Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 912301, 10 pages
doi:10.1155/2009/912301

Research Article

A System on a Programmable Chip Architecture for Data-Dependent Superimposed Training Channel Estimation

Fernando Martín del Campo,[1] René Cumplido,[1] Roberto Perez-Andrade,[1] and A. G. Orozco-Lugo[2]

[1] *Computer Science Department, National Institute of Astrophysics, Optics and Electronics, CP 72840, Puebla, Mexico*
[2] *Section of Communications, CINVESTAV-IPN, CP 07360, Mexico City, Mexico*

Correspondence should be addressed to Fernando Martín del Campo, fmartin@ccc.inaoep.mx

Received 25 December 2008; Accepted 25 May 2009

Recommended by Peter Zipf

Channel estimation in wireless communication systems is usually accomplished by inserting, along with the information, a series of known symbols, whose analysis is used to define the parameters of the filters that remove the distortion of the data. Nevertheless, a part of the available bandwidth has to be destined to these symbols. Until now, no alternative solution has demonstrated to be fully satisfying for commercial use, but one technique that looks promising is superimposed training (ST). This work describes a hybrid software-hardware FPGA implementation of a recent algorithm that belongs to the ST family, known as Data-dependent Superimposed Training (DDST), which does not need extra bandwidth for its training sequences (TS) as it adds them arithmetically to the data. DDST also adds 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 too high for the commercial processors used in mobile systems, a System on a Programmable Chip (SOPC) approach is used in order to solve the problem.

1. Introduction

The air is inherently noisy and its nature can contribute to the presence of different kinds of interference, as the one known as Intersymbol Interference or ISI, in which the energy of the message symbols is spread in such way that a part of each symbol 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. Nevertheless, this channel can be modeled as a linear system, whose effects can be reverted in the receiver, if one knows its parameters with enough precision.

To obtain these parameters, the majority of the digital wireless communication systems use sequences of known symbols that are also called training sequences. These groups of symbols, after a certain analysis, allow the estimation of the communication channel. Once this has been performed, the original information can be extracted, using well-known mathematical formulas and DSP techniques for data recovery.

The most extended technique to integrate the training sequences to the information is known as *time-division multiplexed channel estimation or time-division multiplexed training (TDMT)*, where some of the transmission slots are used for the pilots or training symbols [1]. The performance of this approach is very high, but it has the disadvantage of needing part of the available bandwidth to accommodate the extra data. Even though several options have been proposed, none of them have demonstrated to be more feasible than the usual training.

One of the most promising techniques that has not yet been implemented physically is *Superimposed Training (ST)*, where the training sequence is arithmeticly added to the information, saving the necessity of more bandwidth at the expense of a little power loss on the information signal [2, 3].

The method named *Data-dependent Superimposed Training* goes beyond ST, adding another sequence (the data-dependent training sequence) to the information. When estimating the channel, what is analyzed is the training sequence so, from this point of view, the original data can

be considered additive noise that distorts the object of study. The data-dependent sequence cancels the contribution of the input signal at the frequency bins at which the training sequence has energy, improving the channel estimates over ST [4].

The problem with DDST, talking about its possible implementation, is its high computational complexity, that renders the commercial mobile and low power demanding processors useless for this purpose, at least taking into account the performance demanded by the systems in which it could be used. Even though simulations with the DDST method show a performance that can compete with the TDMT [4–6], 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, with both fixed point and simple precision floating point (32 bits) arithmetic. Resulting speeds from these architectures (specially the floating point one) are in the order of kHz, so, even when they are useful for error comparison, it is impossible to use them in commercial systems.

This work will describe a combined software/hardware implementation of the DDST algorithm using an SOPC approach, highlighting the most challenging issues that have arisen, and the way in which they have been tackled. It is true that obtaining a fast and functional DDST solution is a highly complex task, but the promise of a larger available bandwidth for the information is very attractive. 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 has a special interest, 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, unfeasible, 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 parallel processing and pipelining) and the complex control required by some of the mathematical operations that have to be performed. It is true that some of the top branch FPGAs available form Altera and Xilinx can accommodate a full hardware architecture, and in fact a solution like this one is the final goal of a DDST receptor, but actually, two problems arise from this option: first, the transformation of the FPGA prototype into an ASIC solution can be far from optimal, as it would be neither cheap nor low power consuming, overall because the architecture presents several DSP like structures, that usually do not map very well to ASIC implementations [7]. The second problem with this solution is the highly complex control, that is, necessary to process the amount of data that a digital receptor usually must handle. As this work presents the first implementation of the DDST channel estimation algorithm on an FPGA, a hybrid software/hardware implementation presents the advantage of a simple control through a series of software coded instruction and the power of dedicated hardware coprocessors to perform the most time and resources demanding tasks of the DDST receiver stages for the obtaining of the estimate.

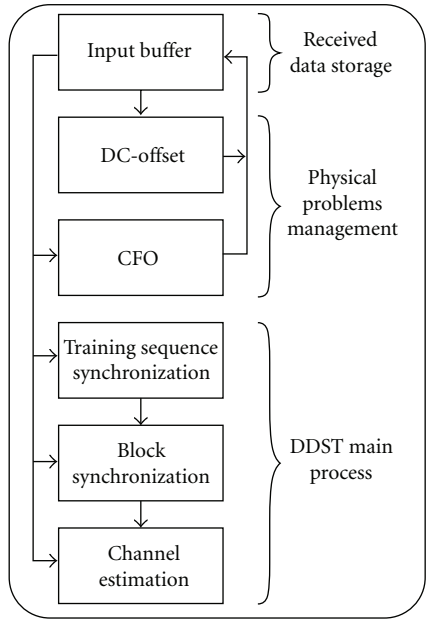

Figure 1: DDST general block diagram.

2. DDST Algorithm Review

Figure 1 shows a general block diagram of a digital communication receiver based on DDST.

Before describing each block, it is important to note that they cannot be executed in a parallel fashion, that is, to start the process of each block, it is absolutely necessary that all the previous stages have already finished their own function. The lines with the arrow markers indicate from where each block receives its input and to where it feeds its output.

It is also important to mention that the DC-offset, along with the two synchronization steps, and the channel estimation itself, exploit the *cyclostationarity,* that is, induced in the transmitted signal, when superimposed training is employed [2, 3].

This work does not focus on the mathematical meaning of the formulas used to solve the different steps of the DDST approach, but in the computational burden that they present and in the procedure followed to implement them in the system.

2.1. Input Buffer. To begin with the steps of the algorithm, it is necessary to store the input data samples as they are being received, because it is not possible to correct them "on the fly." As shown in Figure 1, the input data samples are

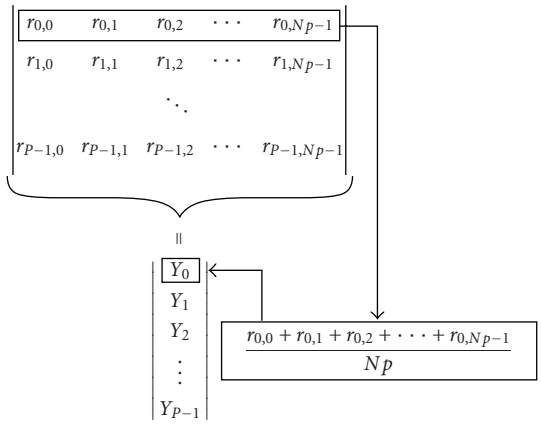

Figure 2: Arithmetic mean from the input buffer.

corrected several times as the different stages of the DDST channel estimation are fulfilled.

Before going further into the process, it is fair to mention that all data samples are complex valued quantities, so all operations performed in the following steps involve the computation of both a real and an imaginary part of several numbers.

2.2. DC-Offset Estimation and Correction. Practical systems commonly face a physical problem resulting from the building techniques used; their output, seen as voltage levels, presents an unwanted constant value, that is, added to the expected signal. Even though this value is almost always very small, it must be considered as the method works with first order statistics [2].

This block involves the reshaping of a vector formed by an N size subset of the original input data into a matrix of size $\|P \times (N/P)\|$, whose rows are then summed to form a vector of length P. Each element of the vector is then multiplied by $1/(N/P)$ so, in fact, each element of the output vector corresponds to the arithmetic mean of each of the rows of the matrix mentioned above. This process is shown in Figure 2.

Once this vector has been obtained, the process reaches an iterative phase that involves matrix multiplications, norm of a vector (the square root of the sum of the squares of the real and imaginary parts of each element of the vector), several other multiplications, and a division. All of these operations have a high computational complexity, where the square root and the matrix multiplication are the more challenging. Once the DC-offset has been obtained, it is removed from the input data by simply subtracting it from each input element.

2.3. Carry Frequency Offset (CFO) Estimation and Correction. Due to the lack of perfect oscillators and because of Doppler shifts, receivers in practical pass-band systems always experiment a carry frequency offset [8]. Among the mathematical operations found in a DDST-based digital communications receiver, CFO estimation has the more complex of all, both in time and resource usage. This block requires several summations, Fast Fourier Transforms and vector norms, but even these operations result insignificant when compared with the real problem of this stage: to obtain a CFO estimate, a simulation run, for example, needed to calculate 36864 complex exponentials. If they are solved using the Euler's Formula, 73728 trigonometric operations have to be performed. As the CFO estimation is an iterative process, the complexity is not the only problem, because low data resolutions lead to fast growing errors, that in the end can result in a very inaccurate estimate. Once this process is complete, the CFO is removed from the input data by multiplying each input sample by one complex exponential term.

2.4. Training Sequence Synchronization Estimation. In practical applications, it is usually impossible to suppose a perfect synchronization between the transmitter and the receiver at the training sequence level, so channel estimation must consider this issue.

When there is no perfect synchronization, the estimate is just a circular shifted version of the real channel estimation that would have been obtained under ideal conditions. Using the cyclostationarity of the signal, 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. Mathematical operations in this stage are almost identical to those performed for the dc-offset estimation [5].

2.5. Block Synchronization Estimation. As with the training sequence estimation, block synchronization is also based on the particular structure of the channel output's cyclic mean vector, and can be achieved even in the presence of a DC-offset. Due to its special characteristics, in DDST it is not enough to locate the start of a training sequence period, because it is also necessary to find the start of each received block. Only the vector encompassing a full DDST block will provide a cyclostationary mean vector independent from the data sequence, with a reduced "data" noise compared to the rest of the estimates. This procedure is achieved through a specific cost function, that will give a minimum value only with the right version of the cyclostationary mean vector [5]. Even though this block is, in concept, very different from the Training Sequence Synchronization estimation, the necessary operations for the Block Synchronization Estimation also include matrix multiplications, norm of a vector, and other multiplications.

2.6. Channel Estimation. Once the two synchronization estimations have been obtained, the channel estimation stage can be tackled with very similar operations to those that have already been used. In fact, this easy step only needs a vector reshape, the mentioned vector obtained from the arithmetic mean of that reshaped matrix, and a matrix multiplication. At the end, what is obtained is a vector whose complex elements correspond to the values of each tap of the estimated channel.

Table 1: Computational complexity of the stages of the DDST channel estimation algorithm.

Stage	Complexity
Input buffer	$O(1)$
DC offset	$O(n^2)$
Carry frequency offset	$O(n)$
Training sequence synchronization	$O(n^2)$
Block synchronization	$O(n^2)$
Channel estimation	$O(n^2)$

2.7. Computational Complexity Review. To show the problems generated by each of the different stages of the DDST channel estimation algorithm, from a different point of view, Table 1 enlist their computational complexity. Nevertheless, this approach is deceptive, as it supposes that the atomic operations in the process consume the same amount of time. For example, a matrix multiplication requires two nested cycles, but the basic operation of the multiplication presents an $O(n)$ complexity, so, in fact, the full operation presents an $O(n^3)$ complexity. Another example is the square root operation, which presents an $M(n)$ complexity, or the FFT calculation, with $O(n \log_2 n)$. Moreover, parameters as P and N from the algorithm are variable, so it is very difficult to give an idea of the real magnitude of the problem in terms of computational complexity. Section 6 tackles this issue by giving the results from the implementations in terms of consumed time and necessary clock cycles to fulfill the operations of each stage.

3. DDST and Its Hardware Implementation

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 performed tests 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.

At this point, it can be inferred that the difficulty for implementing the algorithm in hardware is caused by two main reasons: the complexity of the operations (square root and trigonometric functions are far from an optimal hardware solution), and the amount of data that is used, transformed, and updated constantly. The first of these issues leads to the generation of huge hardware components, that usually need several multipliers, units that are scarce in mid-range FPGAs. The second one requires a very complex control unit and the need of a high amount of memory accesses.

Moreover, the huge amount of data dependencies (operations that require several previous results from past stages) makes it very difficult to use techniques such as parallel processing and pipeline implementation. Even in those few cases when it is possible to identify those stages of the process in which either the parallel operations or the pipelining is feasible, their performance gain versus that of a software only implementation can be small, and they usually require a considerable FPGA area, due to the amount of information that has to be operated in a concurrent fashion.

This implementation tackles the problem by using a hybrid software/hardware approach. A set of C language programs running over an *NIOS II soft processor* spare the need of a complex control, while dedicated hardware coprocessors perform the most time and resource demanding operations (like the FFTs), also allowing operations with nonstandard data lengths (e.g., 48 bits) that are hidden to the C programs of the system, making it easier to control, modify, and extend the software section of the SOPC. 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 and high speed processing of these last ones. Nevertheless, the PCs are also outperformed in applications where parallel or pipelined processing of large amounts of data is required, or in those where price, size, and power consumption constraints are very strict. These are the reasons why the proposed solution tries to take advantage from both approaches generating a specific purpose SOPC as a proof-of-concept solution for the DDST problem.

4. Hardware Architecture

Figure 3 depicts the DDST hardware architecture. It has been designed to run in an Altera Stratix II FPGA as a system controlled by an NIOS II. As it can be seen, the architecture resembles that of a common computer system, with the difference that it presents several dedicated memories for fast data fetching and processing, and a set of special hardware accelerators that interact directly with the rest of the system. The processor only passes them certain parameters and activates them (through the use of their slave ports), but they execute its processing in a stand alone fashion. This means that they can read and write, using the master ports, all the memories in the system (although they almost always interact with the dedicated ones) and, while one of them is working, neither the processor nor do the other accelerators need to be interrupted. The control necessary to avoid the resource competition is performed by the software section of the system.

4.1. NIOS II Soft-Core Embedded Processor. The NIOS II from Altera is a soft-core microprocessor that, for the DDST implementation, presents several advantages, like its low power consumption and its small required FPGA area, along with the ease to interact with custom hardware accelerator modules.

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 allows to even add new

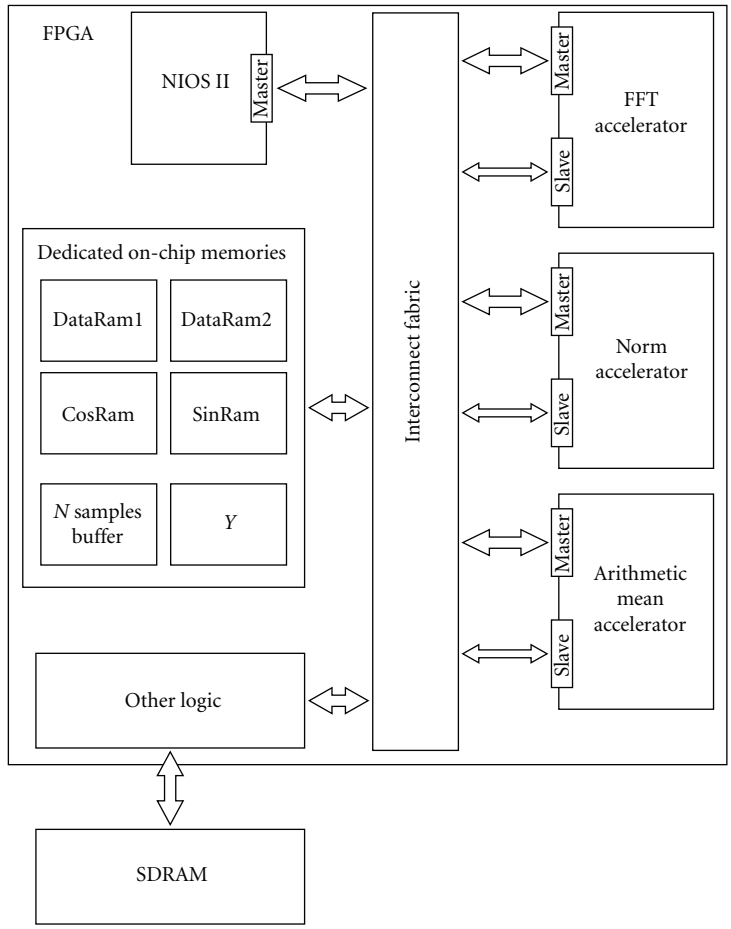

Figure 3: Hardware architecture of the system.

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 through a relatively transparent interface written in C language, or even in assembler.

4.2. Dedicated On-Chip Memories. The on-chip memories are structures that allow the transparent management and use of the memory blocks contained inside the FPGAs. They have the smallest latency (1 cycle) of all the available memories in the Altera boards. 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 declaring a pointer to its assigned address, inside a typical C language program, that then will be compiled for the soft processor architecture. Dedicated memories DataRAM1, DataRAM2, CosRAM, and SinRAM are used by the FFT accelerator, while the N Samples Buffer and Y are accessed by the other two coprocessors. The NIOS II can access all of them.

5. 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 an SOPC that runs a series of C programs, but leaves the most *computer intensive* or *memory demanding* operations to special hardware accelerators.

Contrary to the majority of *Systems on a Chip* and common computer systems, Altera does not use a conventional bus scheme. The systems built on the manufacturer's programmable hardware feature a switch interconnect fabric (Avalon) 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. The Avalon 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 three accelerators in Figure 3 (that will be explained in the following sections) 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 to add ADCs to directly digitize the received data is being studied.

5.1. FFT Accelerator. The FFT coprocessor is based on an original design of Altera that uses a tool named C2H to convert C language instructions to hardware elements directly. There are other options (to use an IP core or a custom hardware design) that can achieve a better performance than this solution, but the C2H solution shows an interesting capability of the system being built: if the DDST algorithm is modified to have a better performance or to reduce the computational complexity of a certain step, it is possible to implement such change in the architecture by just modifying some lines of code in a C program, instead of redesigning a full hardware accelerator. As it is, the FFT coprocessor executes its function with acceptable speed and area consumption. This implementation uses a decimation in time FFT algorithm known as decimation in time Cooley-Turkey. Results are obtained through the use of 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 Nios II C2H Compiler maps ANSI C constructs directly to RTL. For example, an if construct will be mapped directly to a multiplexer, and a multiplication will be mapped to one of the available embedded multipliers.

The FFT module is different from the other hardware accelerators programmed directly with VHDL, and has three main advantages: first, modifications over the code are very easy to perform, and even a programmer that does not know anything about the system can perform them. Second, the complex control that the algorithm requires is automatically generated by the tool. This is important overall taking into account that the optimizations of the method require several accesses to four different memories (two for the real part of the data and another two for the imaginary one) in the ping-pong buffering stage. 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.

The implementation of this kind of FFT algorithm consumes significant FPGA memory resources, because of the amount of data that have to be temporarily stored for the ping-pong method. Moreover, two extra memories are necessary to store the twiddle factors, that are basically two sets of prestored values from sine and cosine calculations, that are used to combine successive FFT results. Consequently, to implement two FFT modules would consume resources that could be used 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, this result is preferred to the extra FPGA area that the FFT modules would consume for a faster implementation.

5.2. Arithmetic Mean Accelerator. In the low complexity blocks of the DDST SOPC, 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, but to concentrate the efforts on the other steps necessary for this kind of block (dc-offset, TSS, block synchronization, and 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. As mentioned in Section 2, the reshape operation is performed constantly along the DDST execution, and then followed by the arithmetic mean of the rows of the resulting matrix. At the end, the result is a vector of P elements, where P is the length of the training sequence.

This coprocessor, whose block diagram is shown in Figure 4, performs the summations over the vector to reshape in parallel, sending the result to a dedicated memory that corresponds to the vector of P elements. In this way the step of the reshaping can be bypassed and the execution time is reduced considerably.

As Np is a parameter that has to be known before computing, then its inverse can also be previously calculated, and the P necessary divisions can be performed as multiplications by $1/Np$, accelerating the process. In fact, many of

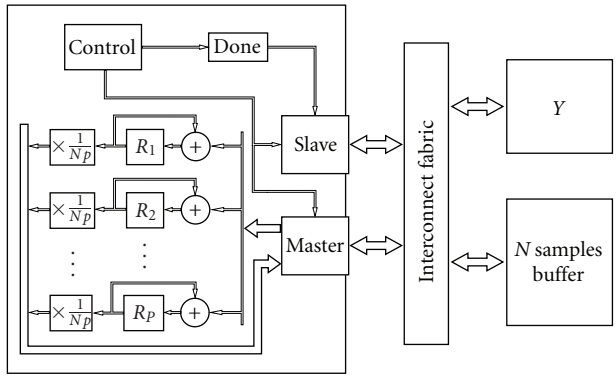

Figure 4: Arithmetic mean hardware accelerator.

the parameters used along the algorithm are also previously known, so it is possible to reduce the time required by several operations.

Summarizing, this coprocessor reads directly from the *N samples buffer* dedicated on-chip memory, **P** data, each one of 32 bits, accumulating their respective values to **P** registers of 32 bits width. At the end of the process, they are multiplied by $1/(N/P)$, so now they contain the arithmetic means of the rows from the reshaped matrix. Finally, the results are stored in another on-chip memory.

As it will be seen in the results section, this accelerator outperforms a software only version by more than 30 times, giving also a smaller error as it works with higher resolutions, during the multiplication stage.

5.3. Norm or Magnitude Accelerator. 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. Sometimes, as in the DC-offset estimation, it is possible to work with the square norm of the complex samples, but this is not the case in steps like the CFO estimation.

The norm of a complex number $a + bi$ is calculated as in (1) and it represents the magnitude of that number,

$$\sqrt{a^2 + b^2}. \tag{1}$$

The computational burden of this operation is high, as it does not only need to multiply the real and imaginary part by themselves, but also to obtain a square root. This last operation is difficult to implement with the required speed both in software and hardware, and the algorithms used for this computation are iterative, involving the use of multiplications, substractions, and comparisons.

This problem is solved by using an accelerator that has several advantages over the software-only version: it fetches both the real and the imaginary part of the FFT elements each time it performs a read operation; it works with 64 bits arithmetic, so there is no loss in the accuracy of the result. In addition, it accumulates the calculated magnitudes as it works, so at the end of the process this section of the system will have the summation of all of the results, a parameter needed for the iterative part of the CFO block. Finally, it is possible to assign an *"offset"* so, for example, the module can obtain only the norm of the complex samples in positions $0, 4, 8, \ldots$, and so forth, and not from every sample in the input vector. Figure 5 shows the block diagram of the accelerator. The operation of the square root block will be explained in the following section. The magnitude accelerator calculates all the norms of the complex samples in a vector V of n elements (as shown in (2)),

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

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

As it can be seen, it is possible to send to the N Samples Buffer the result of each of the magnitudes as they are obtained, or their total summation. These decisions are fed to the module by the software program, along with the address of the memory and the amount of complex numbers to process (e.g., 1024 numbers resulting from the FFT).

5.3.1. Square Root Hardware Submodule. The square root is solved by a submodule with an operation based on the nonrestoring algorithm implementation, like the one of [9], 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 *adjusted*. For example, the square root of 5 is ≈2.236. A value of 2 would be obtained from the look-up table, so only the decimal part of the result would have to be calculated. This increases the speed of the coprocessor drasticly while still using very little FPGA area. Second, each iteration calculates, in parallel, 4 bits of the root, and not only one. This system is depicted in Figure 6.

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.

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 the entries take the same amount of cycles to be calculated. For example, if we try to find the root of 14.0625, 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 $\approx 7.15 \times 10^{-7}$.

It is important to consider the bit length 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 length grows, the necessary size of the memory to store it also increases

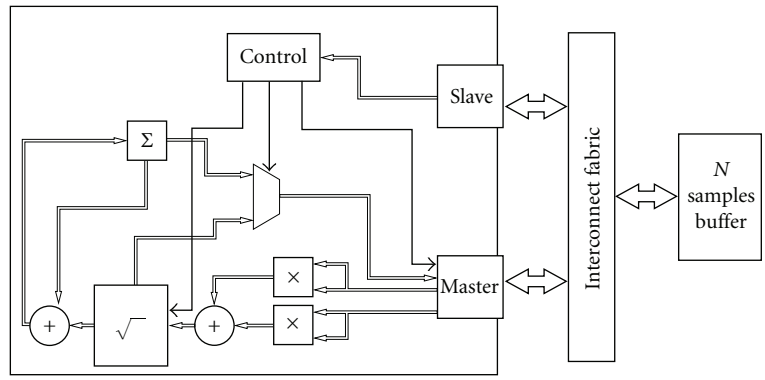

Figure 5: Magnitude hardware accelerator module.

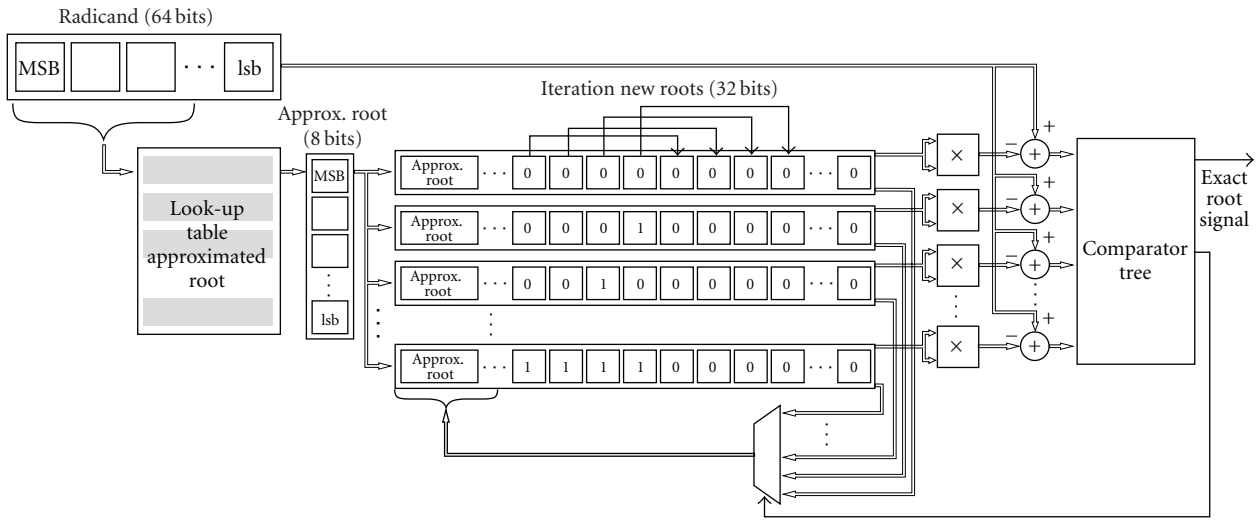

Figure 6: Square root submodule.

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 tradeoff, that is, particularly important in the case of architectures based on FPGAs.

6. Results

The hybrid architecture was compared against an optimized software-only version of the DDST algorithm implementation. The system for this nonhardware implementation is similar to the accelerated version, but it does not have the coprocessors or the dedicated on-chip memories. The reason for this decision is that a commercial implementation of the algorithm would run in a microprocessor built for mobile devices, with characteristics very similar to those of the NIOS II.

Both systems were tested with a set of 3765 complex data, with 32 bits for the real and imaginary part, respectively. In fact, lower resolutions (e.g., 16 bits) degrade the performance so drasticly that the obtained results are far too different from the expected ones (obtained from a Matlab simulation using double precision floating data). Execution time was measured using the Altera module *performance counter*, that physically times a group of code lines and outputs both seconds consumed and cycles taken for the operation to finish.

Table 2: Synthesis summary.

Implementation	Software	Hardware
Max. frequency	238.72 MHz	238.72 MHz
Space (total)	29%	68%
ALUTs	15%	60%
Registers	13%	24%
DSP blocks	10%	100%

Table 2 shows an abstract of the synthesis report for the software-only and hybrid systems. Space percentages refer to an estimate that the synthesizer makes according to the total available resources. The equal frequencies are expected as all the coprocessors would run at higher speeds than the NIOS II if working in a standalone way. When functioning together, the slowest device (the NIOS II microprocessor) determines

Table 3: Comparisons between software-only and hardware optimized operations.

Operation	Implementation				Performance increment
	Software only		Hardware accelerated		
	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	3.54249	319750003	3.1975	1.1x

the maximum performance of both designs. Because of the large amount of performed multiplications along the algorithm, it can be seen from the table that the total of available DSP blocks were used by the hardware accelerated implementation.

On the other hand, Table 3 reports the time and cycles taken to complete some specific operations in the software program and its hybrid architecture counterpart.

All the arithmetic and manipulation operations of the accelerated SOPC outperform their software-only version, not only in speed, but also in precision, thanks to the non standard data lengths that are used in the intermediate results. This cannot be done, with such efficiency, in a common program, due to the fixed data lengths that have to be used. For example, a series of 14 bits wide data have to be stored in 16 bits wide variables, and their multiplication in variables of 32 bits, unless a data resolution loss can be tolerated. On the other hand, the hardware solution can take registers of 14 bits, and store the products in 28 bits wide structures, both if they are send to general purpose memories or to registers inside the FPGA. If the fact that the input data of the system are 32 bits wide is considered, this characteristic gets more importance, as a 64 bit resolution in the software-only program is more difficult to use, and it would be a necessity from the first performed multiplication.

As it can be seen in Table 3, the only operation in the system that still needs to be optimized is the CFO iterative section. This is expected as both the hybrid implementation and the software-only one use the $\sin f$ and $\cos f$ functions from the Altera version of the *math.h* library, that calculates the sine and cosine functions of simple precision floating point inputs. To find a way to accelerate these sine and cosine calculations, used for the complex exponential operations, the use of look-up tables and a CORDIC generator [10], are being tested. These experiments consider the tradeoffs between precision (so the final estimate has enough accuracy), speed (for a solution that can equal the performance of the other coprocessors), and occupied FPGA area.

7. 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 the Data-dependent Superimposed Training algorithm for channel estimation. 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. The problem of the complex exponentials has yet to be solved. As a solution using common mathematical series is not suitable for FPGAs, a cordic generator and a hybrid approach using small look-up tables is being analyzed. In addition, 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.

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 the maximum frequency of the system directly. 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 it allows the storage of a significantly bigger amount of data. Furthermore, experimental results show that the difference of maximum frequencies 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.

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 make the calculation of trigonometric functions (sines and cosines) the only obstacle left in order to obtain a practical system for DDST.

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.

References

[1] M. Dong, L. Tong, and B. M. Sadler, "Optimal insertion of pilot symbols for transmissions over time-varying flat fading channels," *IEEE Transactions on Signal Processing*, vol. 52, no. 5, pp. 1403–1418, 2004.

[2] A. G. Orozco-Lugo, M. Mauricio Lara, and D. C. McLernon, "Channel stimation using implicit training," *IEEE Transactions on Signal Processing*, vol. 52, no. 1, pp. 240–254, 2004.

[3] E. Alameda-Hernandez, D. C. McLernon, M. Ghogho, A. G. Orozco-Lugo, and M. Mauricio Lara, "Improved synchronisation for superimposed training based channel estimation," in *Proceedings of the 13th IEEE/SP Workshop on Statistical Signal Processing Proceedings*, pp. 1324–1329, Bordeaux, France, July 2005.

[4] M. Ghogho, D. C. McLernon, E. Alameda-Hernandez, and A. Swami, "Channel estimation and symbol detection for block transmission using data-dependent superimposed training," *IEEE Signal Processing Letters*, vol. 12, no. 3, pp. 226–229, 2005.

[5] E. Alameda-Hernandez, D. C. McLernon, A. G. Orozco-Lugo, M. Mauricio Lara, and M. Ghogho, "Synchronisation and DC-offset estimation for channel estimation using data-dependent superimposed training," in *Proceedings of the European Signal Processing Conference (EUSIPCO '05)*, Istanbul, Turkey, September 2005.

[6] A. G. Orozco-Lugo, G. M. Galvan-Tejada, M. Mauricio Lara, and D. C. McLernon, "A low complexity iterative channel estimation and equalisation scheme for (data dependent) superimposed training," in *Proceedings of the European Signal Processing Conference (EUSIPCO '06)*, Florence, Italy, September 2006.

[7] M. Helfenstein and G. S. Moschytz, *Circuits and Systems for Wireless Communications*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.

[8] A. G. Orozco-Lugo, M. Mauricio Lara, E. Alameda-Hernandez, S. Moosvi, and D. C. McLernon, "Frequency offset estimation and compensation using superimposed training," in *Proceedings of the 4th International Conference on Electrical and Electronics Engineering (ICEEE '07)*, vol. 5, pp. 118–121, September 2007.

[9] K. Piromsopa, C. Aportewan, and P. Chongstitvatana, "An FPGA implementation of a fixed-point square root operation," in *Proceedings of the International Symposium on Communications and Information Technology*, vol. 14–16, pp. 587–589, Thailand, 2001.

[10] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *Proceedings of the 6th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '98)*, pp. 191–200, Monterey, Calif, USA, February 1998.