said that there is *perfect training sequence synchronization* (perfect TSS) at the receiver, and thus it is very easy to extract an estimate of the channel from the data. In fact, the last $P - M$ elements from the product $C^{-1}y$ (where M is the number of taps of the channel) are all equal [30]. Nevertheless, it is almost a fact that perfect TSS in practical systems is not possible, as the receiver cannot know τ before it processes the incoming information.

After the analysis of $C^{-1}y$ when τ is not known, it is possible to note that this product is just a cyclic permutation of the case when perfect TSS is present. One of the possible permutations in the circular array is equal to the estimate supposing perfect synchronization, so the problem is reduced to obtain the knowledge of the correct permutation. Figure 2.2 shows a graphical sketch of the product when there is no knowledge of τ. In this small example, $P = 7$ and $M = 3$. The red bins represent the P-M equal elements from the product that can be used to determine the right permutation.

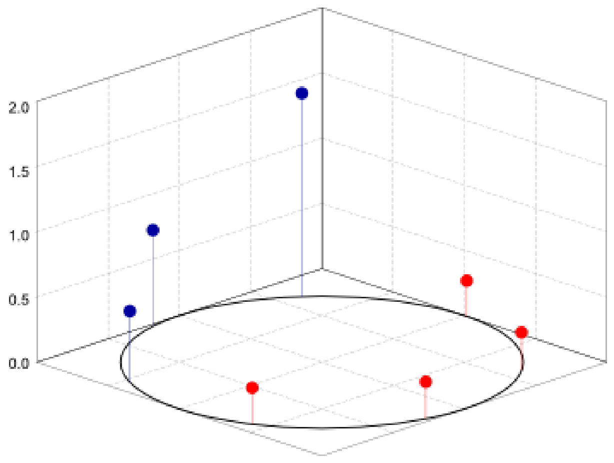


Figure 2.2: Graphical representation of $C^{-1}Y$ when τ is unknown.

- Block Synchronization Estimation

As it was mentioned before, DDST adds a second sequence to the information. As this sequence is different for each block, training sequence synchronization is not enough for a correct channel estimate, because taking data from two blocks for the estimation, the independence property between the information sequence and the training sequence is lost.

Block synchronization is also performed based on the particular structure of the channel's output cyclic mean vector. The data-dependent sequence e is also added to the information sequence as a matrix (E). There is an infinity of matrices E that can satisfy the conditions to be used in a DDST system. However, since E will behave like a perturbation of the data matrix, its power should be minimized [27]. The problem can therefore be solved using a cost function to determine the vector that minimizes the data noise, what indicates that the start of the block has been found. The obtained value is known as τ_p'

- Channel Estimation

  Once the values of both τ and τ_p' have been calculated, the channel estimate can be easily obtained by using the modified version of the $C^{-1}y$ product [29].

- Symbol Detection

  After the channel has been estimated, it is possible to remove the effects that it induced over the information signal. In the implicit training methods this procedure must be complemented or carried out along the training and data-dependent sequences removal.

  All the processing or filtering which is used to remove or reduce the interference effects (the ISI mainly) is known as *equalization*, and the block that addresses these questions is called an *equalizer*. Described in a different way, an equalizer is a filter designed to have an impulse response as equal as possible to the inverse of the impulse response of the channel.

As channels can change their characteristics continuously, an static filter is not an option for this kind of block. Many digital wireless systems use an specific kind of filter known as *transversal filter*, which is basically an adder of weighted coefficients or taps adjusted by an algorithm designed to carry out this task (figure 2.3). Each tap generates, with a certain magnitude, an "*echo*". All the added echos ideally cancel the effects caused by the ISI, subtracting the effects of the interference according with the information obtained through the channel estimation.

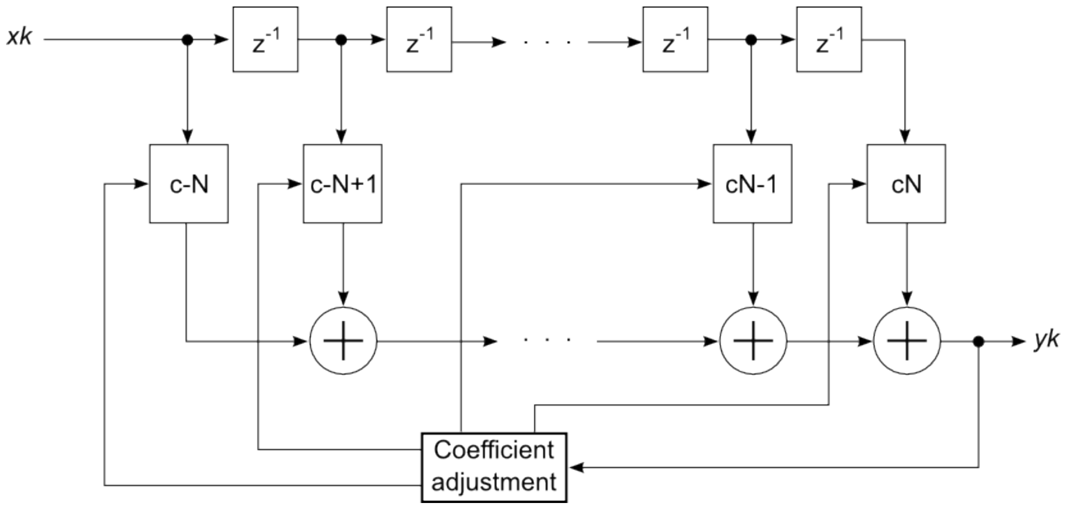


Figure 2.3: Transversal equalizer

With respect to the mathematical criterion behind an equalizer, there are two main solutions: the *zero-forcing* solution, which tackles the problem in a deterministic way, and the *Minimum mean square error* (MMSE or simply MSE) which uses an statistical approach.

– Zero-Forcing Solution

The zero-forcing or *Peak Distortion* criterion is simply defined as the minimization of the worst-case intersymbol interference (which is a performance index of the system) at the output of the equalizer [31].

This approach arises from the fact that it is not necessary to eliminate or minimize ISI with neighboring pulses for all t. All that is needed is to eliminate or minimize interference with neighboring pulses at their respective sampling instants only, because the decision is based only on sample values [32].

This causes the combined channel-equalizer response to become a purely δ-sample delay with zero ISI. ZF equalizers tend to perform poorly when the channel noise is significant and when the channel's transfer function has zeros near the unit circle.

– Minimum-mean Square Error

A more robust equalizer is obtained if the equalizer's tap weights are chosen to minimize the mean-square error (MSE) of all the ISI terms plus the noise power at the output of the equalizer. MSE is defined as the expected value of the squared difference between the desired data symbol and the estimated data symbol. To achieve the minimum MSE (statistical) solution, the processing starts with an overdetermined set of equations and hence a *nonsquare* $\mathbf{x}$ matrix, which then gets transformed to a square autocorrelation matrix, whose solution leads to tap weights that minimize the MSE. Most high-speed telephone-line modems use an MSE weight criterion because it is superior to a zero-forcing criterion; it is more robust in the presence of noise and large ISI [6].

Minimizing the sum of squared errors is an optimal approach when the probability distribution of the observation noise is a Gaussian distribution. Although this is hardly ever true, for any particular set of observations, the central limit theorem of probability theory indicates that this Gaussian assumption is a valid one for many cases.

The MMSE can be obtained from the inversion of the statistical correlation matrix or it can be calculated iteratively, using methods like the one known as *Gradient Descent Technique* [39]. Both the MMSE equalizer and the ZF equalizer are of a general infinite impulse response (IIR) form. However, adaptive linear equalizers are usually implemented as FIR filters due to the difficulties inherent in adapting IIR filters [8].

This DDST receiver implementation uses this last approach (MMSE) for the channel equalization, but before this process can be done, it is necessary to remove the training sequence c from the received signal. To achieve this, in the frequency domain, the DFT components of the signal at the pilot frequencies are set to zero.

The equalization can be performed in the frequency domain because the matrix that contains the channel estimate (H)is circulant. The inverse FFT of the product of G by z (2.6), where G is a diagonal matrix of $|P\ x\ P|$ with each $G(k) = 1/H(k)$, and z is the received signal after training sequence removal. u is known as *the rough data estimation*.

$$u = IFFT(G \cdot z) \tag{2.6}$$

Once this first stage of the symbol detection is ready, the algorithm reaches an iterative phase, where the objective is to find the closest element for each u(k) from a constellation points vector, according to (2.7). For example, when using QPSK, a value u(k)=0.627-0.921i will be in the fourth quadrant. It is important to mention that the Jw parameter is considered as extra additive noise, where J is a Kronecker product of a matrix of dimensions $|Np\ x\ Np|$ with all its elements equal to $1/Np$ by an identity matrix of dimensions $|P\ x\ P|$, and w is the result from the previous iteration.

$$w = u + Jw^{i-1} \tag{2.7}$$

According to the simulations that have been done with DDST, the first iteration of the process gives a performance comparable to that of the superimposed training method [3].

2.2 Implementations of Superimposed Training and DDST

2.2.1 Simulations

As DDST is an experimental method for channel estimation, there is not any commercial implementation nor a full prototype receiver for it. Until now, the great majority of the proofs performed correspond to software simulations, in which the main purpose is to compare the performance of the superimposed methods with the one of techniques like the TDMT, with respect to errors and noise tolerance.

Matlab is the most common tool used for these simulations. Unfortunately, Matlab, even though its multiple advantages, brings some problems when used as the reference implementation for a new technique:

- Matlab is slow compared to simulations programmed in languages like C or FORTRAN.
- Its programming sintaxis makes difficult to manually translate it to other languages.
- Many of the available functions hide a more complex set of operations. This can make very complicated the process of identifying the critic sections of the code.
- Matlab makes very easy to program code that is far from optimal, so a computational complexity analysis could not reflect the real burden of a technique under study.

2.2.2 Hardware-aided Implementations

The superimposed training and the Data-dependent Superimposed training have been implemented in both fixed point and simple precision floating point (32 bits). Resulting speeds from these architectures (specially the floating point) are in the order of kHz, so, even when they are usefull for error comparison, it is impossible to use them in commercial systems.

This is the first FPGA architecture to be used for the DDST digital receiver, so it is expected that it comes usefull as a tool to make visible the main obstacles for a method's hardware implementation, and to serve as a reference for future architectures toward the creation of an ASIC.

2.3 In perspective

Data-dependent Superimposed Training is a promising technique for removing the extra bandwidth required by common pilot sequences in wireless communications, but as ti can be seen, it presents several challenges for the designing and building of a commercial implementation. The next chapter will discuss the main technologies used for *Digital Signal Processing*, giving background information to understand the proposed *proof of concept* digital receiver and the reasons behind the election of its specific set of characteristics, over all the other available options.

Chapter 3

Hardware Approaches in Wireless Digital Communication Systems

Many methods used in wireless digital communications require the implementation of very computational intensive DSP algorithms. The available solutions for this purpose cannot perform such operations using a software only approach, as the achieved speeds of the implementations of the algorithms, using this method, are too low for the requirements of the majority of the systems needing them.

According to [35], the main criteria for DSP algorithms are:

- They involve intensive amount of data.
- They require multiple use of the same data.

- There may be feedback paths.
- Fast multiplications are required.
- Intensive intermediate data manipulation and address generation.
- Most of the algorithms are amenable for parallel computation.

This situation has forced the designers to develop alternative solutions based on hardware modules, coprocessors, or full architectures, which release the main processor from the most computational intensive operations. To implement these hardware architectures, several platforms have been used according to the systems constraints and to the advance of the available technology.

3.1 Available Digital Signal Processing Options

3.1.1 Microprocessors and Microcontrollers

As the majority of the microprocessors are designed to be used as general purpose devices, there are many DSP algorithms for which this option is not suitable, either because of the low processing speeds or because of the great amount of data to process. Moreover, when a microprocessor is used, other components have to be included (RAM and ROM memories, buses, peripherals). Thinking about this restriction, the idea of building a computer-on-a-chip led to the creation of the microcontroller device.

Microcontrollers include in the same chip memories for different purposes, peripherals, and input/output ports. They run in kHz or in the low MHz range. Nevertheless, its power consumption, price and implementation complexity are low, which make them perfect for many low power DSP applications where speed is not a priority.

Their program memory is usually reprogrammable, making them very flexible and suitable for testing and upgrade applications. Because of their low cost, the microcontrollers can be easily replaced in the case of failure, or when a system is going to be updated.

The speed problems of the microprocessors are the result from a trade-off between performance and flexibility. For example, PCs are designed to be used in a wide variety of applications, from office suites to computer aided design. Each of these applications have very specific and occasionally very different needs. Nevertheless, it is very expensive, and sometimes impossible to build a microprocessor that is optimal for hundreds or even thousands of different operations. The strategy of the general purpose processors is to work as good as possible with the majority of the applications, even if this means to run many of them in a less than optimal fashion.

3.1.2 ASICs

When optimizing an specific process or a set of processes is absolutely necessary, sometimes the only alternative is to build an specific architecture. These chips, known as *Application Specific Integrated Circuits* or ASICs, are very common in the DSP field, but their high performance also means a compromise: its development and initial build process is expensive, their flexibility is null, and there is no place for errors in the design once they are ready, so it is necessary to expend time and money in a test-and-correct cycle, assisted by a set of simulation tools.

3.1.3 DSPs

As microcontrollers cannot solve complex DSP algorithms because of their limited resources, and because of the cost and lack of flexibility of the ASICs, the programmable digital signal processors (PDSPs or DSPs for short) were introduced in order to solve the problem. These devices usually have the following characteristics [35]:

- Fast arithmetic units (specialized in multiply and accumulate operations).
- Multiple functional units.
- Parallel computation.
- Pipelined functional units.
- Proximity of storage.
- Localized data communication.
- High bandwidth buses.

There are three main categories of VLSI DSPs architectures:

- General purpose DSP chips

    - Floating-point DSP processors
    - Parallel DSP chips
    - RISC DSP chips
- core-based systems
    - Generic core processor
    - DSP controllers
- Application specific chips

Even though the components of digital signal processors increment the number of DSP algorithms that can be implemented in hardware architectures, it is also true that they are not flexible enough for many applications. As they were designed to accelerate mathematical operations, other possible needs from the algorithms are difficult to carry out, so it is very common to add external logic to the inputs or outputs of the DSPs in order to expand their capabilities.

3.1.4 CPLDs and FPGAs

In the late 70's, a company named Monolithic Memories introduced a kind of integrated circuits with the property of being programmable. Although very simple, the PALs (Programmable Array Logic) could be used to implement basic logic functions. A problem with the PALs is that the majority of them were programmable only once, but this action could be performed by the use of hardware description languages, nowadays the norm for circuit creation and programming.

The Complex Programmable Logic Devices (CPLDs) are integrated circuits evolved from the PALs, composed by more simple devices (simple programmable logic devices) with many elements in common with the PALs [37]. CPLDs feature larger amounts of gates and have the possibility of reprogramming them a high quantity of times. They are programmed with very mature HDL's, like VHDL and Verilog.

The Field Programmable Gate Arrays (FPGAs) exceed the number of gates of the CPLDs, and, even if the majority of them cannot function immediately on system start-up (a characteristic of CPLDs), they have other advantages, like their lesser granularity. FPGAs are composed by elements called logic blocks, that can implement not only simple logic functions but also more sophisticated behaviors, like complex mathematical operations. These devices could be considered ASICs, as they are, after all, specific application integrated circuits, but, according to [37], it is generally assumed that the design of a classic ASIC required additional semiconductor processing steps beyond those required for field-programmable logic (FPL) devices.

Of all the hardware solutions mentioned to this point, the FPGAs are the most flexible of all them. They can be used to model a great variety of DSP algorithms, consuming little power and still running at high frequencies (architectures in the range of hundreds of MHz are pretty common). As structures like multipliers implemented with logic blocks give inferior performance to ASIC solutions, FPGAs display embedded devices like DSP blocks (a series of multipliers and adders to implement a very common operation of the digital signal processing known as MAC or *multiply and accumulate*), memory elements (from simple flip-flops to complete dedicated memory blocks), and even microprocessors, like the PowerPC exhibited by a Xilinx family of FPGAs. In fact, in the world of FPGA there are two main types of microprocessors: soft-core and hard-core.

Hard processors are common physical devices mounted on the same board that the FPGA, or even integrated inside this last one. The most known example is the PowerPC included with the Virtex FX series. On the other hand, soft processors are implemented using logic indicated by a hardware description language, that is then synthesized to a device that has programmable logic, as the FPGAs or the CPLDs. Examples of this branch of microprocessors are the MicroBlaze of Xilinx and the NIOS II from Altera. As the code that describes them can be modified by the designer or by an API (it depends on the openness of the processor architecture), they are usually more flexible, even if they commonly run slower than hard processors.

3.2 FPGA Approaches

FPGAs allow the solution of DSP problems in ways that are very characteristic of these devices. Altough a system built for an FPGA can look very similar to a common ASIC or to a computational system, this approach rarely gives the performance of their counterparts. To analyze the way in which FPGAs are used to solve DSP problems, their characteristics can be seen from several angles.

3.2.1 Hardware Architecture

An FPGA architecture can have several differences with a traditional architecture of a computational system. These discrepancies arise from the freedom and flexibility existing when programming for the logic blocks of the FPGAs. Next, some of the most common characteristics of the FPGAs architectures are explained.

Parallel Processing

Parallelism is a technique used in both software and hardware implementations, where two or more operations or groups of operations are performed at the same time. In a software program, this means to execute several instructions simultaneously, while in hardware it refers to the replication of certain structures so two or more operations can be executed at the same time. For example, to include two adders that can solve the sums $A + B$ and $C + D$ in a parallel fashion, instead of using the same adder for all the operations. The difference between both approaches can be seen in figure 3.1. When processing data in a serial fashion, like in figure 3.1(a), a small quantity of elements are needed in order to solve an operation. In the figure, only two registers can be seen, one output register, and the adder for the operation. Nevertheless, data is processed slower. First, $A+B$ is computed and only when the result is ready (right part of the figure), $C+D$ can be performed. On the other hand, parallel processing requires more hardware (like the two adders of this example), but it computes its data faster. In the figure 3.1(b), $A+B$ and $C+D$ are processed simultaneously, so, at the next time, the input registers are ready to receive four new values.

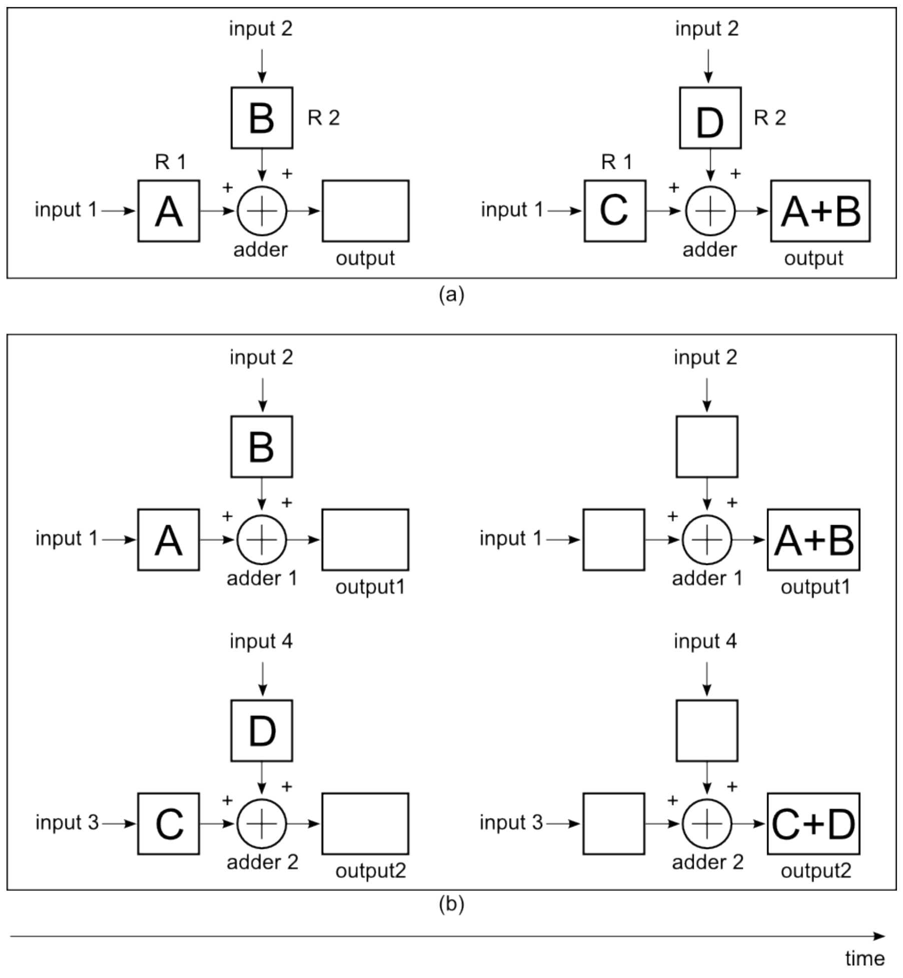


Figure 3.1: (a) Serial execution of two sums; (b) Parallel execution of two sums.

Parallel processing implies a trade-off between FPGA area and speed or maximum frequency of operation. Nevertheless, the majority of available FPGAs have an enormous amount of gates (millions) and the use of parallel structures can result in a performance increase of several orders of magnitude.

Pipelining

Pipelining is an implementation technique in which multiple instructions are overlapped in execution [38]. For example, lets suppose a situation where a set of one hundred data is present. Each datum x_k, with k from 1 to 100, has to pass for the following process:

1. $a_k = x_k + 2$
2. $d_k = c_k >> 1$, where "$>> 1$" implies a logical shift to the right.
3. $c_k = b_k * 5$
4. $e_k = d_k - x_k$

So four operations have to be performed to obtain all the e_k corresponding to the final results: an addition, a shift, a multiplication, and a subtraction. In a traditional sequential execution, supposing that each operation can be performed in one cycle, a system will need 500 cycles to process all data. Nevertheless, it can be seen that when the first a_k has been obtained and b_1 is going to be computed, the adder that performs $x_k + 2$ is idle, while it could be used to compute a_2. Figure 3.2 shows a graphic of the consumed time on the first three data for both approaches. If each operation takes more or less the same amount of time, the speed up of the pipelined execution is almost equal to the number of stages in the pipeline, as long as enough operations are performed. From figure 3.2(b), it can be seen that pipeline does not improve the time needed to compute each datum, but as the number of operated data increase, a performance increase of almost four times can be seen.

Pipelining improves performance by increasing operation throughput, as opposed to decreasing the execution time of any individual operation [38]. Throughput is a more significant metric than single operation execution time when a high amount of operations need to be performed.

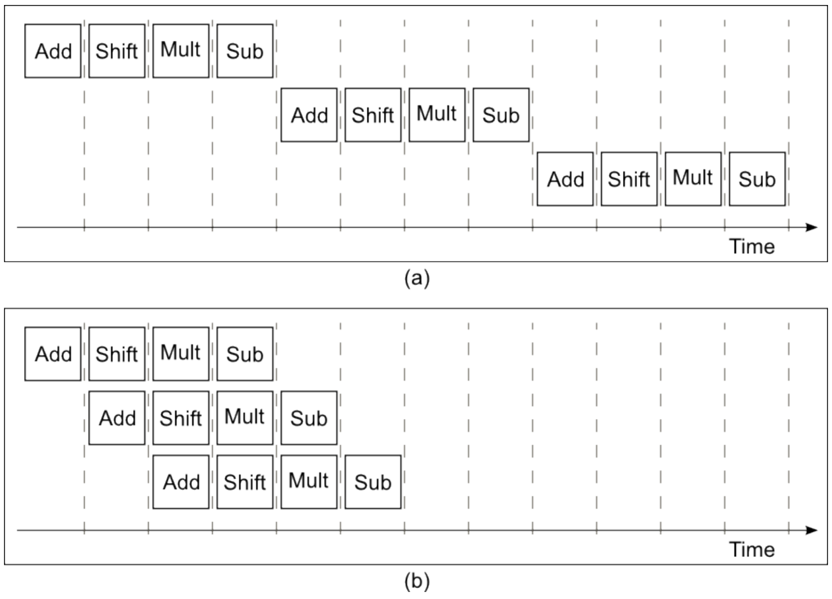


Figure 3.2: (a) Sequential execution; (b) Pipelined execution

Pipelining is extensively used in DSP solutions due to the intrinsic dataflow regularity of the algorithms. Its principle can be applied not only to PDSPs, but also to FPGA designs, at little or no additional cost since each logic element contains a flip-flop, which is otherwise unused, to save routing resources. With pipelining it is possible to break an arithmetic operation into small primitive operations, improving throughput significantly when the quantity of data to process is high.

Neither parallel processing nor pipelining are techniques exclusive of PDSP or FPGA architectures. Available CISC microprocessors usually have a large amount of pipeline stages, while RISC processors usually display a reduced quantity of them, a result of the efforts made in order to maintain its reduced instruction set and high frequencies of operation. Last generation processors can also perform several instruction in parallel, due to the existence of two or four cores in each chip. Notwithstanding, these methods usually look different in FPGAs because of the kind of applications implemented in them. For example, parallelism in PCs is usually performed at a process level, while in FPGAs what is usually operated in parallel are the operations like additions, multiplications, or logical shifts.

Dedicated Modules

When an specific operation in a DSP algorithm is required, there are two options, at least when the solution is of a software-only kind. For example, if a multiplication is necessary, it can be performed directly only if the main processor has an embedded multiplier available. Other way, the operation will have to be performed as a series of sums. A dedicated coprocessor can be used if the system has that feature, but this is rarely the case. FPGAs are different in which their free logic blocks can be programmed into dedicated modules fit to perform a very specific action or set of actions. This way, it is possible to add, for example, a dedicated FFT module when frequency analysis is necessary.

Many kinds of dedicated modules can be added to the rest of the FPGA logic, as long as the area constraint is met. When combined with a soft or hard-core processor, the control necessary to manage the dedicated modules can be reduced to a set of software instructions, so in fact such module can be considered a coprocessor used in a very similar way to those found in classic computational systems.

3.2.2 Embedded Systems (SOPCs)

Many FPGAs are offered in boards that also present RAM and ROM memories, peripherals and expansion slots. The FPGA can be programmed to access and control these elements, converting the board into a *system on a chip* or SoC. As SoCs usually cannot be modified once they are build, some designers prefer to give different names to the solutions based on an FPGA, like Altera, that calls their system *system on a programmable chip* (SoPC), to remark the capability of these devices for being customizable to debug or improve them.

Embedded systems are a fast growing tendency in the computational and electronic devices field thanks to the increase in the number of transistors per square millimeter, along with the necessity of smaller and power efficient systems for areas like mobile communications and personal computing. Even though the majority of end user embedded systems use classic ASICs or mobile microprocessors (also considered by many as ASICs), the manufacture process usually implies the use of an FPGA SoC for testing and debugging, both in hardware and software parts of the final system, because of their flexibility, capacity, frequencies of operation, and reconfigurability. This process is usually known as *emulation*. In fact, the two main FPGA manufacturers (Xilinx and Altera) offer direct FPGA to SoC convertions: Xilinx solution, known as EasyPath, only removes the FPGA reconfigurability capacity, while in Altera's option (HardCopy) the programmable routing fabric is replaced by fixed wire interconnect.

3.2.3 Hardware-software Coprocessing

In spite of the FPGA advantages, it is undeniable that they cannot compete in many fields with the general purpose microprocessors computer systems (like the PCs) due to the huge amount of available memory capacity and high speed processing of these last ones. Nevertheless, the PCs are also outperformed in applications where parallel or pipelined processing of large amount of data are required, or in those where price, size and power consumption constraints are very strict. Hardware-software coprocessing is a series of techniques that exploit the advantages of both options, trying to give a better solution to specific problems:

- Very complex operations with many data dependencies

  The existence of few data dependencies between operations allows the implementation of parallel structures and efficient pipeline processing in FPGA hardware architectures. Both techniques normally require a very simple control, as the data flow "rules" how the different stages of the process occur, like in a systolic array or processor. When this is not the case, overall when the process requires the completion of complex operations like transcendental functions (exponentials, logarithms, trigonometric functions). A situation like this usually requires a very complex hardware control that, along with the necessary lines of interconnection between the system elements, can increase the occupied FPGA area drastically, so a problem of both space and a complex design is present. In a hardware-software coprocessing, the FPGA can only contain the hardware modules necessary to perform the operations required by the algorithm, while the control is implemented by a software program running on a traditional computational system. In fact, a hard or soft-core processor can be used to control the hardware modules, so the whole system can be implemented in the FPGA board without the need of external devices.

- High amounts of input data to process using the same operations

  When the same operation has to be performed over a large quantity of data, the best performance is almost always obtained by the scheme known as SIMD (*simple instruction-multiple data*), where special instructions order the system to apply the same operation over a range of data. For example, *multi(10, 0x0010,0x01ff)* can reffer to an instruction where every datum from address 0x0010 to 0x01ff is multiplied by the constant 10. Nevertheless, the majority of the general purpose microprocessors do not implement this kind of operations because they are not usefull when the objective is to be as efficient as possible with the bigger amount of applications while maintaining a low cost. There are, however, certain applications where a SIMD approach gives very good performance, like in audio analysis and in image processing. In fact, common GPUs of the video cards implement this approach effectively.

  The mentioned applications can be accelerated by implementing, in an FPGA, those operations that work better in a SIMD scheme. For example, a PC application can search into an image database and then pass an specific picture to the FPGA, that applies a series of digital filters and then returns the result to the PC. This approach can decrease significantly the total execution time of a great variety of DSP applications. This is one of the main reasons why there are several FPGAs board that can be interfaced to a PC through an expansion slot like the PCI.

- Necessity of very large amounts of memory to store output data

  Even when a full hardware FPGA implementation can have the speed, low power consumption, and reduced area required by a DSP application, the amount of output data can be so large that it is impossible to store it in the FPGA memory blocks and even in the storage devices included in the board (like flash memories and SD cards readers). In these cases, the high capacity of hard drives can be a suitable solution, but interfacing it to the FPGA can be complicated, also increasing power consumption and necessary physical space. A software interface in a PC can be easily programmed to communicate with the FPGA through a variety of ways (parallel communication, ethernet protocol, wireless link) to store in a local or remote hard drive all the produced data.

3.3 Wireless Communications Systems Components

The development in the field of wireless communication systems is characterized by a shift from purely analog techniques to those employing digital or hybrid approaches. The advantage of digital communications techniques are well established both theoretically and practically, and include more efficient use of bandwidth, greater immunity to impairment due to noise, and lower power dissipation. In addition, digital communications techniques can exploit the relentless advance of digital integrated circuit technology to realize improved functionality at lower power dissipation and cost [33].

The trend of wireless communications industry is to move to higher levels of integration in order to meet the aggressive cost targets associated with the commercial/consumer market. Due to this fact, the majority of the available hardware solutions for wireless communications are of the monolithic type (all elements in a single chip), but it is still possible to make a distinction between the different implemented components.

3.3.1 RF Transceiver

In the portable wireless communication industry, it is well known that the design of the RF transceiver is the key element that determines the cost, the size, and the usefull battery life of the equipment, as well as how the equipment is used. RF functions of a typical transceiver are now being integrated at increasingly high levels onto a single semiconductor die, much the same way that digital IC technology is integrating more and more logic gates on a typical board. A common IC circuit of a digital transceiver usually aspire to include the components listed bellow:

- Duplexer

  A duplexer is a device that allows an RF system to use only one antenna for both transmission and reception. This is the reason why it is also called duplexer switch, at it isolates transmitter from receiver as the situation requires it. Commercial duplexers available for cellular telephone industry use have the largest physical volume of all other RF components in the transceiver.

  Digital duplexers are being introduced to the wireless communication market by companies like Sharp and Panasonic, but there are still many devices in which duplexing is still a fully analog task.

- Amplifier

  Low noise amplifiers (LNA) amplify the magnitude of weak signals received by the antenna of the transceiver. They are analog devices that are not energy efficient. Current low noise amplifier components, either in terms of discrete low noise transistors or low noise amplifier blocks, have very little margin for trade-off among LNA gain, receiver sensitivity and intermodulation protection.

- Mixer

  A mixer basically multiplies a signal by a sinusoid, shifting it to both a higher and lower frequency, and selects one of the resulting sidebands. They are almost always implemented as analog circuits based on diodes or transistors, like the FETs.

- Filters

  Among all the components of the transceiver, the filter designs are the most mature ones. Filters basically allow the elimination of unwanted frequency components of a signal, performing tasks as noise reduction and isolation from unwanted RF transmitted signals. Although analog filters use is still very common, digital filters are widely used because of their high performance and low cost. They can be integrated into the DSP processor of the device and even programmed when this capability is offered.

- Frequency Synthesizer

  These devices can generate signals between a defined range of frequencies having as their base a fixed oscillator. They can be generally categorized into direct and indirect architectures. Direct frequency synthesizer architectures utilize no feedback to generate the appropriate output frequencies. These direct synthesizers can be further divided into analog and digital approaches. Indirect frequency synthesizers rely on feedback, usually in the form of a *phase-locked logic* (PLL). As with direct techniques, indirect ones can be implemented in both digital and analog forms. Purely digital PLLs are often implemented for clock generation on digital VLSI chips.

3.3.2 DSP

DSP functions in wireless communications devices are usually performed by either a dedicated chip or by the main processor. When a DSP algorithm has to be performed several times or almost in a continuous way, specialized DSP ASIC are preferred, while in the case of rarely used functions, the microprocessor is used, as it implies no extra cost to the whole system.

The complexity of DSP systems grows at an ever-increasing rate while the implementation of these designs must meet criteria like minimum cost and a short time to market. Most of the designs for DSP systems result in fixed-point implementations, due to the fact that these systems are sensitive to power consumption, chip size, and price per device. Fixed-point realizations outperform floating-point ones by far with regard to these criteria. In fact, the resulting gap between a floating point prototype and the fixed-point implementation represents one of the major bottlenecks in today's digital designs.

The current trend towards hardware intensive signal processing has uncovered a relative lack of understanding of dedicated VLSI architectures. Many hardware-efficient algorithms exist, but these are generally not well known due to the predominance of software solutions employing general-purpose DSP chips. Unfortunately, algorithms optimized for this reprogrammable devices do not always map nicely onto specialized hardware. Therefore, current work in the field of VLSI signal processing is focused on the joint study of both algorithms and architectures for the custom implementation of DSP systems, exploiting the iterations between the two to derive efficient solutions and thereby bridging the gap between system and circuit level design [36].

3.4 In perspective

Now that the most important hardware devices used for DSP have been described, it could be easier to understand the election of the FPGA as the base of the system that will implement the DDST algorithm. Moreover, the explanation of the usual techniques that are commonly used in reconfigurable devices will make more evident the reasons why they are used in certain ways in the architecture. For example, the trade-off of parallel processing with respect to the speed and used area explains by itself the motivation for using this technique when the most time demanding stages of the DDST method, as long as the amount of used logic of the FPGA would not be excessive.

Chapter 4

DDST Digital Receiver

This chapter describes the architecture of the proposed Data-dependent Superimposed Training digital receiver, along with some background information to understand the techniques and tools it is based on. The chapter starts with a short explanation of the tools used to create the system, and then it mentions some basic components used to explain more deeply the implementation from both the hardware and the software point of view.

4.1 Tools Used to Create the Architecture

All the tools used to build the DDST receiver are from the Altera developer. The reason behind this decision is based on the focus that this company puts on the development of systems that combine both hardware and software, which is the approach that was decided the more adequate to build the prototype for the receiver.

4.1.1 Altera Quartus II Development Software

This is the main design tool offered by Altera. It allows the design, synthesis, and FPGA programming of the hardware part of the architecture. It is possible to synthesize the designs to be speed or small area oriented, simulate the systems before “downloading” them to the FPGA, between other functions. It is possible to create a system using a hardware description language (VHDL, Verilog, AHDL) or through the available schematic tools available to the user.

4.1.2 SOPC Builder

SOPC means *System on a Programmable Chip* and it refers to the construction of an electronic system, usually a computer (microprocessor, memory, peripherals, and external interfaces) on an FPGA. It is a term coined by Altera, but it describes perfectly what the design of embedded systems using their tools means. At the end of the process, the result is an specific system which uses internal and external resources of the board that contains the FPGA. Figure 4.1 shows an example of SOPC.

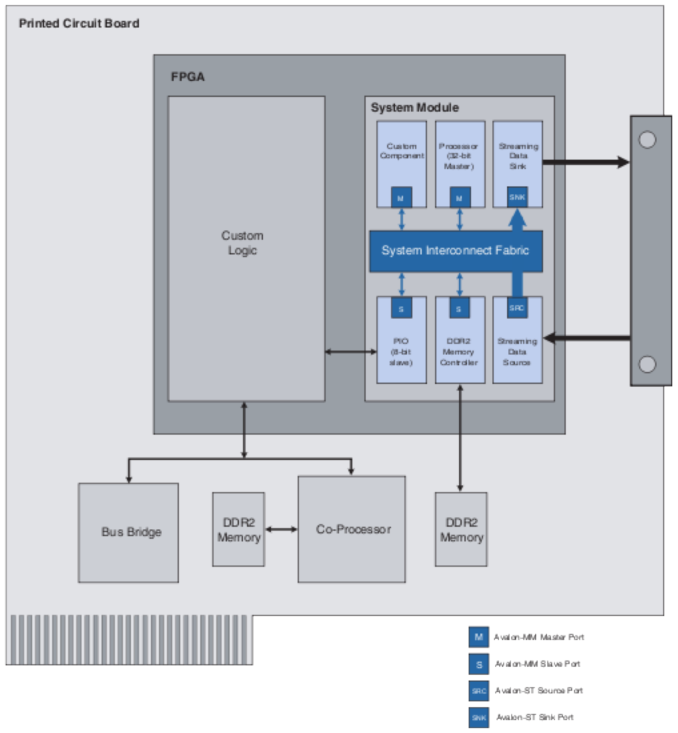


Figure 4.1: SOPC example

The tool used for the creation of these systems is the SoPC Builder. It is designed to create complex systems by just selecting their components from the set offered by Altera, or by adding custom components (custom peripherals) created by the user through a hardware description language. Once the system is ready, SOPC Builder generates all the necessary VHDL or Verilog files to interfacing it to the rest of the project and synthesize it with Quartus II.

4.1.3 C2H

The Nios II C-to-Hardware Acceleration Compiler is an Altera tool that can convert C language subroutines into hardware accelerators. Even though the designer does not have full control over the process, C2H can improve some of the architecture's designing and implementation time, creating automatically the necessary control for very complex algorithms and techniques, like those usually found in DSP. It is not the perfect C-to gates compiler (it was not created for that), but it integrates naturally to a SOPC, allowing a fast design, implementation and update of performance-critical functions.

4.2 Components of the Architecture

4.2.1 Nios II soft-core Embedded Processor

A Nios II processor system is equivalent to a microcontroller or "computer on a chip" that includes a processor and a combination of peripherals and memory on a single chip. The term "Nios II processor system" refers to a Nios II processor core, a set of on-chip peripherals, on-chip memory, and interfaces to off-chip memory, all implemented on a single Altera device. Like a microcontroller family, all Nios II processor systems use a consistent instruction set and programming model [41].

Figure 4.2 shows the block diagram of the NIOS II processor. It is a soft-core microprocessor with a 32 bits RISC architecture and the possibility of including a *memory management unit* (MMU) to support operating systems that require such characteristic.

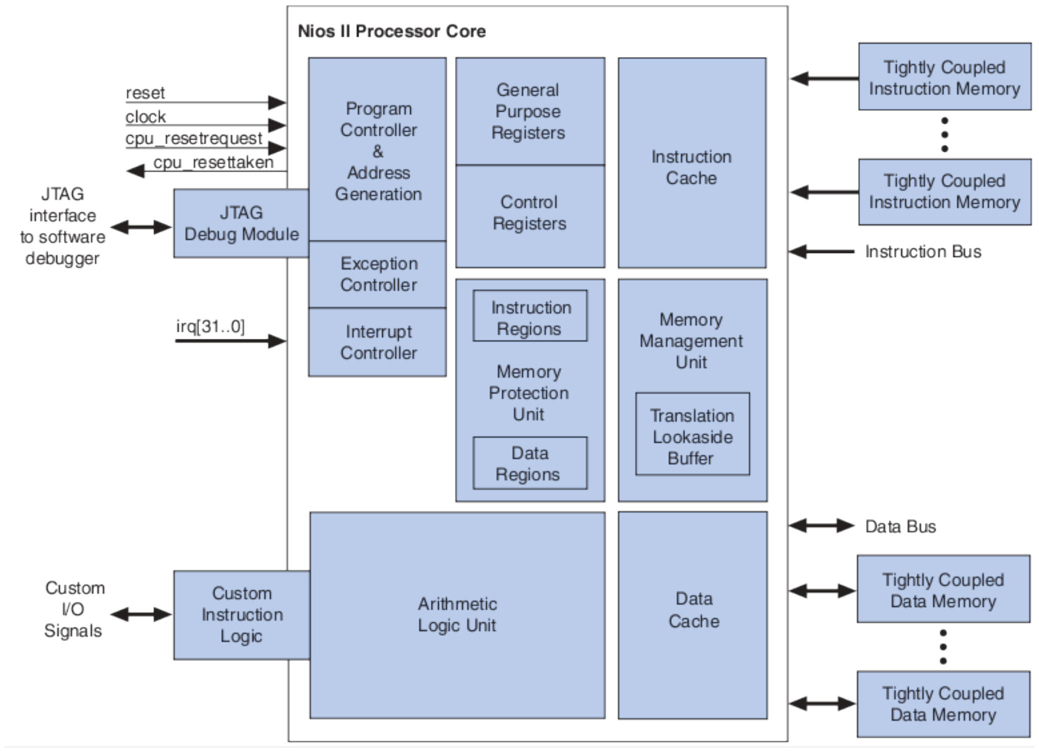


Figure 4.2: NIOS II block diagram (taken from [41])

Its main advantage over similar processors is its flexibility and configurability capacity, that allows not only to use a great variety of included peripherals, but also to create custom peripherals that can then be easily interfaced to the processor. NIOS II flexibility even allows to add new instructions to the instruction set. This is performed by hardware modules that are connected directly to the NIOS II ALU. Both the custom peripherals and the custom instructions can be used to a relatively transparent interface in a C language program, or even in assembler.

When instantiating a NIOS II processor in the SOPC Builder, Altera offers three core versions of it:

- NIOS II/f: This is the “fast” NIOS II core. Is is programmed for speed, so it will have the best performance while sacrificing FPGA area.
- NIOS II/s: The “standard” core. It offers the best balance between performance and area.
- NIOS II/e’: The “economy” core. It focuses on area saving, so many features will be unselectable, and performance will be lower.

4.2.2 SDRAM and On-chip Memories

When working with an approach like the one used on this architecture, there are several options for data storage:

- SRAM
- SDRAM
- Flash memory
- External memory devices (USB memories, SD memory cards)
- Registers inside the FPGA fabric (memory blocks)
- Microprocessor internal registers *

- On-chip memories

* These registers use the same memory blocks that the ones created with a hardware description language; they are differentiated only because of the way in which each one is accessed in the embedded system.

Every type of memory has their advantages and disadvantages, as the SRAM and SDRAM, which provide a “big capacity” but have a high latency. The opposite example are the internal registers of the microprocessor, that can be accessed immediately but can store an small amount of information. An additional problem with this option is that they cannot be accessed directly by other hardware modules and, in fact, it is impossible to do it from the program unless assembler language is used. Otherway, they are hidden to the designer or to the programmer. A more versatile alternative would be a medium capacity memory which could be accessed by all modules in the systems and, if possible, with a very low latency.

The on-chip memories, structures that allow the transparent management and use of several memory blocks, have a latency of 1, the lowest available. Low latency reduces the number of cycles needed to obtain, operate, and store a datum or a group of them. Fixed latency means that the system does not need to access the memory sequentially to achieve the highest throughput.

In the Altera NIOS II IDE, on-chip memories can be used as if they were part of the general data memory by just using a pointer to its assigned address, inside a typical C language program, that then will be compiled for the soft processor architecture.

4.2.3 Avalon Interconnect Fabric

Contrary to the majority of *Systems on a Chip* and common computer systems, Altera does not use a conventional bus scheme. Their systems feature a switch interconnect fabric which bypasses bus contention in most applications and gives a higher-performance pipe between processors and peripherals. This improves the DDST implementation execution time and prevents the necessity of extra control in both the software and hardware parts of the SOPC.

4.2.4 Hardware Coprocessors

The architecture uses three custom hardware coprocessors to accelerate the critical sections of an optimized version of the DDST algorithm. Two of them were programmed using pure VHDL, while the third one (the FFT accelerator) was created through the use of the C2H tool. All these accelerators will be explained in detail in the present and following chapters.

4.3 Paradigm of the Architecture

A pure VHDL or Verilog implementation of the DDST architecture results in a very complex control, and in a logic that cannot fit on the majority of FPGAs without sacrificing speed for resources usage. The alternative proposed in this work is a SOPC that runs a series of C programs, but leaves the most *computer intensive* or *memory demanding* operations to special hardware accelerators. A very general architecture of the system can be seen in figure 4.3.

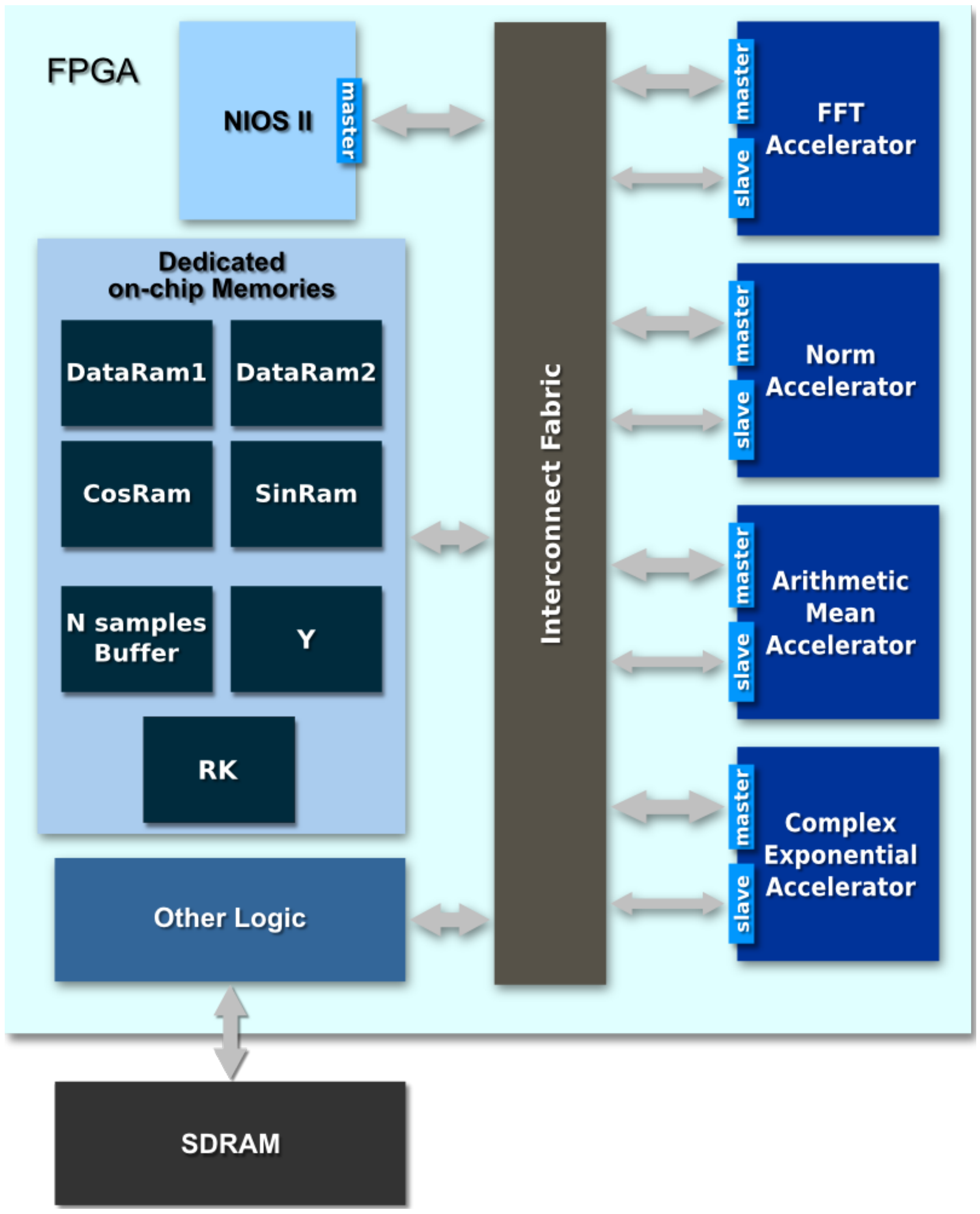


Figure 4.3: DDST architecture

Altera's SOPCs have a main difference with respect to other embedded systems based on FPGAs: the avalon interconnect fabric. This is a nonblocking interface, created by the **SOPC Builder** tool, that interconnects all the components in the system and permits multiple simultaneous master-slave transactions, while still requiring minimal FPGA resources. It replaces the traditional *shared bus* of usual electronic systems.

There are two kinds of ports that can access or be accessed by the Avalon: the slave and the master. Slaves are used to receive signals from other components of the system so they can be controlled. Meanwhile, masters can manage other components and perform actions like doing a memory read or write. In *SOPC builder*, a master can read or write up to 1024 bits on each memory access and not only can they communicate with on-chip memories, but also with any other memory device in the system. All that is needed is the base address of such memory and the existence of a controller for this last one. Those controllers are usually provided by Altera, like in the case of the SDRAM.

The four accelerators in Figure 4.3 will be discused in chapter 5, but at the moment it can be said that they are activated by the processor through their slave ports. They can perform memory accesses using their master ports and, after their work is finished, they indicate it with a signal that can be read from the processor using the slave port again. Their operation is hidden to the user by a series of C functions that read from and write to the slave port. The result of each of the accelerator processes is stored in the on-chip memories **DataRAM2**, **Y**, and **RK**.

At the moment the only device outside the FPGA that is used is the SDRAM, but it is scheduled to add the use of the ADCs to directly digitize the received data.

4.3.1 Arithmetic of the Architecture

As the C program can be considered the algorithm's implementation core, it was designed maintaining always standard data lengths, that is, 8, 16, 32 and 64 bits types of data. Moreover, the program uses floating point arithmetic as little as it is possible.

4.4 Block Diagram Description

The Data-dependent Superimposed Training involves complex mathematical operations in both the time and frequency domains. Although this work describes these operations as they are executed in the different stages of the algorithm, the reason of why they work in that defined way is out of the scope of an implementation as the one discussed here. This is the main reason why the DDST is described by the use of its block diagram and not directly by the steps involved in the algorithm. Nevertheless, some clarification is given when needed.

The figure 4.4 shows the block diagram of the DDST algorithm. This section will describe each block from two angles: first, the mathematical operations necessary to realize each block's functionality, after that, the changes applied in a software-only approach to reduce the execution time of that part of the algorithm. If any hardware module is used to accelerate part of the operations performed by the block, it is also mentioned, but a deeper explanation of such coprocessors will be given in the next chapter.

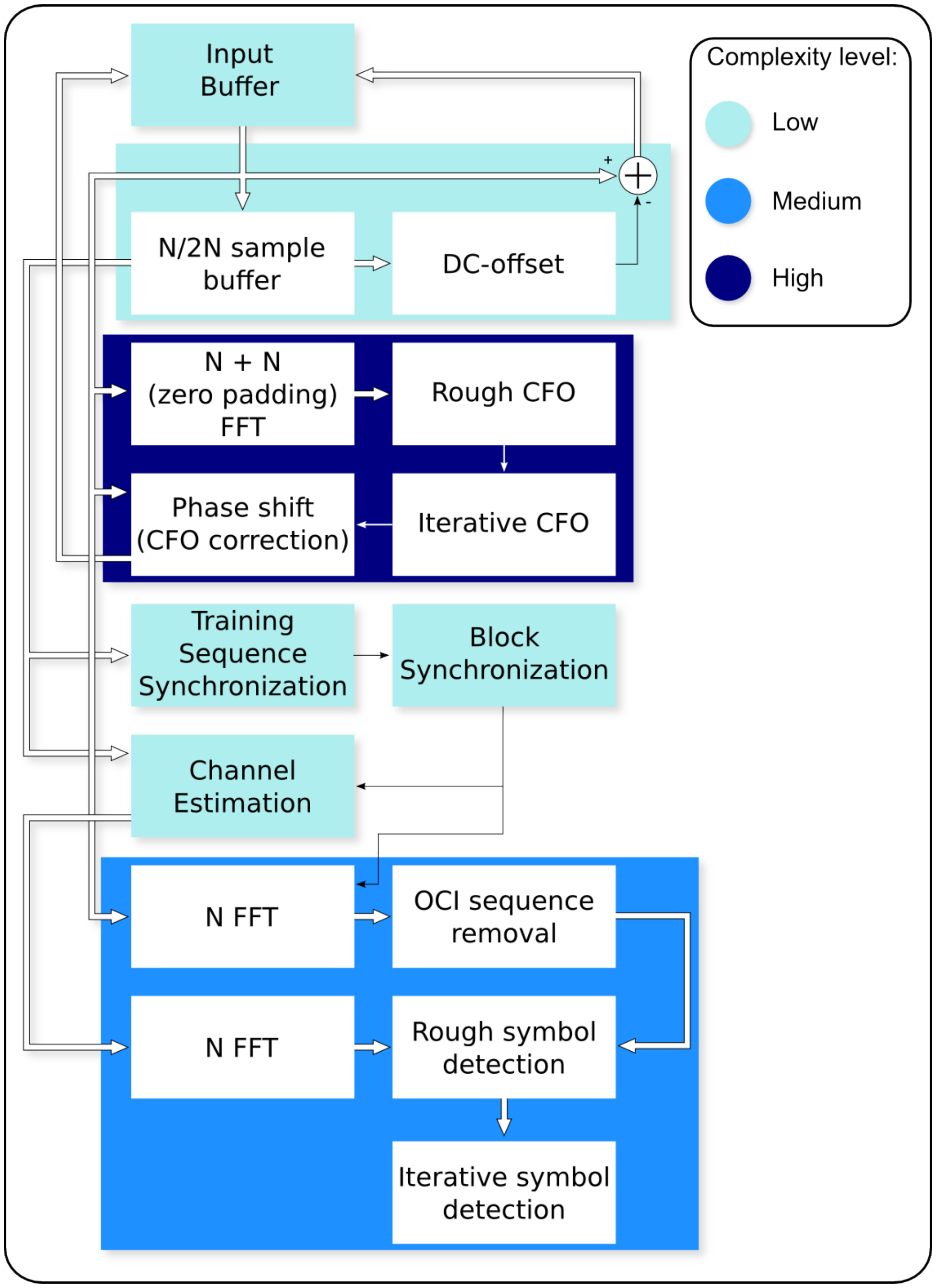


Figure 4.4: Descriptive block diagram of the DDST algorithm

4.4.1 Input Buffer

All the information that is received has to be saved, as the system needs to wait for the total channel estimation before the delivering of the transformed data. The input buffer block only stores the received data and the modified versions of them as they are changed by the DDST receiver algorithm steps.

- Mathematical operations: No mathematical operation is performed as all this module do is to store the received data.
- Software adaptations: Because of the huge amount of data, this block is implemented in the general data memory.
- Hardware modules: No custom logic or available hardware modules are used for this part of the system.

4.4.2 N/2N Samples Buffer

Along the algorithm process, there are some steps that require to work with a subset of the total amount of received data. In fact, even if the diagram of figure 4.4 shows this block as part of the DC offset estimation and removal stage, it can also be seen that it is used in other stages, like in the training sequence estimation. Not all processes need the same number of samples, but it would be a waste to create buffers for all of them. This is the reason why the block is called N/2N samples buffer: it is a memory structure that can store up to 2N data, but it can also be used by those steps that require a smaller amount of samples.

- Mathematical operations: At this point all that is needed is to store part of the data so no complex operations are performed.
- Software adaptations: The data are not stored in the general data memory, but in one of the dedicated *onchip memories* discused before.
- Hardware modules: the *onchip memory* can be accessed directly by hardware modules containing an *Altera avalon* master port, but this block does not use any custom logic.

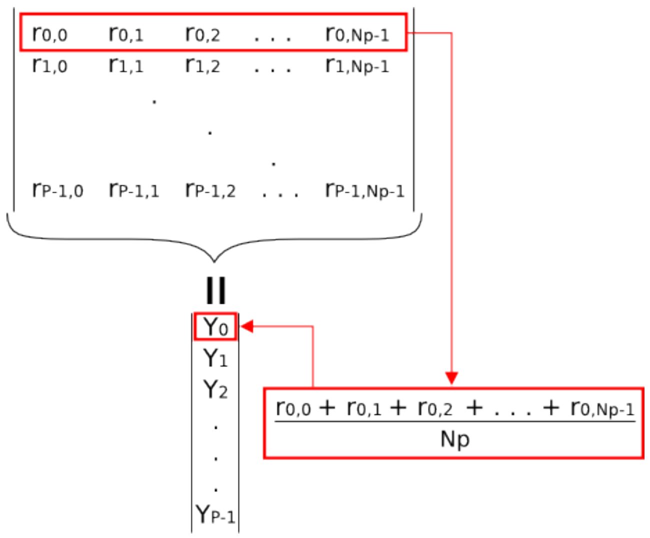
4.4.3 DC-offset Estimation and Removal

A deep analysis of the DC-offset estimation stage results essential for the design of a DDST receiver as optimal as possible, due to the fact that some of the necessary operations performed are repeated several times along the rest of the algorithm with little or no changes.

From all the operations required by this estimation, the most time demanding of them is the matrix multiplication, overall as the value of **P** grows. Nevertheless, the acceleration of this operation is well studied and basically requires one thing: to multiply, in a parallel fashion, as many elements of the matrices as possible. As the amount of available embedded multipliers is limited, this operation was left as a software-only one, turning all the attention to the other steps of the dc-offset estimation stage.

- Mathematical operations:

  1. The **N** stored data are redimensioned or regrouped as a matrix of dimensions $|PxNp|$, where Np=N/P. The matrix's columns are then summed in a row fashion, as shown in figure 4.5 and each of the resulting P data is divided by Np. In other words, the arithmetic mean of each row of the matrix is being obtained. The result is known as the Y vector that was mentioned before.



$r_{0,0}$ $r_{0,1}$ $r_{0,2}$ . . . $r_{0,Np-1}$
$r_{1,0}$ $r_{1,1}$ $r_{1,2}$ . . . $r_{1,Np-1}$
$r_{P-1,0}$ $r_{P-1,1}$ $r_{P-1,2}$ . . . $r_{P-1,Np-1}$
=
Y_0
Y_1
Y_2
Y_{P-1}
$\frac{r_{0,0} + r_{0,1} + r_{0,2} + ... + r_{0,Np-1}}{Np}$

Figure 4.5: Arithmetic mean of resulting matrix

  2. Once the Y vector has been obtained, the process reaches an iterative phase:

     (a) The inverse matrix of C is multiplied by the vector Y, resulting in a vector CY.

     (b) The last P-M elements of CY are taken, where M is the number of taps of the channel. With this new matrix the complex number $\widehat{\widehat{m}}_{T'}$ is calculated as:

$$\widehat{\widehat{m}}_{T'} = \frac{1}{P-M} 1_{1x(P-M)} (C_{T'}^{-1}\hat{Y})_{[P-M]}. \tag{4.1}$$

(c) $\widehat{\widehat{m}}_{T'}$ is used to calculate the scalar $\widehat{\tau}$ (4.2), which basically is the norm of the rest $(C_{T'}^{-1}\hat{Y})_{[P-M]} - 1_{1x(P-M)}\widehat{\widehat{m}}_{T'}$. This value is stored.

$$\widehat{\tau} = argmin_{-\frac{P}{2}<\tau'<\frac{P}{2}} \left\| (C_{T'}^{-1}\hat{Y})_{[P-M]} - 1_{1x(P-M)}\widetilde{\widehat{m}}_{T'} \right\|. \tag{4.2}$$

(d) The C matrix is circularly shifted one row (the last row in the original matrix will be the first one of the new version) and the steps from (a) are repeated.

These steps are done P times, counting the first operations before the circular shifts.

At the end of this cycle P τ values, will have been obtained.

3. The smallest of the P values is selected, and its respective $\widehat{\widehat{m}}_{\widehat{\tau}}$ is used to calculate the dc offset with (4.3), where $\bar{c}$ and ${\sigma_c}^2$ are, respectively, the arithmetic mean and the variance of **C**.

$$\widehat{m} = \frac{\bar{c}}{{\sigma_c}^2}\widehat{\widehat{m}}_{\widehat{\tau}} \tag{4.3}$$

- Software adaptations: To explain how this software section works, it is necessary to first demonstrate two useful properties of the circulant matrices and of the norm of a complex vector.

  Postulate: Given an inverse circulant matrix C^{-1} of dimensions $|PxP|$ and a matrix Y of dimensions $|Px1|$, the circular shifting of the multiplication $C^{-1}Y$ is equal to the circular shifting of C^{-1} by the vector Y. In other words:

$$circshift(C^{-1}Y) = circshift(C^{-1})Y \tag{4.4}$$

where *circshift* indicates a circular shifting of one row.

This is important by the following reason: the multiplication of $C^{-1}Y$ has to be done only once, and not P times like in the original version. This spare PxP-1 multiplications to the algorithm.

Moreover, the part of the multiplication that is used (it is important not to forget that only the last P-M rows of $C^{-1}Y$ are needed) does not need to be shifted once each cycle, the needed indexes can be calculated each time directly from the full multiplication vector. For example, if one calculation uses elements 3, 4, 5, and 6, the shift indicates that the next time 4, 5, 6, and 7 will be used. This pattern can be inferred, so no shifting has to be applied. All that is needed is to increment each of the indexes and, if one of them has reached the P value, it is restarted to zero. Following the past example, lets suppose that P=7, so indexes go from zero to 6. In the first cycle there is no problem as all 3, 4, 5, and 6 are in the range of permitted values. This is not the case of 4, 5, 6, and 7, where the last index is restarted to 0 (7-P or 7-7).

Now, lets analyze the way in which the norm of a vector, containing complex elements, is performed.

For a vector V (4.5), of *n* elements, where v_{kr} and v_{ki}, with *k* from 0 to *n-1* define the real and imaginary part of each element respectively, its norm is defined as in equation (4.6).

$$V = \left\{ \begin{matrix} v_{0r} + v_{0i} \\ v_{1r} + v_{1i} \\ v_{2r} + v_{2i} \\ . \\ . \\ . \\ v_{n-1r} + v_{n-1i} \end{matrix} \right\} \tag{4.5}$$

$$Norm(V) = \sqrt{(v_{0r})^2 + (v_{0i})^2 + (v_{1r})^2 + (v_{1i})^2 + ... + (v_{n-1r})^2 + (v_{n-1i})^2} \tag{4.6}$$

Nevertheless, the norm that the DDST needs comes from the result of a subtraction (4.2), so in the original version it is necessary to wait until each $\widehat{\widetilde{m}}_{\widehat{\tau}}$ is calculated. The mentioned subtraction $((C_{T'}^{-1}\hat{Y})_{[P-M]} - 1_{1x(P-M)}\widehat{\widetilde{m}}_{T'})$ can be expressed as:

$$\left\{ \begin{matrix} (C_{T'}^{-1}\hat{Y}_{Mr} + C_{T'}^{-1}\hat{Y}_{Mi}) - (\widehat{\widetilde{m}}_{\widehat{\tau}r} + \widehat{\widetilde{m}}_{\widehat{\tau}i}) \\ (C_{T'}^{-1}\hat{Y}_{M+1r} + C_{T'}^{-1}\hat{Y}_{M+1i}) - (\widehat{\widetilde{m}}_{\widehat{\tau}r} + \widehat{\widetilde{m}}_{\widehat{\tau}i}) \\ (C_{T'}^{-1}\hat{Y}_{M+2r} + C_{T'}^{-1}\hat{Y}_{M+2i}) - (\widehat{\widetilde{m}}_{\widehat{\tau}r} + \widehat{\widetilde{m}}_{\widehat{\tau}i}) \\ . \\ . \\ . \\ (C_{T'}^{-1}\hat{Y}_{P-1r} + C_{T'}^{-1}\hat{Y}_{P-1i}) - (\widehat{\widetilde{m}}_{\widehat{\tau}r} + \widehat{\widetilde{m}}_{\widehat{\tau}i}) \end{matrix} \right\} \tag{4.7}$$

where numerical subindexes correspond to the number of element, while *r* and *i* subindexes correspond to the real and imaginary part of the numbers.

To make the following explanation easier to understand, 4.8 is rewritten as:

$$\left\{\begin{array}{c} (CY_{Mr}+CY_{Mi})-(m_{\widehat{\tau}r}+m_{\widehat{\tau}i}) \\ (CY_{M+1r}+CY_{M+1i})-(m_{\widehat{\tau}r}+m_{\widehat{\tau}i}) \\ (CY_{M+2r}+CY_{M+2i})-(m_{\widehat{\tau}r}+m_{\widehat{\tau}i}) \\ . \\ . \\ . \\ (CY_{P-1r}+CY_{P-1i})-(m_{\widehat{\tau}r}+m_{\widehat{\tau}i}) \end{array}\right\} = \left\{\begin{array}{c} (CY_{Mr}-m_{\widehat{\tau}r})+(CY_{Mi}-m_{\widehat{\tau}i}) \\ (CY_{M+1r}-m_{\widehat{\tau}r})+(CY_{M+1i}-m_{\widehat{\tau}i}) \\ (CY_{M+2r}-m_{\widehat{\tau}r})+(CY_{M+2i}-m_{\widehat{\tau}i}) \\ . \\ . \\ . \\ (CY_{P-1r}-m_{\widehat{\tau}r})+(CY_{P-1i}-m_{\widehat{\tau}i}) \end{array}\right\} \tag{4.8}$$

And its norm will be:

$$\sqrt{(CY_{Mr}-m_{\widehat{\tau}r})^2+(CY_{Mi}-m_{\widehat{\tau}i})^2+...+(CY_{P-1r}-m_{\widehat{\tau}r})^2+(CY_{P-1i}-m_{\widehat{\tau}i})^2} \tag{4.9}$$

Expanding the *k-th* binomial inside the square root, with *k* from *M* to *P-1*:

$$(CY_{kr}-m_{\widehat{\tau}r})^2 = {CY_{kr}}^2 - 2\cdot CY_{kr}\cdot m_{\widehat{\tau}r} + {m_{\widehat{\tau}r}}^2 \tag{4.10}$$

Replacing (4.10) in (4.9):

$$\sqrt{A-2\cdot B+C} \tag{4.11}$$

where:

$$A = {CY_{Mr}}^2+{CY_{Mi}}^2+{CY_{M+1r}}^2+{CY_{M+1i}}^2+...+{CY_{P-1r}}^2+{CY_{P-1i}}^2 \tag{4.12}$$

$$B = m_{\widehat{\tau}r} \cdot (CY_{Mr} + CY_{M+1r} + ... + CY_{P-1r}) + m_{\widehat{\tau}i} \cdot (CY_{Mi} + CY_{M+1i} + ... + CY_{P-1i}) \tag{4.13}$$

and

$$C = (P - M) \cdot ({m_{\widehat{\tau}r}}^2 + {m_{\widehat{\tau}i}}^2) \tag{4.14}$$

Now it can be seen that A and B are calculated at the same time that each $\widehat{\widehat{m}}_{\widehat{\tau}}$ are operated, so, at the end of each iteration, all that is needed is to use (4.11) to obtain the corresponding norm.

Apart from this, it can be demonstrated that if $a < b$, $\sqrt{a} < \sqrt{b}$, so it is not necessary to compare the norms of each iteration: the comparison of the square norms will give the same result.

- Hardware modules: The above lines show that many memory accesses are needed to obtain the arithmetic mean vector **Y**. This process is accelerated by using the **Arithmetic Mean Coprocessor**.

4.4.4 FFT of N and 2N Points

The FFT or Fast Fourier Transform is a method to perform the Discrete Fourier Transform by exploiting the redundancy in the DFT and dividing the original set of data, with a size N until having N/2 two elements DFTs. For a formal explanation of the method, see [31].

- Mathematical operations: This implementation uses a decimation in time FFT algorithm known as decimation in time Cooley-Turkey.
- Software adaptations: The version of the FFT code used is a modification from the Altera's *Accelerating Nios II Systems with the C2H Compiler Tutorial*, which can be found in [40].
- Hardware modules: For the two sizes of needed FFTs, just one module (**the FFT coprocessor**) is used.

4.4.5 Rough and Iterative CFO Estimation

As this block makes its processing in the frequency domain, it is necessary to perform a Discrete Fourier Transform in order to move the input data from its own time domain. The 2N points FFT discused in the past section is used for this.

- Mathematical operations:

  1. An N size block from the input data, after the dc-offset removal, is taken.
  2. If necessary, *zero padding* is applied to the data before the FFT.
  3. The FFT is performed.
  4. After the FFT, only the magnitude of its elements is used, so (4.15) is done on each $R_r(k) + R_i(k)$ (the real and imaginary terms of the elements resulting from the FFT), with *k* from 0 to 1023.

$$\sqrt{R_r(k)^2 + R_i(k)^2} \tag{4.15}$$

5. The FFT result vector is reshaped as a matrix of dimensions $|NpxP|$, where $Np = 1024/P$.

6. All rows are added to obtain a $|Npx1|$ matrix.

7. The highest of this matrix is found and its index is stored as *ind*.

8. An fb frequency is calculated as:

$$fb = \begin{cases} \dfrac{ind}{1024} & \text{if } ind < 1024/2P \\ \\ \dfrac{ind}{1024} - \dfrac{1}{P} & \text{if } ind \geq 1024/2P \end{cases} \tag{4.16}$$

9. Three more frequencies are obtained:

$$Afb\_c = fb$$

$$Afb\_i = Afb\_c - \frac{2^{-it}}{1024} \tag{4.17}$$

$$Afb\_d = Afb\_c + \frac{2^{-it}}{1024}$$

with it=1.

10. The vector zk_c is calculated as:

$$zk\_c = r(k) * e^{-j2\pi Afb\_c \cdot k} \tag{4.18}$$

with k from 0 to L-1.

L is the length of **r** (the original input set of data).

This is equal to:

$$zk\_c = \left\{ \begin{array}{c} r(0) \cdot e^{-j2\pi Afb\_c \cdot 0} \\ r(1) \cdot e^{-j2\pi Afb\_c \cdot 1} \\ r(2) \cdot e^{-j2\pi Afb\_c \cdot 2} \\ . \\ . \\ . \\ r(L-1) \cdot e^{-j2\pi Afb\_c \cdot (L-1)} \end{array} \right\} \tag{4.19}$$

11. zk_i and zk_d are also calculated using Afb_i and Afb_d, respectively.
12. 1024 points FFT's are performed on zk_c, zk_i, and zk_d, obtaining the vectors ZK_c, ZK_i, and ZK_d.
13. J_c is calculated as:

$$J\_c = \sum_{i=1}^{P-1} \| ZK\_c(i * 64) \| \tag{4.20}$$

14. J_c, J_i, and J_d are calculated replacing ZK_i and ZK_d in (4.20), respectively.
15. fb takes the new value:

$$fb = \begin{cases} Afb\_c & \text{if } J\_c > J\_i \text{ and } J\_c > J\_d \\ Afb\_i & \text{if } J\_i > J\_c \text{ and } J\_i > J\_d \\ Afb\_d & \text{if } J\_d > J\_c \text{ and } J\_d > J\_i \end{cases} \tag{4.21}$$

16. *it* is incremented in 1.

17. Steps 9 to 16 are repeated eleven times. At the end of the process, Afb_c will have the value of the estimated CFO.

- Software adaptations: It is important to note that while no input data is bigger than 2, the FFT produces higher values (a 54 is not uncommon). When obtaining the magnitude or norm of each datum, the format used can originate overflows, so a software-only solution has two options: before each multiplication, both factors can be scaled to a different format. For example, right-shifting both elements 5 positions results in a new format with 15 integer bits and 17 fractional ones. Obviously, this means that each product will lose its original resolution, so at the end it is possible to have high accuracy errors. The second option is to pass to a 64 bit fixed-point arithmetic, but this produces problems with other parts of the code, and sooner or later, another shifting has to be performed, or all following operations would have to be made with 64 bits variables. This implementation uses the first alternative, when no hardware accelerator is used. Under these same conditions, it is important to stand out the necessity of a square root for the calculation of each magnitude. The software-only implementation uses the NIOS II IDE version of the sqrt function inside the math.h library. Although the values entered to this function are integers representing a fixed-point real number, sqrt works with floating point values, so the advantages of the original format are lost during this operation.

  For the second important software adaptation, it is important to review (4.18). Almost no programming language has a function that calculates directly a complex exponential, so it is necessary to resort to the *Euler formula* (4.22).

$$e^{ix} = cosx + isinx \tag{4.22}$$

if $x = -2\pi Afb\_c \ \cdot k$, then 4.22 can be rewritten as:

$$e^{-i2\pi Afb\_c \ \cdot \ k} = cos(-2\pi Afb\_c \ \cdot k) + isin(-2\pi Afb\_c \ \cdot k) \tag{4.23}$$

or, using the sine and cosine symmetries:

$$e^{-i2\pi Afb\_c \ \cdot \ k} = cos(2\pi Afb\_c \ \cdot k) - isin(2\pi Afb\_c \ \cdot k) \tag{4.24}$$

As with the square root, the *sin* and *cos* functions of math.h work with floating point, taking a considerable amount of time to give a result.

- Hardware modules: It was already mentioned that this block uses the FFT accelerator of the past section. It also uses the **Norm Coprocessor**, to obtain the magnitude of all FFT's elements, and the Complex Exponential module to accelerate the iterative section of the code.

4.4.6 CFO Correction

Once the CFO has been estimated, it is necessary to remove it from the input data.

- Mathematical operations: The corrected **r** is obtained by applying (4.25).

$$r(k) = r(k) \cdot e^{-j2\pi Afb\_c \ \cdot k} \tag{4.25}$$

that is almost identical to (4.18).

- Software adaptations: The Euler formula is used again to solve (4.25).
- Hardware modules: This implementation also uses the Complex Exponential hardware accelerator. The possibility of changing it for a*CORDIC or a BKM generator* for the sine and cosine calculations is under study for systems in which space is the most important constraint. Because of the resolution required by the DDST, look-up tables cannot be used as they would require an enormous amount of memory. For example, if 32 bits logic is used, $32 \cdot 2^{32}$ bits would be needed for look-up tables storage. This is 16 GBytes of memory. An unrealistic requirement even for desktop computers.

4.4.7 Training Sequence Synchronization Estimation

As it will be explained bellow, the training sequence synchronization estimation performs almost the same operations of the dc-offset estimation, which is an advantage as already written code and built accelerators can be used again with just a few changes.

- Mathematical operations: The TSS estimation follows all the steps of the dc-offset, with the following two exceptions:
  - The buffer used takes 2N data from the **r** vector, not only N.
  - Step 3 is not performed as the only needed datum to obtain is the τ.

  The reason why these two blocks cannot be performed at the same time is that the CFO and the dc-offset are requisites for a correct TSS estimation.

- Software adaptations: The same adaptations of the dc-offset block are used here.

- Hardware modules: The arithmetic mean accelerator is also used for the TSS estimation block.

4.4.8 Block Synchronization Estimation

- Mathematical operations: The Block Synchrony Estimation block is calculated as follows:

  1. First, the vector **V** is calculated as:

$$V = I - \frac{1}{P - M} * A \tag{4.26}$$

     where I is an *identity matrix* of dimensions $|P-M\ x\ P-M|$ and A is a *matrix of ones* of the same dimensions. An expanded version of (4.26) is shown in (4.27).

$$V = \begin{Bmatrix} 1 & 0 & 0 & ... & 0 \\ 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 1 & ... & 0 \\ . \\ . \\ . \\ 0 & 0 & 0 & ... & 1 \end{Bmatrix} - \begin{Bmatrix} \frac{1}{P-M} & \frac{1}{P-M} & \frac{1}{P-M} & \cdots & \frac{1}{P-M} \\ \frac{1}{P-M} & \frac{1}{P-M} & \frac{1}{P-M} & \cdots & \frac{1}{P-M} \\ \frac{1}{P-M} & \frac{1}{P-M} & \frac{1}{P-M} & \cdots & \frac{1}{P-M} \\ . \\ . \\ . \\ \frac{1}{P-M} & \frac{1}{P-M} & \frac{1}{P-M} & \cdots & \frac{1}{P-M} \end{Bmatrix} \tag{4.27}$$

2. The variable Tp is initialized with 0.

3. The vector yk_Tp is calculated as:

$$Yk\_Tp(j) = \frac{P}{N+P} \sum_{i=0}^{Np-1} \{r((i*P) + (Tp*P) + \tau + j)\} \tag{4.28}$$

   where **r** is the original data set minus the dc-offset and the CFO removal, N is the length of **r**, $Np = \frac{N}{P}$, and τ is the value returned by the TSS estimation block.

4. The vector CYp is obtained:

$$CYp = C^{-1} * Yk\_Tp \tag{4.29}$$

5. The vector Aux_{CY} is defined as the one composed by the last P-M rows of (4.29).

6. The norm NormY is calculated as:

$$NormY = \|V * Aux_{CY}\| . \tag{4.30}$$

   and its value is stored.

7. Tp is incremented, and the steps 3 to 6 are repeated Np-1 times.

8. At the end of the process, all the norms are compared, and the index corresponding to the greatest of them is stored in Ts.

- Software adaptations: As vector Yk_Tp changes on each Tp iteration, the calculation of the full multiplication in (4.29) is not usefull, as it was in the dc-offset estimation case. Nevertheless, it is possible to reduce the amount of operations by directly selecting only the C^{-1} elements that would be used for Aux_{CY}, using an index calculation outline as the one discused in the dc-offset removal software adaptations section. It is important to mention that it is not necessary to use a double nested loop to browse the C^{-1} elements, as the first of its rows contains all the elements that will be multiplied. This is possible because of the circulant matrix properties. For example, a multiplication of Aux_{CY} has the form:

$$C^{-1}(i) * YK\_Tp(j) \tag{4.31}$$

  With *i* from M to P-1 and *j* from 0 to P-1. Both indexes can be incremented each iteration of *j*, but as *i* begins with an M value, indexes out of the matrix length will be obtained before the end of the process. All that is needed to avoid this problem is to check if *i* is greater than P-1, in which case this index is restarted to 0.

For the next software adaptation, lets analyze (4.30). If the variable MP is used to describe $\dfrac{1}{P-M}$, (4.27) can be rewritten as:

$$V = \left\{\begin{matrix} 1 & 0 & 0 & ... & 0 \\ 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 1 & ... & 0 \\ . \\ . \\ . \\ 0 & 0 & 0 & ... & 1 \end{matrix}\right\} - \left\{\begin{matrix} MP & MP & MP & ... & MP \\ MP & MP & MP & ... & MP \\ MP & MP & MP & ... & MP \\ . \\ . \\ . \\ MP & MP & MP & ... & MP \end{matrix}\right\} \tag{4.32}$$

This is equal to:

$$V = \left\{\begin{matrix} 1-MP & -MP & -MP & ... & -MP \\ -MP & 1-MP & -MP & ... & -MP \\ -MP & -MP & 1-MP & ... & -MP \\ . \\ . \\ . \\ -MP & -MP & -MP & ... & 1-MP \end{matrix}\right\} \tag{4.33}$$

Rewriting 1-Mp as a, -Mp as b, and the elements of Aux_{CY} as C(k), with k from M to P-1, the multiplication inside (4.30) in it expanded version looks like:

$$\left\{\begin{matrix} a & b & b & ... & b \\ \\ b & a & b & ... & b \\ \\ b & b & a & ... & b \\ . & & & & \\ . & & & & \\ . & & & & \\ \\ b & b & b & ... & a \end{matrix}\right\} * \left\{\begin{matrix} C(M) \\ \\ C(M+1) \\ \\ C(M+2) \\ . \\ . \\ . \\ \\ C(P-1) \end{matrix}\right\} \tag{4.34}$$

The first element of the resulting vector is:

$$aC(M)+bC(M+1)+bC(M+2)+...+bC(P-1) = b(C(M+1)+C(M+2)+...+C(P-1))+aC(M) \tag{4.35}$$

Unfortunately, the C(k) multiplied by a is different for each element of the multiplication vector, but rewritting (4.35) as:

$$b(C(M)+C(M+1)+C(M+2)+...+C(P-1))+aC(M)\boldsymbol{- bC(M)} \tag{4.36}$$

all of the products can be generalized as:

$$b\sum_{i=M}^{P-1}(C(i)) + aC(K) - bC(k) = b\sum_{i=M}^{P-1}(C(i)) + C(K)(a-b) \tag{4.37}$$

with k from M tp P-1. But a=1-MP and b=-MP, so:

$$b\sum_{i=M}^{P-1}(C(i))+C(K)(a-b) = b\sum_{i=M}^{P-1}(C(i))+C(K)(MP-1-(MP)) = b\sum_{i=M}^{P-1}\{C(i)\}+C(K)) \tag{4.38}$$

So, calculating $\sum_{i=M}^{P-1}(C(i))$ along with each element of Aux_{CY}, it is possible to obtain $V * Aux_{CY}$ very fast. Moreover, as square values can be compared, sparing the use of the square root, at the end of this process the index of the greatest norm will also be obtained, without an extra loop to calculate it.

- Hardware modules: The arithmetic mean accelerator could be used in this block, but only after an extra ordering step, so at the end it was decided to leave this section as a software-only one.

4.4.9 Channel Estimation

- Mathematical operations: Channel estimation requires the following steps:

  1. This stage works over the received block, for which its beginning has already been obtained. The block is referred by the vector x, that is calculated as indicated by [[4.39].

$$x(k) = r(k + Ts) \tag{4.39}$$

with k form 0 to $N + Ts - 1$

2. x is redimensioned as a matrix of $|P\ x\ Np|$ and the average of each row is calculated to obtain the vector y of P elements.

3. The channel estimation h is calculated as the first M elements of the vector resulting from the multiplication:

$$C^{-1} \cdot y \tag{4.40}$$

- Software adaptations: As only the first M elements of the multiplication are required, the last ones can be omitted from the operation. This saves $(P - M)^2$ multiplications and $P - M$ additions.

- Hardware modules: Because of its characteristics, the arithmetic mean accelerator is used in this block.

4.4.10 OCI Training Sequence Removal

An *optimum channel independent* training sequence or OCI training sequence satisfies $C^H C = CC^H = P\sigma_c^2 I_{PxP}$, thus simplifying the projection operations between subspaces.

- Mathematical operations: To remove the OCI training sequence from the received block (now saved in vector x), it is sufficient with equaling to zero all the elements of X where the training sequence was added to the information, that is:

$$x(k * Np) = 0 \tag{4.41}$$

with k form 0 to $P - 1$, Np=N/P, and X is the N points FFT of x.

- Software adaptations: As it was mentioned before, the FFT accelerator used only obtains transformations from 2N points data sets, while this block only requires a N points FFT. The result from this last FFT is in fact stores in the even results from its 2N version, that is:

$$X_{FFT_N}(k) = X_{FFT_{2N}}(2 * k) \tag{4.42}$$

To take advantage of this fact, [4.41] is rewritten as:

$$x(2 * k * Np) = 0 \tag{4.43}$$

- Hardware modules: No accelerator is used for this block.

4.4.11 Rough and Iterative Symbol Detection

The equalization of the data can now be performed. It is important to mention that the DDST receiver prototype works with data that have been coded with 4QAM or QPSK modulation, so the effort of this block is to recover the symbols corresponding to this modulation from the input data.

- Mathematical operations: For the rough detection, two vectors have to be obtained G and Z. This last one is equal to the discrete Fourier transform of x, which was already obtained and named X, so:

$$Z = X \tag{4.44}$$

G is obtained by following these steps:

1. H is obtained as the FFT of the estimated channel (h) with zero padding until having N points.
2. An auxiliar vector A is obtained as in [4.45]

$$A = conj(H(k))/|H(k)| \tag{4.45}$$

where $conj(H(k))$ indicates the conjugate of each H and $|H(k)|$ its magnitude.

3. G is the diagonal matrix version of A:

$$\left\{\begin{matrix} A(0) & 0 & 0 & ... & 0 \\ \\ 0 & A(1) & 0 & ... & 0 \\ \\ 0 & 0 & A(2) & ... & 0 \\ & . & & & \\ & & . & & \\ & & & . & \\ \\ 0 & 0 & 0 & ... & A(N-1) \end{matrix}\right\} \tag{4.46}$$

Once G and Z have been obtained, the rough symbol detection u_e is obtained through [4.47].

$$u\_e = IFFT(G \cdot Z) \tag{4.47}$$

that is, the inverse FFT of the product of G by Z.

The iterative symbol detection is obtained as follows:

1. The rough estimate u_e is copied to another vector aux.
2. For each complex datum of aux ($aux_r + aux_i$), its phase is obtained as:

$$phase(k) = arctan(\frac{aux_i(k)}{aux_r(k)}) \tag{4.48}$$

3. For each $phase(k)$:
   - if $0° \leq phase(k) \leq 90°,\ aux(k) = c + c * i$

- if $90^\circ < phase(k) \leq 180^\circ,\ aux(k) = -c + c * i$
- if $180 < phase(k) < 270^\circ,\ aux(k) = -c - c * i$
- if $270 \leq phase(k) < 360^\circ,\ aux(k) = c - c * i$

where $c = \sqrt{\sigma_{wk\_2}/2}$ and σ_{wk_2} is a previously known constant.

4. A vector J is obtained as:

$$J = \frac{1}{Np} * O_{NpxNp} \otimes D_{PxP} \tag{4.49}$$

where O_{NpxNp} and D_{PxP} are, respectively, the matrices of dimensions $|Np\ x\ Np|$ and $|P\ x\ P|$:

$$O_{NpxNp} = \left\{ \begin{matrix} 1 & 1 & 1 & 1 & ... & 1 \\ 1 & 1 & 1 & 1 & ... & 1 \\ 1 & 1 & 1 & 1 & ... & 1 \\ & & . & & & \\ & & & . & & \\ & & & & . & \\ 1 & 1 & 1 & 1 & ... & 1 \end{matrix} \right\} \tag{4.50}$$

and

$$D_{PxP} = \left\{ \begin{matrix} 1 & 0 & 0 & ... & 0 \\ \\ 0 & 1 & 0 & ... & 0 \\ & . & & & \\ & & . & & \\ & & & . & \\ 0 & 0 & 0 & ... & 1 \end{matrix} \right\} \tag{4.51}$$

The operator $\otimes$ indicates a Kronecker product. In this operation each member of the left side matrix (of dimensions $|MxN|$) is multiplied by the whole right side matrix (of dimensions $|OxP|$), resulting in a matrix of dimensions $|(MxO)x(NxP)|$. For example:

$$\left\{ \begin{matrix} 3 & 1 & 4 \\ 1 & 4 & 0 \\ 0 & 2 & 2 \end{matrix} \right\} \otimes \left\{ \begin{matrix} 2 & 0 \\ 0 & 2 \end{matrix} \right\} = \left\{ \begin{matrix} \boxed{\begin{matrix} 6 & 0 \\ 0 & 6 \end{matrix}} & \boxed{\begin{matrix} 2 & 0 \\ 0 & 2 \end{matrix}} & \boxed{\begin{matrix} 8 & 0 \\ 0 & 8 \end{matrix}} \\ \\ \boxed{\begin{matrix} 2 & 0 \\ 0 & 2 \end{matrix}} & \boxed{\begin{matrix} 8 & 0 \\ 0 & 8 \end{matrix}} & \boxed{\begin{matrix} 0 & 0 \\ 0 & 0 \end{matrix}} \\ \\ \boxed{\begin{matrix} 0 & 0 \\ 0 & 0 \end{matrix}} & \boxed{\begin{matrix} 4 & 0 \\ 0 & 4 \end{matrix}} & \boxed{\begin{matrix} 4 & 0 \\ 0 & 4 \end{matrix}} \end{matrix} \right\} \tag{4.52}$$

5. A vector w_e is calculated as in equation [4.53].

$$w\_e = u\_e + J * aux \tag{4.53}$$

6. The vector aux is updated with the values of w_e and the process is repeated from step 2.

The number of iterations is variable, but according to simulations, the first one offers a satisfying performance. The receiver performs two iterations in total.

- Software adaptations: The calculus of phase of each complex pair is a computer intensive operation. Nevertheless, as it can be seen from step 2 in the iterative symbol detection, this data characteristic is obtained to determine the quadrant of the complex plane to which each datum belongs. This information can be obtained from the sign of the real and imaginary parts of the complex numbers, so step 1 can be omitted, rewritting the mentioned step 2 and its conditions as:

  For each $aux(k)$:

  - if $aux_r(k) \geq 0$ and $aux_i(k) \geq 0,\ aux(k) = c + c * i$
  - if $aux_r(k) < 0$ and $aux_i(k) \geq 0,\ aux(k) = -c + c * i$
  - if $aux_r(k) < 0$ and $aux_i(k) < 0,\ aux(k) = -c - c * i$
  - if $aux_r(k) \geq 0$ and $aux_i(k) < 0,\ aux(k) = c - c * i$

  These changes significantly reduces the complexity of the symbol detection block.

  The iterative part of this stage requires a very large amount of multiplications, first to obtain J from the Kronecker product and then to compute $J * w\_e$. An analysis from J shows that it contains, in its majority, elements equal to zero. In fact, $N^2(1-\frac{1}{P})$ elements of J are equal to zero. For typical values of $P = 16$ and $N = 512$, this means that the 93.75% of the elements of J are zero. Under this scheme, N^2

multiplications are required to obtain J, and another N^2 multiplications and N^2 additions to compute $J * w\_e$.

The receiver uses a modified version of the algorithm in which J is never calculated. After all, the non zero elements are equal to $\frac{1}{Np}$ and their positions are known. For the first row, they are in positions 0, P, 2P, ..., N-P. In row two, they are 1, P+1, 2P+1, ..., N-P+1. In this way, everything that is necessary to calculate $J * w\_e$ is w_e and to know the positions of the non zero elements of J. Moreover, in each row of J there are only Np elements different from zero, and as they are all equal, at the end only one multiplication per row is needed. At the end, from the total of $2 * N^2$ multiplications and N^2 additions, the receiver only needs to obtain N multiplications and $Np * N$ additions.

- Hardware modules: No accelerator is used for this block.

4.5 In perspective

It is possible that the length of this chapter makes difficult to understand all its content, but the task is easier with the knowledge that all the information describes the two techniques used ti speed up the implementation of the DDST algorithm: the hardware acceleration modules, and the analysis and adaptation of the algorithm itself to be more “friendly” with the characteristics and available resources of the FPGA. Now that a general view of the architecture has been given, it is the moment to describe more deeply the hardware accelerators of the system.

Chapter 5

Coprocessors Insight

This chapter discusses with more depth the hardware accelerators, or coprocessors, that work with the main system to perform the most demanding operations needed by the DDST digital receiver. Moreover, these coprocessors also work with larger data resolutions, so the loss of accuracy due to truncations are almost null. All of the modules have a certain degree of flexibility with respect to the possibility of changing some of their parameters, to the point that it is possible to pass them several variables values through the software part of the system, that is, parameters such as the number of times each module will work or the addresses of the memories to read from and write to can be assigned from a C language set of functions.

5.1 Arithmetic Mean

In the low complexity blocks of the DDST digital receiver (figure 4.4), the most critical operations to be performed are the matrices multiplications (basically $C^{-1}Y$). Nevertheless, the hardware acceleration of such multiplications is well studied, and the basic strategy is always the same: to execute as many operations in parallel as possible. This approach gives good performance, but it consumes many resources, in logic elements, embedded multipliers, and routing. As many other operations in the receiver require the embedded multipliers, and because of the space restrictions, it was decided not to accelerate this operation, and to concentrate in the other steps necessary for this kind of block (dc-offset, TSS, block synchronization, channel estimation). An operation that is repeated several times in these blocks is the redimension and average of the sample vector (which lead to the obtaining of Y). This requires many memory accesses, several additions and P multiplications.

The arithmetic mean hardware module reads directly from the dedicated onchip memory (discused on the *N/2N samples buffer* section from the last chapter) **P** data, each one of 32 bits, accumulating their respective values to **P** 32 bits registers. At the end of the process, the values stored in the registers are multiplied by 1/(Np), so now they contain the arithmetic means of the rows from the reshaped matrix. Finally, the results are stored in another onchip memory. Figure 5.1 shows the architecture of this module (control is not explicitly included).

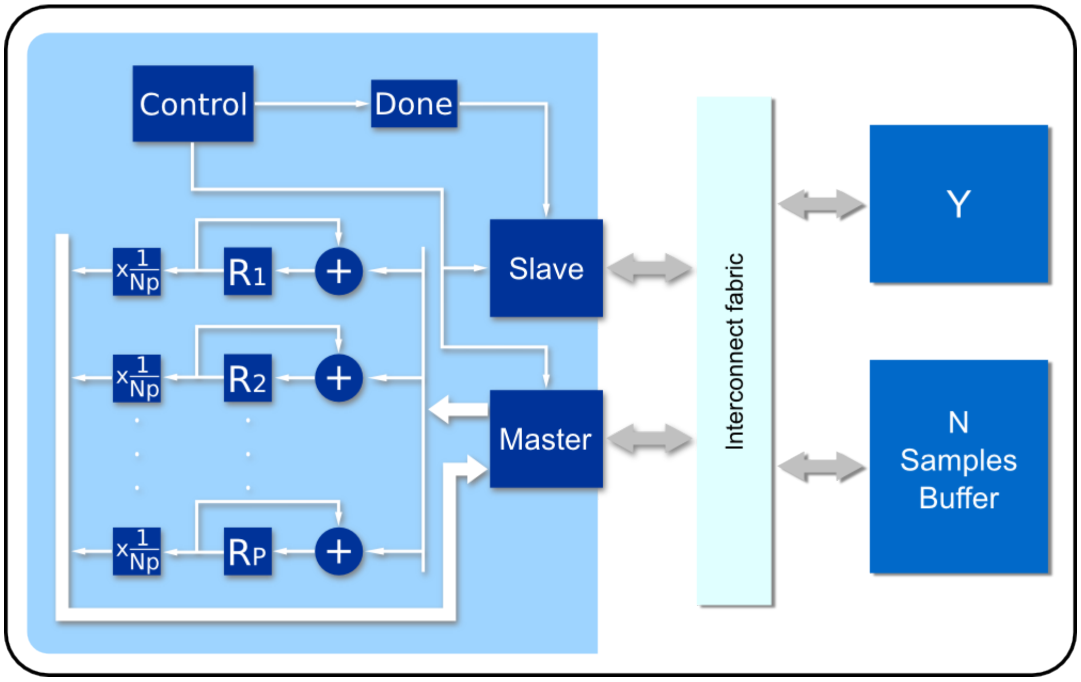


Figure 5.1: Arithmetic mean hardware accelerator.

5.2 Norm / Magnitude

There are several steps of the algorithm in which it is necessary to work only with the magnitude of the complex elements of a vector. The high complexity of these operations comes not from the two multiplications to obtain the squares of the real and imaginary parts of a complex number, but from the necessity to perform a square root. As it was mentioned in blocks like the DC-offset estimation, it is sometimes possible to work with the square norm of the complex samples, but this is not possible in steps like the CFO estimation.

Figure 5.2 presents a block diagram of the magnitude accelerator that, as its name says, calculates each of the norms of the complex samples in a vector V of n elements (as shown in 5.1).

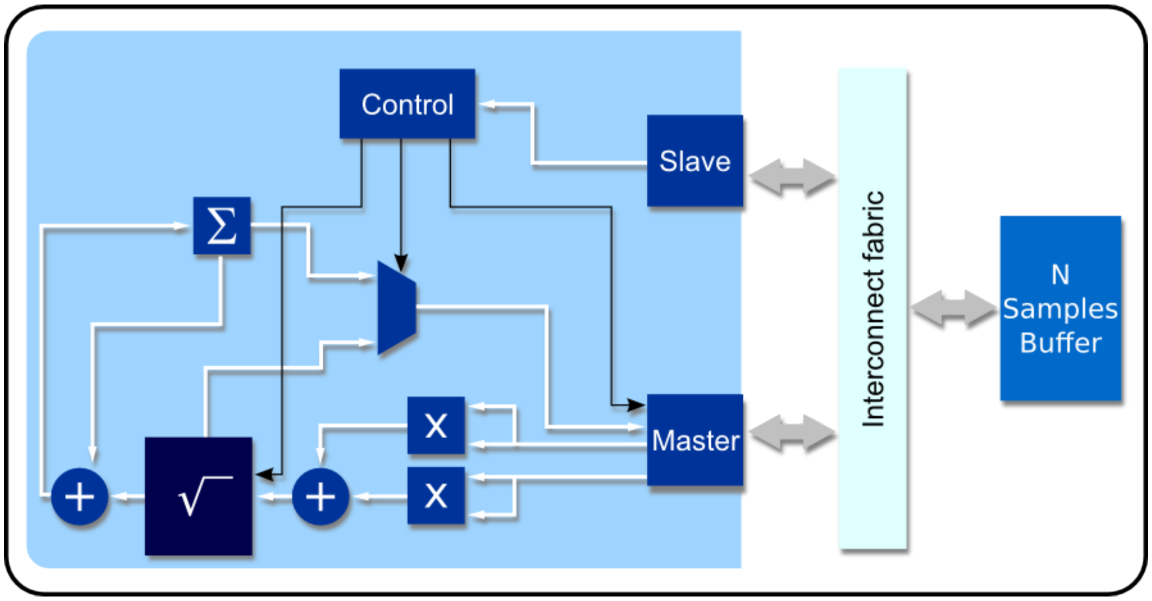


Figure 5.2: Magnitude hardware accelerator module.

$$\sqrt{{v_r}^2(k) + {v_i}^2(k)} \tag{5.1}$$

with $V = \{v_r(k) + v_i(k) * i \in \mathbb{C} \mid \forall k$ from 0 to n-1$\}$

The module has several advantages over the software-only version:

1. It fetches both the real and the imaginary part of the FFT elements each time it performs a read operation.

2. It works with 64 bit arithmetic, so there is no change in the accuracy of the result.

3. As the square root is calculated by a hardware module (explained bellow), it runs faster than the software version, as it does not need to change from fixed to floating point arithmetic.

4. It accumulates all the magnitudes as it works, so at the start of the process it can be decided if it will return either all the norms of the vector, or only their summation, depending on the requirements of the stage in which the module is used. For example, the rough CFO estimation requires all the magnitudes from the sample vector to perform its calculations, while its iterative counterpart only needs the total summation.

5. It is possible to assign an *"offset"* so, for example, the module only obtains the norm of the complex samples in positions 0, 4, 8, ..., etcetera, and not from every sample in the input vector.

6. It has little latency as it obtains and stores data from and to dedicated on-chip memories.

5.2.1 Square Root Submodule

The square root is solved by a submodule which operates in a similar way to a non-restoring algorithm implementation, like the one of [43], but with two main differences:

- First, the 8 more significant bits of the root are obtained from a look-up table. This could be considered an approximated root, that then can be *fine tunned*. For example, the square root of 5 is ≈ 2.236. The look-up table would give a value of 2, so only the decimal part of the result has to be calculated. This increases drastically the speed of the coprocessor while still using very little FPGA area. For example, for a datum of 32 bits, a common non-restoring implementation requires 32 iterations to obtain a result, while this module needs only 5 iterations plus the little time needed to fetch the approximated root.

- Second, each iteration calculates, in parallel, 4 bits of the root, and not only one. This system is depicted in figure 5.3.

The approximated root is obtained from the look-up table using as index the most significant bits of the radicand. Then this value is appended to a set of possible roots that are squared and compared to the original radicand. A comparator tree evaluates all the results and decides which of the possible roots gave the smallest error. This value is then updated as the new approximated root and the next four bits are calculated. With each iteration, the approximated root of the possible set of roots grows four bits, until it reaches the least significant bit.

It is important to consider the bit lenght of the look-up table memory. As more bits are added to this structure, the number of iterations necessary to have the best square root calculation will decrease. Nonetheless, as this lenght grows, the necessary size of the memory to store it also increases significantly, to the point that it is impossible to implement it using the FPGA memory blocks and even the external memories available in the board, like the SDRAM. The size of the look-up table involves a trade-off that is particularly important in the case of architectures based on FPGAs.

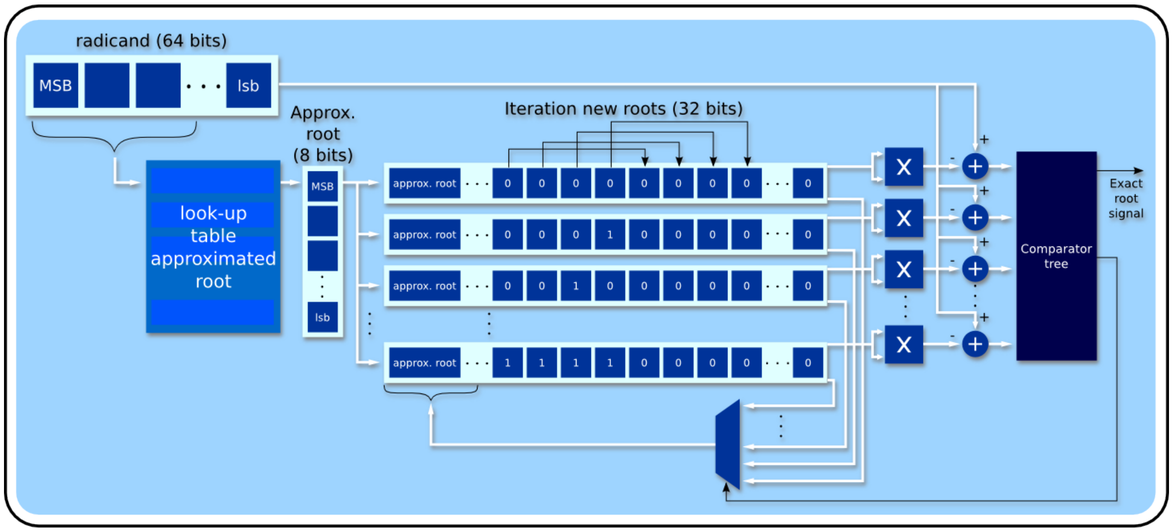


Figure 5.3: Square root submodule

Another advantage of the coprocessor is that it stops its operation as soon as it finds an exact root of an introduced number, so not all entries take the same amount of cycles to be calculated. For example, if the root of 14.0625 is computed, the process will stop as soon as it realizes that 3.75 is its exact square root (on the first iteration), even if the original number is represented as a 64 bits array, that usually requires 6 iterations for full resolution or 5 iterations for a maximum error of $\approx 7.15x10^{-7}$.

5.3 FFT

The FFT module is different from the other hardware accelerators in that it is generated by the c2h tool, and not programmed directly with VHDL. This is due to three main reasons: first, modifications over the code are very easy to do, and even a programmer that does not know anything about the system can perform them. Second, the complexity of the algorithm requires a very hard to build control, overall taking into account that the optimizations require several accesses to four different memories. Finally, changes in the original DDST algorithm can be transformed into C code easily, reducing drastically the time required to update this block of the system.

This implementation uses a technique of ping-pong buffering, in which two memories are used to store the data. At the beginning, the source data are stored in the first memory, and the processed result in the other one. For the next iteration, the source data will be read from the first memory, that is, the results of the past stage are now the input of the next one, and the output will be stored in the first memory. The process is repeated until the FFT is completed.

The implementation of this kind of FFT algorithm consumes significant FPGA memory resources, as the ping-pong buffering requires two memories for the real part of the data and another two for the imaginary one. Moreover, two extra memories are necessary to store the twiddle factors, basically two sets of sines and cosines respectively that are used to combine successive FFT results. As a result, to implement two FFT modules, as the figure 4.4 suggests, would consume resources that could be usefull for other operations. Nonetheless, the structure of the FFT of 2N points contains, implicitly, the result of an N points one. The even elements of the 2N FFT contain all the results of the N points version. In this way, it can be said that the odd elements of the big FFT only provide information that is useless for the following calculations. Obviously, this approach takes twice the time that would be used by a simple N point FFT module, but as it is, even in the 2N case, a fast implementation, this result is preferred to the extra FPGA area that would consume the use of two FFT modules.

5.4 Complex Exponential

As it was mentioned before (chapter 4, section 4.4.5), to shift the complex components of a vector to a certain frequency, each of them has to be multiplied by a complex exponential that involves that frequency (5.2).

$$v_{shift}(k) = v(k) \cdot e^{-j2\pi \cdot f \cdot k} \tag{5.2}$$

where $v(k)$ is the original element k of the vector, and f is the frequency to which the vector will be shifted.

To solve this operation in both software and hardware, it is possible to use numerical analysis methods, but it is more common to rewrite the complex exponential according to the Euler formula. Thus the problem is now reduced to solving both a sine and a cosine. The software implementation of such funtions is usually implemented using a series that converges to the expected result, like in the case of the Taylor series:

$$sinx = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} \quad \text{and} \quad cosx = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} \tag{5.3}$$

Nonetheless, this kind of solutions involve a high degree of computational complexity because of the exponentials and the factorials. In addition, they converge very slowly, so the code in which they are used presents a bottleneck due to such operations. For example, the *sin* and *cos* function from the *math.h* library in the NIOS II processor take more than three hundred cycles each to give a result. This is the main reason why it was decided to create an accelerator for these functions.

The most popular hardware solution for the processing of the trigonometric functions (and in fact for many trascendental functions) is the Coordinate Rotation Digital Computer generator or CORDIC generator. The generalized CORDIC generator include circular, linear and hyperbolic transforms. By the use of an iterative process in which succesive value estimations are made, CORDIC can obtain approximations for operations like trigonometric and hyperbolic functions, and even square roots. The implementation of this generator is very cheap with respect to hardware resources, as it only needs adders, comparators and shifters [37]. The disadvantage of CORDIC is that its iterative nature makes it a slow solution (each iteration increments the precision of the result by one bit), unless a full pipelined implementation is used, which increments the required space.

As the FPGA used has a relatively high amount of logic blocks, and it counts with embedded multipliers, the chosen solution is the Chebyshev Approximation, based on the Chebyshev polynomial:

$$T_k(x) = cos(k \ \times \ arccos(x)) \tag{5.4}$$

that can be writen as a set of true polynomials using:

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x) \quad \forall k \geq 2 \tag{5.5}$$

The Chebyshev Approximation has three advantages over other series solutions like Taylor:

1. It is a very close approximation to the problem of finding the function approximation with a minimum of the maximum error.
2. A pruned polynomial with $M \ll N$ still gives a minimum / maximum estimation.
3. 5.5 can be computed with much fewer coefficients than would be required for a Taylor approximation of the same precision.

The last point is very important for a hardware implementation. For example, the equations:

$$f(x) = sin(\frac{x\pi}{2}) = (51457x + 167x^2 - 21845x^3 + 1183x^4 + 1806x^5)/32768 \tag{5.6}$$

and

$$f(x) = cos(\frac{x\pi}{2}) = (32768 - 18x - 40208x^2 - 949x^3 + 10214x^4 - 1806x^5)/32768 \tag{5.7}$$

are enough for a 16 bit quantization [37].

Figure 5.4 shows the block diagram of this coprocessor. Cs_K and Cc_K represent constants that are equal to the Chebyshev coefficients that multiply each element of the polynomial. At the first cycle, they are multiplied by the input value that is stored in β. The next multiplications depend on the degree of the polynomial element. For example, the third element (Rs_1) will be updated by the results of three multiplications, to obtain the value $Cs_3 \cdot x^3$. Once all the elements form both sine and cosine polynomials have been added, the estimated values are multiplied by a complex number from the vector to operate (the term $v(k) = v_r(k) + v_i(k)$ from (5.2)). This value is read from the SDRAM and the complex product is written to the same memory. Once this process is finished, the value of β is updated according to k (from 0 to N) and the same explained operations are performed again. In total, the whole process is performed N times.

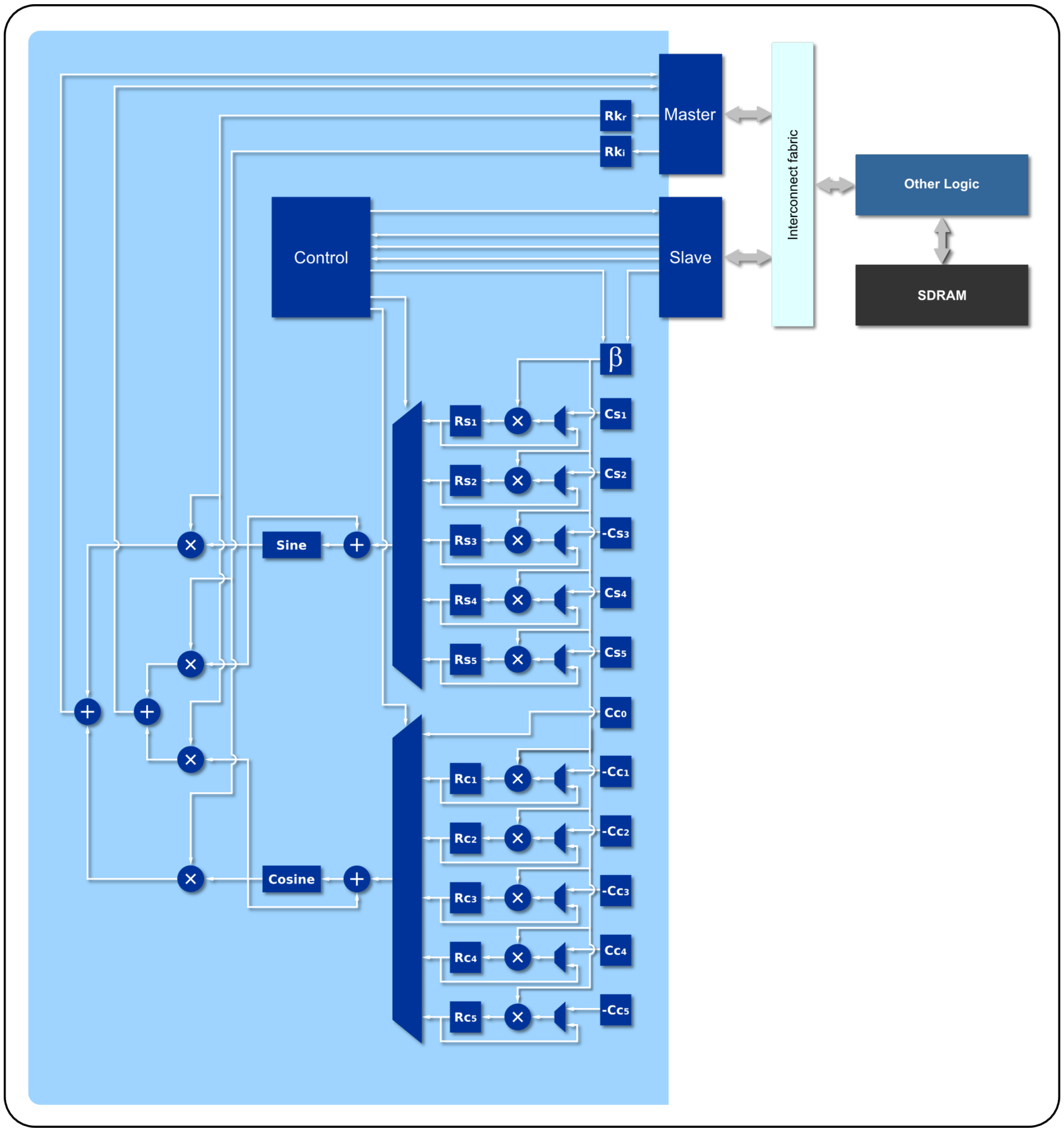


Figure 5.4: Block diagram of the complex exponential coprocessor.

The decision of using the SDRAM instead of the on-chip memories for this module is a result from the high amount of data necessary to operate in the CFO correction stage.

5.5 In perspective

All the described coprocessors are susceptible to modification, and, as the design of the system is fully modular, it is possible to remove any of them, or even to add another one, as long as there is space in the FPGA. These flexibility and modularity are very important characteristics as the DDST algorithm is in continuous improvement.

The next chapter will concentrates on the tests performs with the prototype, describing the characteristics of the platform in which the system was built and giving a comparison of the results obtained with the hybrid architecture against a full software implementation of the algorithm at the digital receiver.

Chapter 6

Tests and Results

The tables in this chapter show experimental results from the built *proof of concept* DDST digital receiver. The data relative to physical characteristics of the system are taken directly from the synthesis report generated by the Altera Quartus II Development software, while the times (in milliseconds) and cycles were measured using a hardware module named **performance counter** offered by Altera. This device measures execution times of certain blocks of code by using directly the available system clocks. Through a set of C language functions, it is possible to print in a console the results of such measurements in an easy and transparent way. The last column of the performance results table shows the times that the hardware accelerated version is faster than its software-only counterpart, both running on the same system (NIOS II based SOPC in a Stratix II DSP board).

6.1 The Board

The system was synthesized for a Stratix II EP2S60 DSP *development board*, from the Altera manufacturer. As its name suggests, the board was built with the specific purpose of being a testing platform for systems and applications that make intensive use of digital signal processing techniques. Its main characteristics are listed here:

- Stratix II EP2S60 FPGA device.
- Two 12-bit 125-MHz A/D converters.
- Two 14-bit 165-MHz D/A converters.
- One 8-bit, 180 megapixels-per-second triple D/A converter for.
- VGA output.
- One 96-KHz Stereo Audio coder/decoder (CODEC).
- 1 MByte of 10-ns asynchronous SRAM configured as a 32-bit bus.
- 16 MBytes of flash memory configured as an 8-bit bus.
- 32 MBytes of SDRAM memory configured as a 64-bit bus.
- Dual seven-segment display.
- Four user-defined push-button switches.
- One female 9-pin RS-232 connector.

- 10/100 Ethernet MAC/PHY.
- Eight user-defined LEDs.
- Socketed 100-MHz oscillator.

The digital to analog and analog to digital converters allow a fast implementation of applications that because of their characteristics require to deal with continuous data, like in the case of RF wireless communications, where the antennas convert electromagnetic waves into electric currents and viseversa. Figure 6.1 shows the block diagram of the board.

6.1.1 Stratix II EP2S60

In the specific case of the FPGA included in the board, the following list of characteristics can give an idea of the capacities of this device, that are suitable for the built and testing of a high range of DSP prototypes, thanks to the quantity of available resources.

- 24176 ALMs.
- 329 M512 RAM blocks.
- 255 M4K RAM blocks.
- 2 M-RAM blocks.
- DSP blocks 36

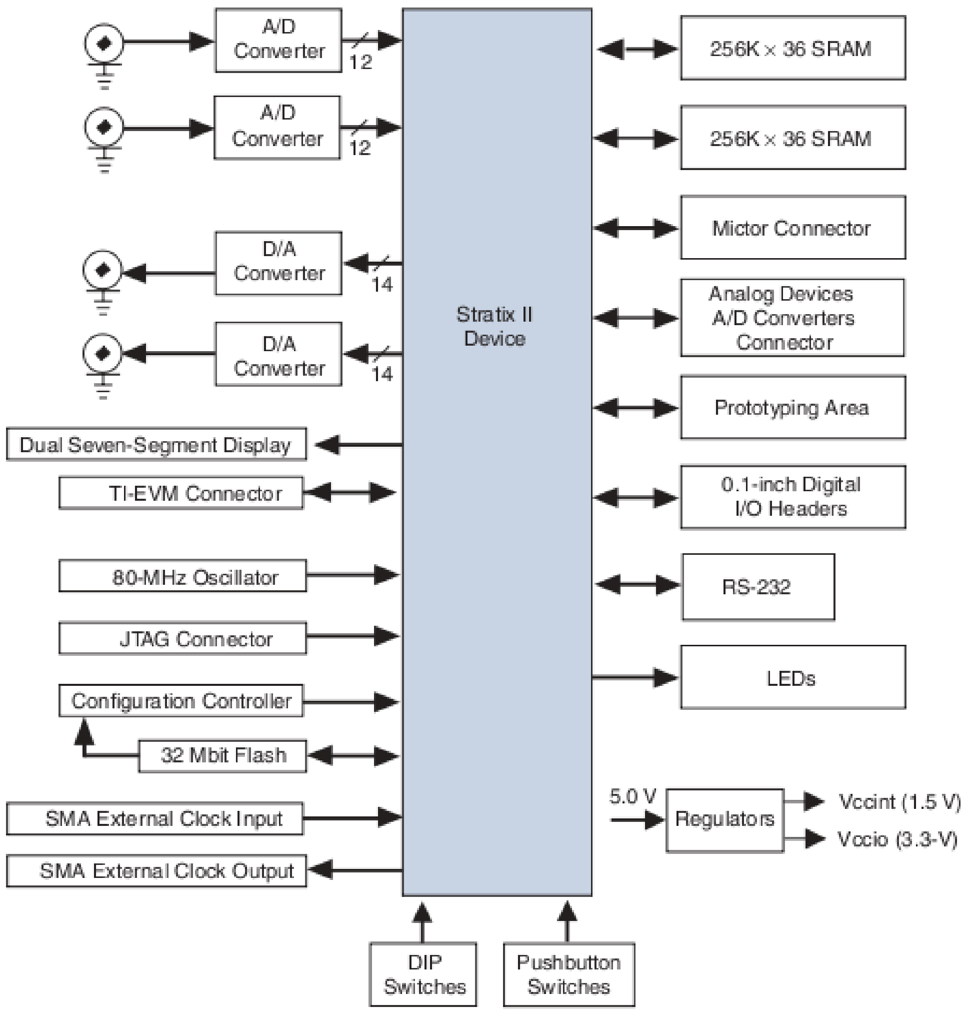


Figure 6.1: Block Diagram of the Stratix II DSP board (taken from [42]).

- 144 18-bit $\times$ 18-bit multipliers.
- 718 user I/O pins.

The ALMs (Adaptive Logic Modules) are the basic building blocks of the Stratix II architecture, that are equivalent to 2.5 of the Xilinx LEs of their FPGAs. The three kinds of memory blocks are known as the *TriMatrix* memory structure, a set of three memory blocks families,each one with specific characteristics. M512 are the smallest blocks, but they have the fastest access time, while the M4K blocks have a balance between speed and capacity, along with the possibility of "preloading" them with specific data. M-RAM blocks present the slowest access time among the FPGA memory structures, but they feature up to 720 Kbytes.

6.2 Tests Description

The following section presents two kind of results: performance results, and data referring to the synthesis of the system in the mentioned board.

The performance test was performed under the following conditions:

- The results are obtained from running ten sets of 3877 input complex data on two systems, which characteristics can be seen in table 6.1.
- The reported times and clock cycles are averages of the ten executions of the process.

- To validate the accuracy and precision of the results for both systems, the output data were compared against a *Matlab* simulation that uses double precision floating point arithmetic.
- Although different theoretical maximum operation frequencies are higher, all the tests were executed at 100 MHz in both systems, as this is the frequency of operation of the oscillator included in the DSP board.

System	Software-only	Hybrid
Platform	Stratix II DSP Board	Stratix II DSP Board
Frequency of operation	100MHz	100MHz
Floating point instructions	Yes	No
Processor	NIOS II/f	NIOS II/f
Hardware coprocessing	No	Yes

Table 6.1: Characteristics of the compared systems

As it can be appreciated, both systems are very similar but, as it will be seen in the next section, performance is very different once the hardware accelerators are working along with the software interface, which can be inferred form the experimental results obtained through the physical prototype of the digital receiver. On the other hand, the synthesis results are taken directly from the synthesis report generated by the Quartus II Development Software.

6.2.1 Performance Results

Performed operations	Implementation				Performance increment
	Software (NIOS II Only)		Hardware accelerated (NIOS II + coprocessors)		
	cycles	time [ms]	cycles	time [ms]	
Vector reshape and arithmetic mean	74824	0.75	2238	0.02	37.5x
1024 points FFT	1591929	15.92	56743	0.57	27.92x
Norm of all the output data from the FFT	3144061	31.44	138511	1.39	22.61x
CFO iterative process	354248859	3542.49	3710676	37.11	95.4x

Table 6.2: Comparisons between software-only and hardware optimized operations

It is important not to forget that the reported times are extracted from real experimental results, in a system with an oscillator of 100MHz. The system can be even faster if an external clock is attached to the board (as the maximum frequency in table 6.2 shows).

6.2.2 Hardware Synthesis Resuls

Implementation	Software (NIOS II only)	Hardware (Nios + coprocessors)
Max. Frequency	223.67MHz	223.67MHz
Space (Total)	29%	87%
ALUTs	10%	80%
Registers	7%	23%
DSP Blocks	10%	100%

Table 6.3: Synthesis Summary

As shown in this last table, a problem with the intensive use of DSP techniques that require the computation of a great quantity of products, is that this situation demanded the use of all the DSP blocks (basically embedded MAC units in the FPGA), which forced some of the multipliers to be created by the use of logic blocks. The result of this is a system that has a smaller maximum frequency than the one that can be obtained in a system where all products are obtained by the embedded multipliers in the DSP blocks.

6.3 Discussion

Before going into a deeper analysis of the obtained results, it is important to highlight the fact that the proposed system demonstrates that it is feasible to build a digital receiver for DDST that can run in the order of the hundreds of MHz, a fundamental requirement that indicates that a commercial implementation of the method is possible.

The results shown in table 6.3 are the perfect example of why the trade-off between performance and FPGA area is so important and how it can render an architecture useless if it is not consciously considered. While the performance increment in the hybrid system is considerable, the difference in the used resources for each versions of the digital receiver is large, overall considering that as more multipliers are implemented by logic blocks, the total performance of the system degrades.

On the other hand, the results in table 6.2 can be discussed in two dimensions: first, under the reasoning of why the comparisons are made between very similar systems, and why a faster general purpose system (like a PC computer with a commercial Intel or AMD desktop processor) was not used. If the DDST were an algorithm designed to work with fixed wireless networks (for example the case of non mobile satellite TV) it could be justified to try the DDST with a powerfull computer, although the power consumption could still be a concern. Nevertheless, mobile devices cannot afford, in terms of cost and power, the use of these processors, so it is almost a fact that the final implementation of the receiver will work, if still being a hybrid system, on a microprocessor very similar to the Nios II. Second, form a perspective of the meaning of the results. It can be seen that the increase in the performance obtained by each accelerator is very different, and for some of them it could be considered low. For this reason, it is important to pay attention to all the columns of the table, because the simultaneous analysis of the consumed time and the speed up. For example, the increment of performance obtained with the arithmetic mean accelerator is of only 37.5, but the time required by the measured stage is of only 0.75 milliseconds in the original software-only system. Moreover, the only way to accelerate more this operations is to process a higher amount of data at the same time, but as this techniques monopolizes the embedded multipliers, it is not an advisable strategy. On the other hand, it is possible to note that the complex exponential coprocessor achieves a performance of almost one hundred times that of the software version. When considering this fact along with the original time required to obtain all the results from this iterative stage (3.54 seconds), it can be seen that the accelerator benefits the full performance of the system significantly. The analysis of the root coprocessor is also important, because the facts behind it must not have to be forgoten. The square root is a very complex operation, and the available solutions are far from optimal. Despite this, the reported implementation increments the results of many other solution while still requiring little space. In addition, the coprocessor could greatly improve its performance if each iteration was performed in a clock cycle. The problem with that solution is that it would reduce the maximum operation frequency for all the system, due to the increase in the worst case clock to output delay time. That situation would not be very critic for the accelerators, as they can also be modified in the same way, but for all the part of the system that still runs as software, a drop in the maximum frequency of operation means that all the instructions will be executed slower.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

An alternative solution that uses both a hardware and a software approach was developed to allow the implementation of a digital communications receiver based on Data-dependent Superimposed Training. A hardware only approach allows the building of fast systems, but problems like the one of the DDST receiver have shown that sometimes the necessary logic for the control of these systems presents a very high complexity. Moreover, when the amount of data to process is too high, and many data dependencies are present, techniques like parallel processing or pipelining are difficult to exploit, usually leading to architectures so large they cannot fit into mid-range FPGAs.

It can be also concluded that, in communications algorithms in which operations like FFTs, matrices multiplications, averages, and norms, among others, are needed, DSP blocks and, overall, embedded multipliers are a very valuable resource, as it is not possible to expect the same speed from a multiplier that has been implemented with logic blocks available in the FPGA; as the dedicated multipliers are assigned to some of the built accelerators, the synthesis tool has to use the logic blocks to implement the rest of such multipliers, an issue that impacts directly the maximum frequency of the system. This situation is very different from that of the memory, due to the multiple options that can be found nowadays in FPGA boards. It is true that the memory blocks inside the reconfigurable chip present the smallest latency, but a reduction in the access speed of off-chip memories (SRAM, SDRAM) can be tolerated if the trade-off allows the storage of a significantly bigger amount of data. Furthermore, experimental results show that the difference of maximum frequency between on-chip and off-chip memories are not very significant, if techniques as the fetching of several data at the same time (like in the case of the arithmetic mean accelerator) are used.

It was shown that it is possible to analyze a software-only code to detect the critical sections that can be translated into faster and more accurate hardware coprocessors, which can both be managed by the central microprocessor and operate independently, accessing the memories in the system without the necessity of interrupting the other components. In fact, this procedure can be written as a method that can be used for the implementation of *proof of concept* designs of many algorithms that share some of their characteristics with the DDST. From the present work, a method as the one that follows offers the possibility of having a prototype in little time:

1. Translation of the algorithm into a series of programs written in C language.

2. Optimization of the functions (when possible) through the use of mathematical transformations.
3. Analysis of data dependencies.
4. Search of parallelizable operations.
5. Selection of possible candidates for hardware acceleration.
6. Complexity and average resources usage analysis of the candidates.
7. Decision of the type of accelerator.
    - HDL accelerator.
    - C2H accelerator.
8. Building of the accelerators.
9. Individual debugging.
10. Building of the whole system.
11. Final debugging.

Of course, this method depends on the Altera technology, but using a combination of other tools, the same steps can be followed. For example, the combination of Impulse C with Xilinx ISE and XPS, or the use of Celoxica's Handel C along with the tools of Xilinx or Altera.

With this work it was possible to detect the most problematic stage of a receiver based on DDST (the Carry Frequency Offset Estimation). Nevertheless, the use of FFTs and a look-up table / parallel / iterative norm accelerators made the calculation of trigonometric functions (sines and cosines) the only obstacle left in order to obtain a practical system for DDST. Among the studied alternatives for this last problem, the Chebyshev polynomials represented the most suitable solution, taking into account the resources offered by the specific FPGA board used.

The hybrid software-hardware approach demonstrated to be very versatile and flexible, allowing fast implementation of several kinds of algorithms and their fast modification, from a small change in the input parameters values to the alteration of a full stage of the process. In fact, the built prototype fits perfectly into the study of the DDST algorithm, as this algorithm is still under study and constant modifications and improvements have been made over it. For example, the first versions of superimposed training worked under the time domain, so operations like the FFT were not necessary. If future changes in the algorithm require a significant modification of any of the accelerators, it will be very easy to adapt the whole system.

7.2 Future Work

The problem of the complex exponentials is yet under study. According to the space and speed necessities of the final receiver, it could be decided to change the Chebyshev solution by an iterative one. Therefore, the real performance of CORDIC and BKM generators has yet to be compared to the current solution. It is true that a full pipelined implementation of a CORDIC generator can be complex, but it would also have the advantage of obtaining one result each cycle once all its stages were filled. On the other hand, this solution would require a change in the rest of the CFO section, as each obtained sine or cosine is multiplied by an element of an stored vector before obtaining a new complex exponential. Here a solution would require an additional memory to store all the sines and cosines before the respective multiplications.

To improve the performance of the whole system, the use of a DMA module is also under study, in order to reduce the time consumed in memory accesses. Combined with the FFT and the arithmetic mean stage, this improvement could increase the system performance by maybe more than one order of magnitude. Notwithstanding, this would require extra FPGA resources, when the actual prototype has already occupied more than 80% of the available logic elements.

A necessary, but more complex task left, is the implementation, if possible, of a full hardware system, where the NIOS II processor would be replaced by a custom control unit, now that the most complex stages have been identified and solved by the use of the four coprocessors. Under this approach, any accelerator created by the C2H tool is not usable, so part of the flexibility of the system would be lost. Nevertheless, there are several options to replace at least the FFT accelerator, like the IP cores available in both Altera and Xilinx tools. There are also several working implementations available from resources like opendevices.org, and the option of creating a custom accelerator is also present. One of the challenges of a hardware only solution is the management of all the data that is operated along all the receiver stages.

If the hardware-only option results unfeasible, another important change that could be made on the system is the replacement of the NIOS II by another available processor, that in the case of being open, could be modified to meet the exact requirements of the

DDST receiver. Once again, this means that the C2H created accelerators cannot be used, but it also makes possible to reduce the space occupied by the system as the processor would contain just the minimum necessary elements to control the receiver.

To analyze and improve the accuracy of the receiver is also a task that has not been performed with enough deepness. The lengths of the majority of the data are of 32 bits, but if the change to a 64 bits resolution demonstrates to improve satisfactorily the accuracy of the whole system, this is a change worth the extra data size and the lost of the uniformity of the software part of the receiver. At this moment, the accuracy lost due to data trunctations, averages, or divisions in all of the accelerators is not uniform with respect to the software only functions.

Among all these mentioned tasks, it is important to mention that only the implementation of a full hardware receiver and the accuracy analysis based on the data lengths are necessary to conclude which is the best possible implementation of the DDST algorithm in a practical system. Between these two activities, the full hardware architecture represents the only challenge left, as its building could require up to six months, considering the necessary tests that would have to be performed. All the other suggestions, along with the length analysis, can be performed maximum in two or three months, even those that imply the removal of the c2H capabilities, as there are already several options available to replace the functions performed by this compiling tool.

List of Figures

List of Tables

Bibliography

[1] Marcelino Lázaro, Ignacio Santamaría, Deniz Erdogmus, Kenneth E. Hild, Carlos Pantaleón, Jose C. Principe: *Stochastic Blind Equalization Based on PDF Fitting Using Parzen Estimator*, IEEE Transactions on Signal Processing, Vol. 53, no. 2, February 2005.

[2] Gunnar Heine, Matt Horrer, GSM Networks: Protocols, Terminology, and Implementation, Artech House, 1st. edition, ISBN: 978-0890064719, pags. 321-325.

[3] Aldo G. Orozco-Lugo, M. Mauricio Lara, and Des C. McLernon: *Channel Estimation Using Implicit Training, IEEE Transactions on Signal Processing*, Vol. 52, no. 1, January 2004.

[4] E. Alameda-Hernandez, D. C. McLernon, M. Ghogho, A. G., Orozco-Lugo and M. Lara: *Improved Synchronisation for Superimposed Training Based Channel Estimation*, IEEE/SP 13th Workshop on Statistical Signal Processing ,2005

[5] Simon Haykin: *Digital Communications*, ISBN 0-471-62947-2, 1988, pags. 99-105.

[6] Bernard Sklar: *Digital Communications, Fundamentals and Applications*, Prentice Hall, Second Edition, ISBN 0-13-084768-7, pags. 136-160.

[7] Andy Bateman: *Digital Communications: Design for the Real World*, Prentice Hall, 1998, ISBN: 978-0201343014, pags. 25-27.

[8] Vijay K. Madisetti, Douglas B. Williams: *The Digital Signal Processing Handbook*, IEEE Press & CRC Press, 1999, ISBN: 0-8493-8572-5, 24-1, 24-5, 68-11.

[9] B. Farhang-Boroujeny: *Pilot-Based Channel Identification: Proposal for Semi-blind Identification of Communication Channels*, IEEE Electronics Letters, Vol. 31, No. 13, June 1995, pp. 1044-1046.

[10] Enrique Alameda-Hernandez, Desmond C. McLernon, Aldo G. Orozco-Lugo, Mauricio Lara and Mounir Ghogho, Synchronization for Superimposed Training Based Channel Estimation, Electronics Letters, Vol. 41, No. 9, April 2005, pp. 565-567.

[11] Enrique Alameda-Hernandez, Desmond C. McLernon, Aldo G. Orozco-Lugo, Mauricio Lara and Mounir Ghogho: Synchronisation and DC-Offset Estimation for Channel Estimation Using Data-Dependent Superimposed Training, EUSIPCO 2005, Turkey, September 2005.

[12] F. Mazzenga, *Channel Estimation and Equalization for M-QAM Transmission with a Hidden Pilot Sequence*, IEEE Transactions on Broadcasting, Vol. 46, June 2000, pp. 170-176.

[13] G. T. Zhou, M. Viberg and T. McKelvey, *Superimposed Periodic Pilots for Blind Channel Estimation*, IEEE Conf. Rec. Thirty-Fifth Asilomar Conf. Signals, Syst., Comput., Vol. 1, Nov. 2001, pp. 653-657.

[14] Jitendra K. Tugnait and Weilin Luo, On Channel Estimation Using Superimposed Training and First-Order Statistics, IEEE ICASSP 2003, Vol. IV, pp. 624-627.

[15] Jitendra K. Tugnait and X. Meng, Synchronization of Superimposed Training for Channel Estimation, IEEE ICASSP 2004, Vol IV, pp.853-856.

[16] Mounir Ghogho, Des McLernon, Enrique Alameda-Hernandez and Ananthram Swami, Channel Estimation and Symbol Detection for Block Transmisión Using Data-Dependent Superimposed Training, IEEE Signal Processing Letters, Vol. 12, No. 3, March 2005, pp. 226-229.

[17] Duong H. Pham and Jonathan H. Manton, Orthogonal Superimposed Training on Linear Precoding: A New Affine Precoder Design, IEEE 6th Workshop on Signal Processing Advances in Wireless Communications, June 2005, pp. 445-449.

[18] Xiaohong Meng and Jitendra K. Tugnait Semi-Blind Channel Estimation and Detection Using Superimposed Training, IEEE ICASSP 2004, Vol. IV, pp. 417-420.

[19] Jitendra K. Tugnait and Xiaohong Meng On Superimposed Training for Channel Estimation: Performance Analysis, Training Power Allocation, and Frame Synchronization, IEEE Transactions on Signal Processing, Vol. 54, No. 2, February 2006, pp. 752-765.

[20] Sung-Yoon Jung and Dong-Jo Park, Linear MMSE Receiver Using Hidden Training Sequence in DS/CDMA, Electronics Letters, Vol. 39, No. 9, May 2003, pp. 742-744.

[21] Ning Chen and G. Tong Zhou, Superimposed Training for OFDM: A Peak-to-Average Power Ratio Analysis, IEEE Transactions on Signal Processing, Vol. 54, No. 6, pp. 2277-2286, June 2006.

[22] Xiliang Luo and Georgios B. Giannakis, Low Complexity Blind Synchronization and Demodulation for (Ultra-) Wideband Multi-User Ad Hoc Access, IEEE Transactions on Wireless Communications, Vol. 5, No. 7, pp. 1930-1941, July 2006.

[23] Jibing Wang and Xiaodong Wang, Superimposed Training-Based Noncoherent MIMO Systems, IEEE Transactions on Communications, Vol. 54, No. 7, pp. 1267-1276, July 2006.

[24] Mounir Ghogho, Des McLernon, Enrique Alameda-Hernandez and Ananthram Swami, SISO and MIMO Channel Estimation and Symbol Detection Using Data-Dependent Superimposed Training, IEEE ICASSP 2005, Vol III, pp. 461-464.

[25] Haidong Zhu, Behrouz Farhang-Boroujeny and Christian Schlegel, Pilot Embedding for Joint Channel Estimation and Data Detection in MIMO Communication Systems, IEEE Communications Letters, Vol. 7, No. 1, January 2003, pp. 30-32.

[26] Aldo. G. Orozco-Lugo, Giselle M. Galvan-Tejada, M. Mauricio Lara and Desmond C. McLernon, A New Approach to Achieve Multiple Packet Reception for Ad Hoc Networks, IEEE ICASSP 2004, Vol. IV, pp. 429-432.

[27] Mounir Ghogho, Ananthram Swami, Channel estimation for MIMO systems using data-dependent superimposed training, Allerton Conference , September 29 - October 01, 2004

[28] Aldo. G. Orozco-Lugo, Giselle M. Galvan-Tejada, M. Mauricio Lara and Desmond C. McLernon, A low complexity iterative channel estimation and equalisation scheme for (data dependent) superimposed training, Proc. European Signal Processing Conference (EUSIPCO 2006), Florence, Italy, 4-8 September, 2006.

[29] Aldo. G. Orozco-Lugo, Giselle M. Galvan-Tejada, M. Mauricio Lara and Desmond C. McLernon, Frame / Training Sequence Synchronization and DC-Offset Removal for (Data-Dependent) Superimposed Training Based Channel Estimation, IEEE IEEE Transactions on Signal Processing, Vol. 55, No. 6, June 2007.

[30] Orozco-Lugo, A.G.; Lara, M.M.; Alameda-Hernandez, E.; Moosvi, S.; McLernon, D.C.: *Frequency Offset Estimation and Compensation Using Superimposed Training*, Electrical and Electronics Engineering, 2007. ICEEE 2007. 4th International Conference on, Vol. 5, No. 7, Sept. 2007, pp. 118 - 121.

[31] John Proakis, Digital Communications, McGraw Hill, 4th edition, ISBN: 9780072321111, pags. 601-617.

[32] B. P. Lathi, Modern Digital and Analog Communication Systems, Oxford University Press, 3rd edition, ISBN: 0-19-511009-9, pags. 323-325.

[33] Lawrence E. Larson, RF and Microwave Circuit Design for Wireless Communications, Artech House Publishers, 1st edition, ISBN: 0-89006-818-6, pags. 73, 75-76.

[34] David M. Pozar, Microwave and RF Design of Wireless Systems, John Wiley & Sons, 2001, ISBN: 0-471-32282-2, pags. 309.

[35] Magdy A. Bayoumi, Earl E. Swartzlander, Jr., VLSI Signal Processing Technology, Kluwer Academic Publishers, 1994, ISBN: 0-7923-9490-9, pags. 1-23.

[36] Markus Helfenstein, George S. Moschytz, Circuits and Systems for Wireless Communications, Kluwer Academic Publishers, 2000, ISBN: 0-7923-7722-2, pags. 265-267, 291.

[37] U. Meyer-Baese, Digital Signal Processing with Field Programmable Gate Arrays, Springer, Third Edition, ISBN: 978-3-540-72612-8, pags. 1-18, 120-141.

[38] David A. Patterson, John L. Hennessy, Computer Organization adn Design *The hardware/software interface*, Morgan Kaufmann, Third Edition, ISBN: 1-55860-604-1, pags. 370-374.

[39] Richard J. Mammone, Computational Methods of Signal Recovery and Recognition, Wiley Series in Telecommunications, 1992, ISBN: 0-471-85384-4, pags. 164-168.

[40] Altera Corporation (May 2006): *Accelerating Nios II Systems with the C2H Compiler Tutorial*
Retrieved 28th March, 2008, from
`http://altera.com/literature/tt/tt_nios2_c2h_accelerating_tutorial.pdf`

[41] Altera Corporation (2008): *Nios II Processor Reference Handbook*
Retrieved 22nd June, 2008, from
`http://altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf`

[42] Altera Corporation (August 2006): *Stratix II DSP Development Board Reference Manual*
Retrieved 12th August, 2008, from
`http://altera.com/literature/manual/mnl_SII_DSP_RM_11Aug06.pdf`

[43] Piromsopa, K, Aportewan, C. and Chongstitvatana, P.: An FPGA implementation of a fixed-point square root operation, Inter. Symposium on Communications and Information Technology, Vol. 14-16, Thailand, 2001, pp. 587-589.