Hardware Implementation of Real-Time Order Statistic Filters on the TMS320C30 DSP

Aparna Kurupati

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

April 1994

ISBN 0-315-97691-8

ACKNOWLEDGEMENTS

I would like to express my profound gratitude to my supervisor Dr. M. N. S. Swamy, for providing the guidance and support that made this work possible. I would also like to thank the members of my examination committee, Dr. E. I. Plotkin, Dr. M. O. Ahmad, and Dr. T. Radhakrishnan, for their helpful comments and suggestions. Last, but certainly not the least, I wish to express my deep gratitude and heartfelt appreciation to Dr. N. Rama Murthy for his constant inspiration and support throughout this work.

Many thanks are due to my family and friends for their help, advice and encouragement.

This work was supported by grants awarded to Prof M. N. S. Swamy by MICRONET, a National Network of Centers of Excellence, and the Natural Sciences and Engineering Research Council of Canada.

ABSTRACT

Aparna Kurupati

Order Statistic Filters are a class of non-linear filters whose output is a linear combination of the order statistics of the input. Analysis of these filters shows that these filters have good edge preservation properties and are very suitable for the removal of impulsive noise. In this thesis, algorithms for the computation of the Running Order Statistics (ROS) are implemented in real-time using the TMS320C30 digital signal processor (DSP). A pratical application of the realized OSF for the demodulation of a double side band AM signal is also presented.

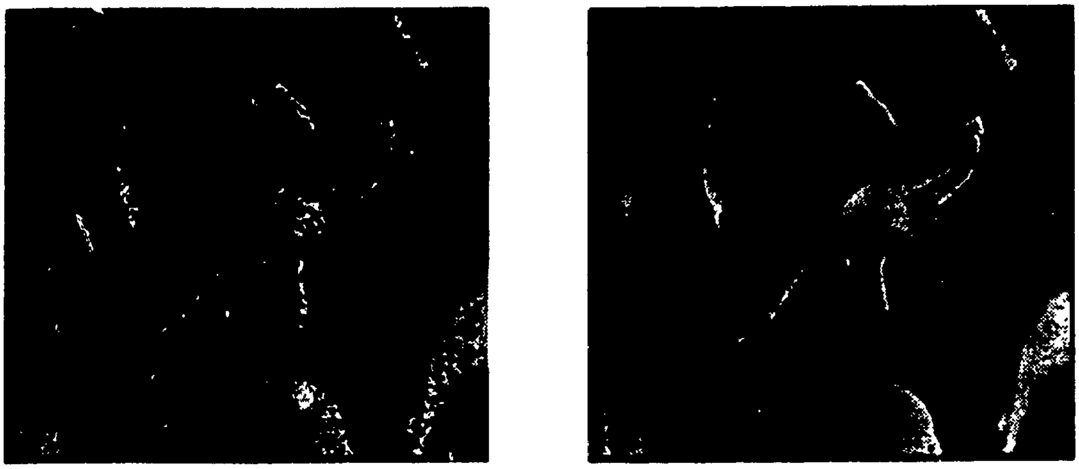
TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Linear filters have long been the primary tool for signal and image processing. They have the very useful superposition property. By use of superposition, a filter's response to an input consisting of a signal plus noise may be separated into an output signal plus noise. Thus a filter's response to signal and noise may be studied separately. Linear filters also have as Eigen-functions the sinusoids, sines and cosines. This means that an input sinusoid of frequency f_1 to a linear filter produces a sinusoid of frequency f_1 at its output. The effect of the filter is to possibly change the amplitude and phase of the sinusoid. This characteristic of linear filters gives rise to the transfer function representation of these filters. Unfortunately, linear filters have poor performance in the presence of noise that is not additive as well as in pr :blems where it is necessary to operate on signals with sharp edges. For example, consider the image shown in Figure 1.1(a). The image is corrupted by mixed impulsive noise.



(a) (b)

Figure 1.1: (a) Image Corrupted by Impulsive Noise. (b) Filtered Image.

Impulsive noise appears as black or white spots. This noise is usually caused by errors during image acquisition or transmission through communication channels. During this process some image pixels are destroyed and they take high positive (white spots) or low values. Filtering this image with a linear filter would smear out the sharp edges as well as the noise. Such smearing of the data is unacceptable in many applications. For such reasons nonlinear filtering techniques for signal and image processing were considered as early as 1958 [1]. One of the most popular family of nonlinear filters for noise removal is the Order Statistic Filter. The median filter which represents one of the simplest such filters was first suggested by Tukey [2] as a time series tool for robust noise suppression. Applying a 3x3 median filter to the noise corrupted image of Figure 1.1 (a) results in Figure 1.1(b). The impulses have been removed almost completely and the edges are preserved. Since then the median filter and its modifications have found numerous applications in digital image processing [3], [4], [5], in digital image analysis [6], in digital TV applications [7], in speech processing and coding [8], [9], in cepstral analysis and in various other applications.

In 1981, Gallagher and Wise [10], and Tyan [11] developed certain deterministic properties of the filter; in particular, they showed that certain signals (labelled root signal in [10]) are invariant to median filtering if they possess a minimum degree of smoothness (local monotoniticity), and that repeated application to any finite-length signal converges to a root in a finite number of passes.

Kuhlmann and Wise [12] derived some properties of the second moment of median filtered sequences of independent data. They derived the bivariate distribution function for median filtered sequences of independent arbitrary, second-order random variables.

Ataman et al. [13] showed that median filters can remove impulsive plus Gaussian white noise better than Hanning filters, when the amplitude of the impulses is large or the energy of the Gaussian noise is relatively low.

These results gave the median filter a theoretical groundwork, and spurred the development of a number of extensions and generalizations, that led into the consequent evolution of the Order Statistic Filter (OSF).

1.1 Generalization of the Median Filter

In this section, a literature survey of filters based on Order Statistics will be considered. Filters based on Order Statistics include L-filters, alpha-trimmed mean filters, max/median filters, median filters, rank-order filters etc. These filters have been designed to meet various criteria, e.g. robustness, adaptivity to noise probability distributions, preservation of edge information, preservation of image details [14].

Nodes and Gallagher [15] extended the median filter to the ith rank order filters. They showed that for nonmedian ith ranked-order operations, repeated application of the operation reduces any signal to a root. Also, they proved that the output of a recursive median filter is invariant to subsequent passes by the same filter.

In 1984, Fitch et al. [16] showed that median filtering an arbitrary level signal to its root is equivalent to decomposing the signal into binary signals, filtering each binary signal to a root with a binary median filter, and then reversing the decomposition.

Several other filters closely related to median filters were then introduced. These include the recursive median filters and the recursive separable median filters. Arce [17], McLoughlin and Arce [18] using threshold decomposition derived closed-form expressions for the statistics of recursive median filters and the root structure of the recursive separable median filter.

Arce and McLoughlin [19] also used this tool to analyze the behaviour of their max/median filters. They showed that the max/median filters are superior to conventional median filters since the roots of the max/median filters consist of very low resolution features which are suppressed by the latter filters.

It was shown by Bovik et al. [20] that an Optimal Order Statistic Filter (OOSF) whose output is a linear combination of the order statistics of the input sequence combines the properties of both the averaging and median filters. They also showed that the optimal (under the MSE criterion) order statistic filter tends toward the median filter as the noise becomes more impulsive. A theory of order statistics filters and their relationship to linear FIR filters was later investigated by Longbotham and Bovik [21].

Bednar and Watt [22] explained the relationship between alpha-trimmed mean filters and median filters, and provided a new explanation of the convergence of repeated median filtering of an arbitrary sequence to a root sequence.

Other generalizations of the median filter include the adaptive approach of median filtering. Lin and Wilson [23] proposed algorithms employing adaptive-length median filters to improve impulse noise rejection.

Nieminen et al. [24] presented median type filters with adaptive substructures suitable for filtering signals with rapidly varying characteristics. Using adaptive filter substructures, the current signal value from future signal values and from past input or output signal values is estimated.

A third article to explore the adaptive approach was by Haweel and Clarkson

[25]. The authors presented a class of adaptive algorithms employing order statistic filtering of the sampled gradient estimates. These algorithms, dubbed order statistic least mean squares, are designed to facilitate adaptive filter performance close to the least squares optimum across a wide range of input environments from Gaussian to highly impulsive.

Another major generalization of median filters was the class of stack filters developed by Wendt et al. [26]. Along with the limited superposition property of median and rank order filters, another property was necessary to define this new and larger class of non-linear filters. This is called the stacking property [26] which is an ordering property, also shared by all rank-order filters. The authors also investigated the convergence and syntactic behaviour of these filters to determine when they preserve monotone signals, edges, and median filter roots.

Coyle [27], and Lin and Coyle [28] showed how to design the optimal stack filter relative to a Markov signal/noise model and a modified mean-absolute estimation error.

Gabbouj et al. [29] developed a theory for the structural behaviour of stack filters. Their theory allowed the designer to pick a filter which minimizes noise subject to constraints on its structural behaviour.

During the same year, two more articles were published: the first by Lin et al. [30] dealt with adaptive stack filtering under the Mean Absolute Error Criterion.

The second paper by Wendt [31] dealt with nonrecursive and recursive stack filters, and their filtering behaviour. He showed that a recursive stack filter has the same roots as the corresponding nonrecursive stack filter.

These theories of threshold decomposition [16] along with the stacking properties [26] have given rise to efficient VLSI implementations for stack filters.

1.2 Hardware Implementations of Rank-Order Filters

Real-time implementation of various forms of non-linear filters requires the real-time implementation of OSFs. Real-time implementation of an OSF involves the computation of Running Order Statistics (ROS) of samples inside a window which get continuously updated with the arrival of a new sample(s). Calculation of ROS requires comparisons, additions, multiplications and nonlinear function evaluations, which result in tremondous amounts of computation. Thus, fast algorithms and structures are essential. In this section we review several fast algorithms and structures for the implementation of OSFs.

VLSI implementations for rank-order based filter have been done by several authors [26], [32]. Oflazer [33] gave the design and implementation of a VLSI chip using odd/even transposition sort-network capable of performing one-dimensional median filtering operation.

Huang et al. [34] presented a fast algorithm for two-dimensional median filtering based on sorting and updating the gray level histogram of the picture elements in the window.

Ataman et al. [35] developed an algorithm which determines the kth bit of the median by inspecting the k most significant bits of the samples. An alternative method of computing the median was given by Rao and Rao [36].

Chen [37] proved that a ranked order filter can be realized by using one binary processing circuit in k steps, if k is the number of the bits in the input signal samples. Such an implementation reduces the time area complexity to $O(k)$ from $O(2^k)$ required by the classical threshold decomposition.

Rama Murthy and Swamy [38] proposed algorithms to compute the Running Order Statistics (ROS) in real-time. The algorithms can be executed in either serial or parallel modes of execution. They also proposed a simple expansion algorithm to compute an $(r + 1)$ ROS of samples in a window using two r-bit OSFs. They also gave a parallel architecture with programmable window size and rank order for the implementation of OSFs and a VLSI architecture which permits the usage of two r-bit OSFs to implement an $(r + 1)$-bit OSF, where r is the resolution of the input signal samples. Further this paper shows how the VLSI chip incorporating the proposed architecture can be used as the basic building block in the real-time implementation of various forms of nonlinear filters.

1.3 Motivation and Objective

In view of the considerable interest in median and order statistic filters, the question of the implementation has itself received considerable attention, both in relation to fast algorithms and to dedicated hardware implementations. Implementation of these filters involves the incorporation of an ordering transformation which creates a significant computational overhead.

For this reason, it is of utmost importance that fast algorithms on dedicated hardware are used. VLSI implementations are ideal for such situations since they can be customized and hence for real-time applications have fast processing times. They also have certain disadvantages, i.e, they are not cost effective and take a much longer time to implement. In order to circumvent the above problems, general purpose DSP boards can be used.

In [38], algorithms to compute the ROS in real-time which are based on sequential and parallel mode of execution [38] are proposed. In addition a simple expansion algorithm is used to compute the $(r+1)$-bit ROS of samples in a window using two r-bit OSFs.

In this thesis we use the above algorithms to implement OSFs on a general purpose digital signal processor (DSP) TMS320C30 from Texas Instruments. A practical application of the realized OSF for the demodulation of a double sideband amplitude modulated (AM) signal is also considered.

1.4 Organization of the Thesis

The organization of the thesis is as follows. In Chapter 2, we define the order statistic filter and consider a few important properties. Some modifications of OSFs are then considered. Next, a brief summary of the algorithms implemented is presented with the flow diagrams and C code.

In Chapter 3, we give a description of the block diagram of the implementation along with the flow diagrams and TMS320C30 C code for each of the algorithms considered. In Chapter 4, we consider a few applications along with their solutions of order statistic filters. We also consider one such application for our implementation of the OSF on the TMS320C30 processor. Finally, in Chapter 5, our conclusions are presented.

Chapter 2

Order Statistic Filters

In the following sections we begin with an introduction to Order Statistic Filters. A few properties of Ranked Order Filters are considered, followed by some modifications of OSFs. The algorithms proposed in [38] for the computation of the ROS are discussed in detail.

2.1 Introduction

The OSF of the ith order is a digital filter consisting of a window, (usually containing an odd number of samples) which is stepped across an input signal. At each step the points inside the window are ranked according to their values, and the output at any time instant is the ith order statistic of the number of samples in the window at its input at the same instant.

Let the window size be N and $x_p(n), p = 1, 2, ..., N$ be the samples inside the window $w_N(n)$, at the n th time instant, i.e.,

$$w_N(n) = \{x_1(n), x_2(n), ..., x_N(n)\} \tag{2.1}$$

Define a new window $w'_N(n)$ as the one obtained from $w_N(n)$ by sorting the N samples inside $w_N(n)$ in the increasing order of algebraic value, i.e.,

$$w'_N(n) = \{x_{(1)}(n), x_{(2)}(n), ..., x_{(N)}(n)\} \tag{2.2}$$

where $x_{(1)}(n) \leq x_{(2)}(n), \leq ... \leq x_{(N)}(n)$. The output of the OSF with input $w_N(n)$ is given as

$$\gamma_{OSF}(n) = x_{(i)}(n), \;\; where \;\; t = (N + 1 - i) \tag{2.3}$$

Suppose, for instance, that $N = 5$ and that the samples, in time order (nth instant), in the window are

$$w_5(n) = \{x_1(n), x_2(n), x_3(n), x_4(n), x_5(n)\} = \{8, 1, 6, 4, 1\}$$

In rank order they would be

$$w'_N(n) = \{x_{(1)}(n), x_{(2)}(n), x_{(3)}(n), x_{(4)}(n), x_{(5)}(n)\} = \{1, 1, 4, 6, 8\}$$

The 2nd largest value is given by x_4, which for the above example would be 6. The window $w_N(n)$ is updated at every sampling instant n with the deletion of the oldest sample (in time) and the inclusion of the latest sample (current or new sample).

Figure 2.1 shows the input and output for a window width of 5 order statistic filter, and also shows how the window moves along the signal. This operation still

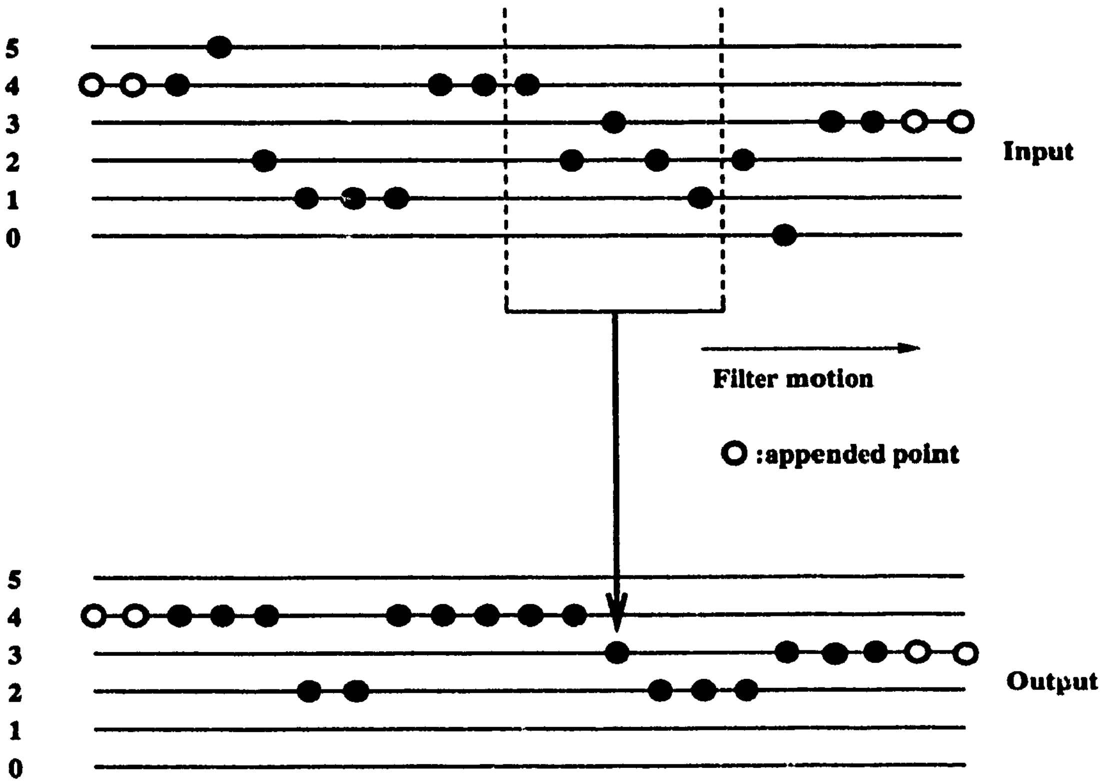


Figure 2.1: An Order Statistic Filter

eliminates some impluses in both directions, it also tends to favor larger valued or rising points.

The output values near the beginning or end of a finite length sequence may not be defined since some samples will be missing from the window. To prevent this, the endpoints of the sequence are repeated a sufficient number of times to fill up the window even when it is centered on an endpoint. If the window has width N, then the endpoints must be repeated $(N-1)/2$ times. In Figure 2.1 each endpoint is repeated twice since the window has width 5.

2.2 Some Properties of Order Statistic Filters

In this section we consider some theoretical results on the behaviour of the median and order statistic filters.

2.2.1 *i*th Ranked Order Operations

The ith ranked-order operation can also be defined by the decision rule used to select the output values at each step. For N points inside the window, the ith ranked point $x_i(n)$ is the point such that there are at least i points with values less than or equal to $x_i(n)$, and at least $N-(i-1) = N+1-i$ points with values greater than or equal to $x_i(n)$ [15]. A number of properties of the ith ranked-order operations are now defined but before we proceed we define a few important concepts.

A constant neighborhood is a region of at least $\frac{N+1}{2}$ consecutive points, all of which are identically valued.

An edge is a monotonically rising or falling set of points surrounded **on both sides** by constant neighborhoods.

An impulse is a set of $\frac{N-1}{2}$ or fewer points whose values are different from the surrounding regions and whose surrounding regions are identically valued constant neighborhoods.

A root is a signal that is not modified by filtering.

Properties

Property 1: A point $x_p(n)$ is unchanged ($\gamma_{OSF}(n) = x_p(n)$) by an ith ranked-order operation if two conditions are met. The point $x_p(n)$ is located in a constant region, and $x_p(n)$'s position is restricted to $b + \frac{N-1}{2} - a \leq n \leq c - |\frac{N+1}{2} - i| + a$ where a is any nonnegative integer of value less than $\frac{N+1}{2} - |\frac{N+1}{2} - i|$, and b and c are the positions of the two endpoints of the constant region.

Property 2: ith ranked-order filters can be used for the elimination of positive impulses of width less than $N + 1 - i$ points or negative impulses of width less than i samples.

Property 3: Upon each pass of an ith ranked-order operation, every edge of a signal will be moved to the left(advanced) by $sgn[edge] \cdot (i - \frac{N-1}{2} - 1)$points, where

$$sgn[edge] = \begin{cases} +1 & \text{if } x_p(n) \leq x_p(n+1) \\ -1 & \text{if } x_p(n) \geq x_p(n+1) \end{cases}$$

for n ranging over all positions of the edge.

Property 4: Any constant region of $N+1-i$ or more points surrounded by constant neighborhoods of lesser values will be changed in width by $2 \cdot (i - \frac{N-1}{2} - 1)$ points after being passed through an ith ranked-order operator.

Property 5: Only constant signals are invariant to ith ranked-order operations if i is not equal to $\frac{N+1}{2}$.

Property 6: If i is not equal to $\frac{N+1}{2}$, then repeated passes of an ith ranked-order process will reduce any finite length signal to a constant.

A signal may be formed from independent identically distributed (i.i.d.) sample points of a random process. Such a signal would be formed if white noise were sampled to form the input signal. For this type of signal, theoretical results from order statistics [39] may be used to obtain the first-order distribution $F_\gamma(\cdot)$ and the density $f_\gamma(\cdot)$ of the output of an ith ranked-order operation. If the distribution $F_x(\cdot)$ and the density $f_x(\cdot)$ of the input are known, then $f_\gamma(\cdot)$ and $F_\gamma(\cdot)$ are given by the following.

Property 7:

$$f_\gamma(x) = \frac{(N)!}{(i-1)!1!(N-i)!} \cdot [F_x^{i-1}(x)(1-F_x(x))^{(N-i)} f_x(x)]$$

Property 8:

$$F_\gamma(x) = \sum_{K=i}^{N} \frac{(N)!}{K!(N-K)!} F_x^K(x)(1-F_x(x))^{N-K}$$

where N is the window size.

Property 9: A median filter $(i = \frac{N+1}{2})$, $x_p(\cdot) \rightarrow \gamma_{OSF}(\cdot)$, with an input of i.i.d sample points will transform the distribution of the input $F_x(\cdot) \rightarrow F_\gamma(\cdot)$ symmetrically about 0.5. That is, for any l such that $F_x(l) \rightarrow F_\gamma(l)$, then $(1 - F_x(l)) \rightarrow (1 - F_\gamma(l))$.

Property 10: The statistical median of a signal of i.i.d sample points is preserved upon median filtering $(i = \frac{N+1}{2})$, or given l such that $F_x(l) = 0.5$, then $F_\gamma(l) = 0.5$. The proofs of the above properties are given in [15].

2.2.2 Recursive Operations

If at every step, the leftmost $\frac{N-1}{2}$ points are replaced in the moving window with the previous $\frac{N-1}{2}$ output points, and we apply the same decision rule as previously given then we obtain the ith ranked-order operation to obtain the next output value.

Property 11: A signal is invariant to recursive filtering if and only if it is invariant to standard filtering.

Property 12: Any signal will be reduced to a root after one pass of a recursive median filter $(i = \frac{N+1}{2})$.

Property 13: If $i \neq \frac{N+1}{2}$, then the last computed output value of a signal being operated on by a recursive ith ranked-order operation is the value of the signal root for that operator. For $i > \frac{N+1}{2}$ $(i < \frac{N+1}{2})$ this value is the value of the maximum(minimum) value to survive the first filter pass.

Again, the proofs of the above properties are given in [15].

2.3 Some Modifications of Order Statistic Filters

In this section some of the filters based on Order Statistics are considered.

2.3.1 Optimal Order Statistic Filters

These filters combine the properties of both the averaging and the median filters. The output is a linear combination of the outputs of the ith OSFs ($i = 1, 2, ..., N$). If $\gamma_{OOSF}(n)$ is the output of an Optimal Order Statistic Filter (OOSF) operating on $w_N(n)$, then

$$\gamma_{OSF}(n) = \sum_{i=1}^{N} \alpha_i x_{(i)}(n)$$

where $x_{(i)}(n)$ is the output of the ith OSF at time instant n and α_i's ($i = 1, 2, ..., N$) are constants that are chosen for a particular application.

2.3.2 α-Trimmed Mean Filters

Median filters discard impulses and preserve edges. However, in the suppression of additive white Gaussian noise, its performance is inferior to that of the moving average. Thus, a good compromise between the median and moving average is the α-Trimmed Mean Filter (ATMF).

The output of the ATMF, $\gamma_{ATMF}(n)$ operating on $w_N(n)$ is obtained by initially deleting certain samples in $w_N(n)$ and then averaging the remaining samples.

The removed samples are the most extreme values (both high and low), with an equal number of samples dropped at each end (symmetric trimming). The number of samples deleted is controlled by the triming parameter α which assumes values between 0 and 0.5.

$$\gamma_{OSF} = \frac{1}{N - 2[\alpha N]} \sum_{i=[\alpha N]+1}^{N-[\alpha N]} x_{(i)}(n)$$

where $x_{(i)}(n)$ represents the ith order OSF output.

2.3.3 Recursive Order Statistic Filter

If $w_N(n)$ at every sampling instant contains $[\frac{N}{2}]$ number of previously computed order statistic output samples ($[\frac{N}{2}]$ is the integer part of $\frac{N}{2}$), then such a filter is referred to as the Recursive Order Statistic Filter (ROSF) [40].

2.3.4 Adaptive Median Filter (AMF)

If N is varied at every sampling instant (i.e., the size of the window is varied) in accordance with signal activity inside the window as defined in [41], then such a filter is referred to as an Adaptive Median Filter (AMF).

2.4 Algorithms for OSF Implementation

In this section we consider the algorithms [38] used for the computation of the ROS. These algorithms are based on sequential and parallel modes of execution. We also consider an expansion algorithm which utilizes two r-bit OSFs to compute an $r+1$-bit OSF.

Before we proceed, for the computation of the ROS. we assume that $w_N(n)$ is as defined in equation 2.1, i is the desired order statistic and $t = N+1-i$. We also assume that the largest possible sample of $w_N(n)$ at any instant has an r-bit representation (resolution) and $L = 2^r$. Without loss of generality, we assume that $x_p(n) \geq 0, p = 1, 2, ..., N$.

2.4.1 Sequential Mode of Execution

This technique consists of two steps. Step A involves the formation of an m-array of length $L = 2^r$ where r represents the number of bits required to represent the larget possible sample of $w_N(n)$. Step B involves the computation of the ROS from the m-array elements.

Step A: *Formation of an m-array*

Form an array referred to as m-array with elements $m_1, m_2, ..., m_L$ by scanning each sample of $w_N(n)$ once. The kth element m_k contains number of samples in $w_N(n)$ greater than or equal to (k-1).

Step B: *Computation of ROS from m-array elements*

Let $a_1, a_2, ..., a_r$ be the r-bit binary representation a number z where a_1 is the MSB and a_r is the LSB. Computation of median $\gamma_{OSF}(n)$ consists of the following sub steps B1-B6:

```
Step B1: Initialize $z = 0$ (i.e. set $a_i$ for $1 \leq i \leq r$)
Step B2: Initialize a counter $k = 1$
Step B3: Set $a_k = 1$
Step B4: If $m_{z+1} < t$, then set $a_k = 0$
Step B5: $k = k + 1$
Step B6: Go to step B3 if $k \leq r$
```

At the end of the above procedure $\gamma_{OSF}(n) = z$

Let us now consider an example which illustrates the above algorithm.

Example 1: Let $w_N(n) = \{0, 1, 6, 3, 7, 3, 2, 4, 1\}$ where $N = 9$. Assume $r = 3$ so that $L = 2^r = 2^3 = 8$ and the maximum possible value for any sample in the window at any instant is 7. Steps A and B of the algorithm are given below:

Step A: *Formation of m-array (initally* $m1, ..., m8 = 0$*)*

Scanned element	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8
$x_1(n) = 0$	1	0	0	0	0	0	0	0
$x_2(n) = 1$	2	1	0	0	0	0	0	0
$x_3(n) = 6$	3	2	1	1	1	1	1	0
$x_4(n) = 3$	4	3	2	2	1	1	1	0
$x_5(n) = 7$	5	4	3	3	2	2	2	1
$x_6(n) = 3$	6	5	4	4	2	2	2	1

$x_7(n) = 2$	7	6	5	4	2	2	2	1
$x_8(n) = 4$	8	7	6	5	3	2	2	1
$x_9(n) = 1$	9	8	6	5	3	2	2	1

Step B: *Computation of median(initially* $z = \{a_1.a_2, a_3\} = \{0,0,0\} = 0$*)*

$K = 1 : a_1 = 1 \quad z = \{1,0,0\} = 4, \quad m_5 = 3, \quad 3 < 5, \quad a_1 = 0$

$K = 2 : a_2 = 1 \quad z = \{0,1,0\} = 2, \quad m_3 = 6, \quad 6 \nless 5, \quad a_2 = 1$

$K = 3 : a_3 = 1 \quad z = \{0,1,1\} = 3, \quad m_4 = 5, \quad 5 \nless 5, \quad a_3 = 1$

$$\gamma_{OSF} = z = \{0,1,1\} = 3$$

For the computation of the running median, everytime the window $w_N(n)$ gets updated by replacing an old sample p by a new sample q, the elements of the m-array are updated as part of Step A depending on the values of p and q as follows:

- If $p < q$, each m_i for which $p + 2 \leq i \leq q + 1$ is incremented by unity
- If $p > q$, each m_i for which $q + 2 \leq i \leq p + 1$ is decremented by unity
- If $p = q$, none of m_i, $i = 1, 2, ..., L$ is modified

Having updated the m-array, the new median is computed by following the procedure in Step B.

The ROS algorithm given above computes $\gamma_{OSF}(n)$ by sequentially computing $a_1, a_2, ..., a_r$ and hence, we refer to this mode of execution of the algorithm as the sequential mode. A flow diagram for the implementation of the above algorithm is given in Figure 2.2. In this flow diagram, the samples in the window $w_N(n)$ at any instant of time are stored in a N location circular buffer $w_N(j), j = 0, 1, ..., N-1$ and the buffer is so configured that a new sample q is written into the same memory location where the oldest sample p that it replaces was originally residing. The addressing of the locations of circular buffer is done by the pointer j which is incremented modulo N at the end of every ROS computation cycle (i.e., prior to the reading of a new input sample q).

A C language code implementing the sequence of operation in the flow diagram given in Figure 2.2 is shown in Figure 2.3 .

2.4.2 Parallel Mode of Execution for the Computation Of Running Median

For high speed implementation of the algorithm presented in Section 2.4.1, parallel processing is necessary. This is possible with the following modification.

Define L logical variables M_i, $i = 1, 2, ..., L$ such that

$$M_i = \begin{cases} logic\ '0' & \text{if } m_i < t \\ logic\ '1' & \text{if } m_i \geq t \end{cases}$$

where $t = N + 1 - i$ and i is the order of interest. With this definition, it is easy to see that r bits of required ROS can be directly obtained from the following relations

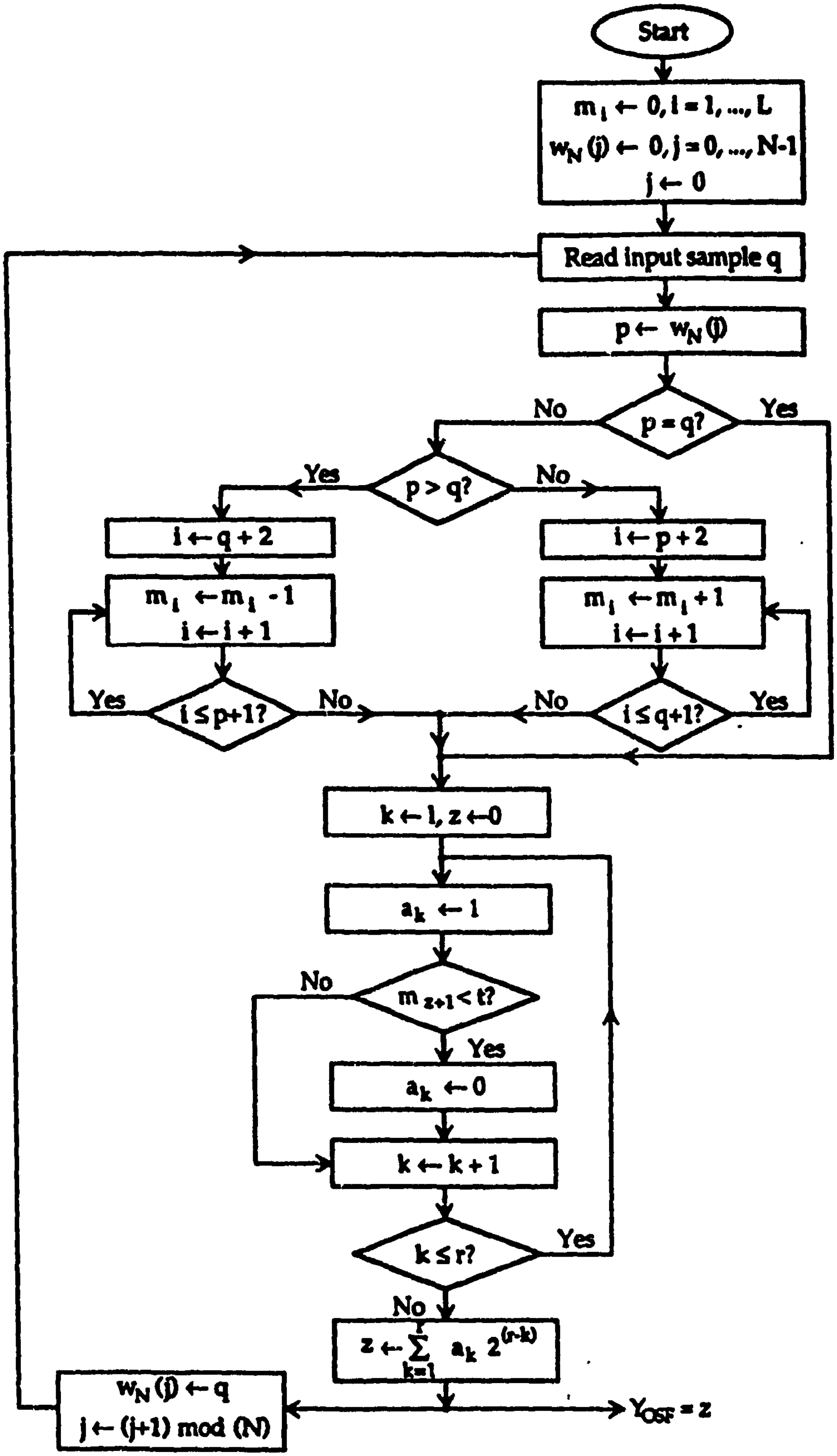


Figure 2.2: Flow diagram for the sequential mode of execution of the ROS algorithm

```
/* C language program for the implementation of the OSF */
/* This program executes ROS algorithm in the sequential mode */

#include <stdio.h>

/* Definitions of OSFs parameters */
#define   r    8    /* r is the resolution of the input signal samples */
#define   N   21    /* N is the window size  */
#define   t    2    /* order of the OSF = 20 so that t = N+1-i */

int     w[N], m[1<<r],  L, a[r];
main()
{
      int p, k, q, z, unity = 1, j = 0;

      /* Initialization of elements of m-array and the window */

      for(k = 0; k < N; k++) w[k] = 0;
      L =  (1 << r) ;
      for(k = 0; k < L; k++) m[k]=0;

      /* Begin computation of ROS */

      while(1){
           scanf("%d", &q); /* Read the new input sample q */
           p = w[j]; /* p is the oldest sample being replaced by q */

           /* compare p and q and update m-array */

           if( p < q){
                 for(k = (p+2); k <= (q+1); k++)
                        m[k-1] += unity;
           }
           if( p > q){
                 for(k = (q+2); k <= (p+1); k++)
                        m[k-1] -= unity;
           }

           /* Compute the i th Order Statistic from m-array elements */

           z = 0;  /* Initialize  z */
           for(k = 0; k < r; k++) {
                  a[k] = 1;
                  z = z + (a[k] << (r-k-1));
                  z = m[z] < t ? z-(a[k] << (r-k-1)) : z;
           }
           printf("%d\n", z);
           w[j] = q; /* replace sample p by q */
           j = (++j) > (N-1) ? 0 : j;
      }

  }
```

Figure 2.3: C program for the execution of ROS algorithm in sequential mode

$(2 \leq k \leq r)$:

$$a_1 = M_{L/2+1}$$

$$a_k = \sum_{d=0}^{2^{k-1}-1} B_d(a_1, a_2, ..., a_{k-1}) \cdot M_{d,k}(a_1, a_2, ..., a_{k-1}. a_k, a_{k+1}, ..., a_r) \qquad (2.4)$$

where $\cdot$ denotes logical AND function, the Boolean function $B_d(a_1, a_2, ..., a_{k-1})$ corresponds to d th $MinTerm$ of (k-1) variables $a_1, a_2, ..., a_{k-1}$, the Boolean function

$$M_{d,k}(a_1, a_2, ..., a_{k-1}, a_k. a_{k+1}, ..., a_r)$$

is equal to logical output M_{1+D}, and D is the decimal equivalent of an r bit binary number $b_1, b_2, ..., b_r$ (where b_1 is the MSB) obtained with $b_1, b_2, ..., b_{k-1}$ corresponding to (k-1) bit binary representation of number d, $b_k = 1$, and $b_{k+1} = b_{k+2} = ... = b_r = 0$. Hence from equation 2.4 it can be seen that all the r bits of required ROS can be expressed directly in terms of the logical variables $M_1, M_2, ..., M_L$. We now consider an example with $r = 3$ and $L = 8$.

$$a_1 = M_5$$

$$a_2 = B_0(a_1) \cdot M_{0,2}(a_1, a_2, a_3) + B_1(a_1) \cdot M_{1,2}(a_1, a_2, a_3)$$

$$= \overline{a_1} \cdot M_3 + a_1 \cdot M_7 = \overline{M_5} \cdot M_3 + M_5 \cdot M_7$$

$$a_3 = B_0(a_1, a_2) \cdot M_{0,3}(a_1, a_2, a_3) + B_1(a_1, a_2) \cdot M_{1,3}(a_1, a_2, a_3) +$$

$$B_2(a_1, a_2) \cdot M_{2,3}(a_1, a_2, a_3) + B_3(a_1, a_2) \cdot M_{3,3}(a_1, a_2, a_3)$$

$$= \overline{a_1} \cdot \overline{a_2} \cdot M_2 + \overline{a_1} \cdot a_2 \cdot M_4 + a_1 \cdot \overline{a_2} \cdot M_6 + a_1 \cdot a_2 \cdot M_8$$

$$= \overline{M_5} \cdot \overline{M_3} \cdot M_2 + \overline{M_5} \cdot M_3 \cdot M_4 + M_5 \cdot \overline{M_7} \cdot M_6 + M_5 \cdot M_7 \cdot M_8$$

The flow diagram for implementing the sequence of operations required for the parallel mode of execution of the algorithm is give in Figure 2.4. In this flow diagram, the samples in the window $w_N(n)$ at any instant of time are stored in a N location circular buffer $w_N(j), j = 0, 1, ..., N-1$ and the buffer is so configured that a new sample q is written into the same memory location where the oldest sample p that it replaces was originally residing. The addressing of the locations of circular buffer is done by the pointer j which is incremented modulo N at the end of every ROS computation cycle (i.e., prior to the reading of a new input sample q).

The corresponding C language program is given in Figure 2.5 .

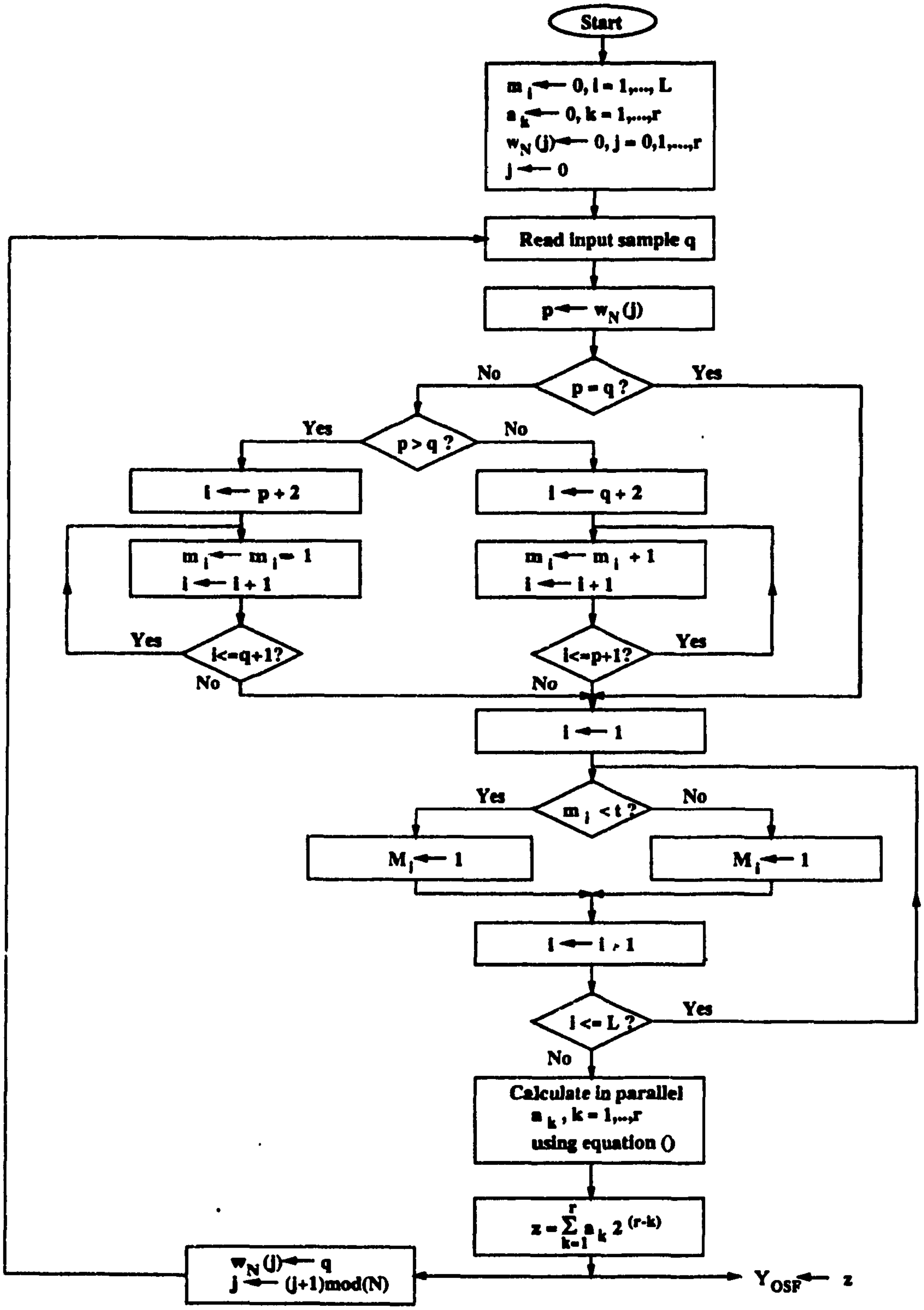


Figure 2.4: Flow diagram for the parallel mode of execution of the ROS algorithm

```
/* C code for the implementation of the OSF */
/* This program executes ROS algorithm in the parallel mode */

#include <stdio.h>

/* Definition of OSF parameters */

#define   r              8   /* r is the resolution of the input samples */
#define   N             10   /* N is the size of the window */
#define   t              2   /* order of the OSF = 9, t = N+1-i */

int     w[N], m[1<<r],  L, a[r], M[1<<r], N5;
int     Q[129][r], b[r], d, n, l, z1, D, sum, sum1, q1, q2, temp[r];

main()
{
        int p, q, z, k, j = 0, inc = 1;

        /* Initialization of the elements of m-array and window */

        for(k = 0; k < N; k++) w[k] = 0;
        L = 1 << r;
        for(k = 0; k < L; k++) m[k]=0;

        /* Begin computation of ROS */

        while(1){
                scanf("%d", &q);  /* Read the new input sample q */
                p = w[j];         /* Read p, the oldest sample */

               /* compare p and q and update m-array */

               if( p < q){
                       for(k = (p+2); k <= (q+1); k++)
                               m[k-1] += inc;
               }
               if( p > q){
                      for(k = (q+2); k <= (p+1); k++)
                               m[k-1] -= inc;
               }

               /* update M-array based on m-array */

               for(k = 0; k < L; k++)
                      M[k] = m[k] < t ? 0 : 1;

               z = 0;
               a[0] = M[L/2];

               for(k = 2; k <= r; k++) {
                      sum = 0;
                      N5 = 1 << (k-1);
                      for(d = 0; d < N5; d++){
                             q1 = d;
                             sum1 = 1;
                             D = 0;
```

...Cont'd

```
             /* Calculation of B[d] */
             for(l = k-1; l >= 1; l--) {
                 if(q1 == 0)
                   sum1 = sum1*(!a[l-1]);

                 else {
                   temp[l-1] = q1 % 2;
                   sum1=temp[l-1]==0 ? sum1*(!a[l-1]):sum1*a[l-1];
                   q1 /= 2;
                 }
            }

       /* Calculation of Q[d][k] */

            b[k-1] = 1;
            n = k - 1;
            q2 = d;
            for(l = n; l >= 1; l--){
                 if(q2 == 0) b[l-1] = q2 % 2;
                 else{
                   b[l-1] = q2 % 2;
                   q2 /= 2;
                 }
            }
            for( l = 1; l <= r;l++){
                 z1 =  1 << (r-1);
                 D += b[l-1]*z1;
            }
            Q[d][k] = M[D];
            sum = sum || sum1*Q[d][k];
            for(l = 0; l < r; l++)
                 b[l] = 0;
        }
       ; a[k-1] = sum;
    }

    /* Compute the i th Order Statistic */
    for(k = 0; k < r; k++)
       z += a[k] << (r-k-1);

    printf("%d \n", z);
    w[j] = q;     /* replace sample p by q */
    j= (++j) > (N-1) ? 0 : j;
  }

}
```

Figure 2.5: C program for the execution of ROS algorithm in parallel mode

2.4.3 Expansion Algorithm

To compute the (r+1)-bit ROS of samples in a window, a simple expansion algorithm can be used by using two r-bit OSFs (say OSF_1 and OSF_2) to perform the necessary computation.

The algorithm consists of the following three steps A, B, C.

Step A: Scan and classify each one of the signal samples $x(\cdot)$ of $w_N(n)$ as belonging to either one of the two windows $w_N^{(1)}(n)$ or $w_N^{(2)}(n)$ according to the following rule:

1. If the sample $x(\cdot)$ is less than 2^r, it is inserted as a member of $w_N^{(1)}(n)$ and a zero valued sample which we refer to as a non-set zero is inserted as a member of $w_N^{(2)}(n)$. A counter $NSZ2$ which keeps track of the non-set zeros inserted into $w_N^{(2)}(n)$ is incremented by unity.

2. If the sample $x(\cdot)$ is greater or equal to 2^r, then $(x(\cdot) - 2^r)$ is inserted as a member of $w_N^{(2)}(n)$ and a zero valued sample (non-set zero as mentioned earlier) is inserted as a member of $w_N^{(1)}(n)$. A counter $NSZ1$ which keeps track of non-set zeros inserted into $w_N^{(1)}(n)$ is incremented by unity.

The idea behind the above classification is to form two windows $w_N^{(1)}(n)$ and $w_N^{(2)}(n)$, each having signal samples of maximum possible value $(2^r$-1),

Step B: Let z denote the required i th ROS of $w_N(n)$. Having classified all the samples of $w_N(n)$ as belonging to either $w_N^{(1)}(n)$ or $w_N^{(2)}(n)$, evaluate z_1, the i_1 th

ROS of samples of $w_N^{(1)}(n)$ and $z2$ the i_2 th ROS of samples of $w_N^{(2)}(n)$ where $i_1 = N+1-t_1$, $i_2 = N+1-t_2$ ε nd t_1 and t_2 are obtained as per the following logic:

```
$t_1 = t - NSZ1$
$SEL = t_1$
$if(t_1 < 1) t_1 = t_1 + N$
$end$
$t_2 = t - NSZ2$
$if(t_2 < 1) t_2 = t_2 + N$
$end$
```

Step C: Obtain z using the following logic:

```
$if(SEL \leq 0)$
$z = z_2 + 2^r$
$elseif(SEL > 0)$
$z = z_1$
```

Example 2: Let $r = 3$, $N = 9$, $w_N(n) = \{0, 1, 15, 11, 3, 2, 1, 8, 9\}$, and $i = 7$ so that $t = N + 1 - i = 3$

Step A: Formation of $w_N^{(1)}(n)$ and $w_N^{(2)}(n)$:

Scanned element of $w_N(n)$	$w_N^{(1)}(n)$	$NSZ2$	$w_N^{(2)}(n)$	$NSZ1$
0	$\{0, \ldots\ldots\ldots\}$	1	$\{\emptyset, \ldots\ldots\ldots\}$	0
1	$\{0,1, \ldots\ldots\ldots\}$	2	$\{\emptyset,\emptyset, \ldots\ldots\ldots\}$	0
15	$\{0,1,\emptyset, \ldots\ldots\ldots\}$	2	$\{\emptyset,\emptyset,7, \ldots\ldots\ldots\}$	1
.	.	.	.	.
.	.	.	.	.
9	$\{0,1,\emptyset,\emptyset,3,2,1,\emptyset,\emptyset\}$	5	$\{\emptyset,\emptyset,7,3,\emptyset,\emptyset,\emptyset,0,1\}$	4

(Note $\emptyset$ indicates a non-set zero)

Step B: Computation of t_1 and t_2:

$$t_1 = 3 - 4 = -1; SEL = -1$$

$$t_1 = -1 + 9 = 8$$

$$t_2 = 8 - 5 = 3$$

Step C: $z_1 = 0$ (2nd ROS of $w_N^{(1)}(n)$)

$z_2 = 1$ (7th ROS of $w_N^{(2)}(n)$)

and $z = 1 + 2^3 = 9$ (since $SEL < 0$) which is the same as 7th ROS of $w_N(n)$.

It is easy to see that at any stage during the ROS computation, the sum of the contents of $NSZ1$ and $NSZ2$ is equal to N. Hence, when $NSZ1$ is incremented by unity, $NSZ2$ has to be decremented by unity and vice-versa. Further, since contents of either $NSZ1$ or $NSZ2$ cannot exceed N or be lower than zero, these counters must be prevented from incrementing beyong N or decrementing below zero. The logic governing the update of $NSZ1$ and $NSZ2$ is given by the following programming loops (written in PRO-MATLAB language):

```
$if(p < 2^r)$
        $if(q \geq 2^r)$
$NSZ1 = NSZ1 + 1$
        $if(NSZ1 \geq N)$
        $NSZ1 = N$
$end$
$NSZ2 = NSZ2 + 1$
        $if(NSZ2 \leq 0)$
        $NSZ2 = 0$
$end$
$end$
$end$
```

```
$if(p \geq 2^r)$
        $if(q > 2^r)$
$NSZ2 = NSZ2 + 1$
        $if(NSZ2 \geq N)$
        $NSZ2 = N$
$end$
$NSZ1 = NSZ1 + 1$
        $if(NSZ1 \leq 0)$
        $NSZ1 = 0$
$end$
$end$
$end$
```

If $p = q$, neither $NSZ1$ nor $NSZ2$ is updated.

The ROS algorithm in the above form computes an $r + 1$ bit OSF using two r-bit OSFs and hence, we refer to this algorithm as the expansion algorithm. A flow diagram for the implementation of the above algorithm is given in Figure 2.6. In this flow diagram, the samples in the window $w_N(n)$ at any instant of time are stored in a N location circular buffer $w_N(j), j = 0, 1, ..., N - 1$ and the buffer is so configured that a new sample q is written into the same memory location where the oldest sample p that it replaces was originally residing. Based on the value of q counters $NSZ1, NSZ2$ and windows $w1_N(n), w2_N(n)$ are formed. The addressing of the locations of circular buffer is done by the pointer j which is incremented modulo N at the end of every ROS computation cycle (i.e., prior to the reading of a new input sample q).

A C language code implementing the above algorithm is given in Figure 2.7.

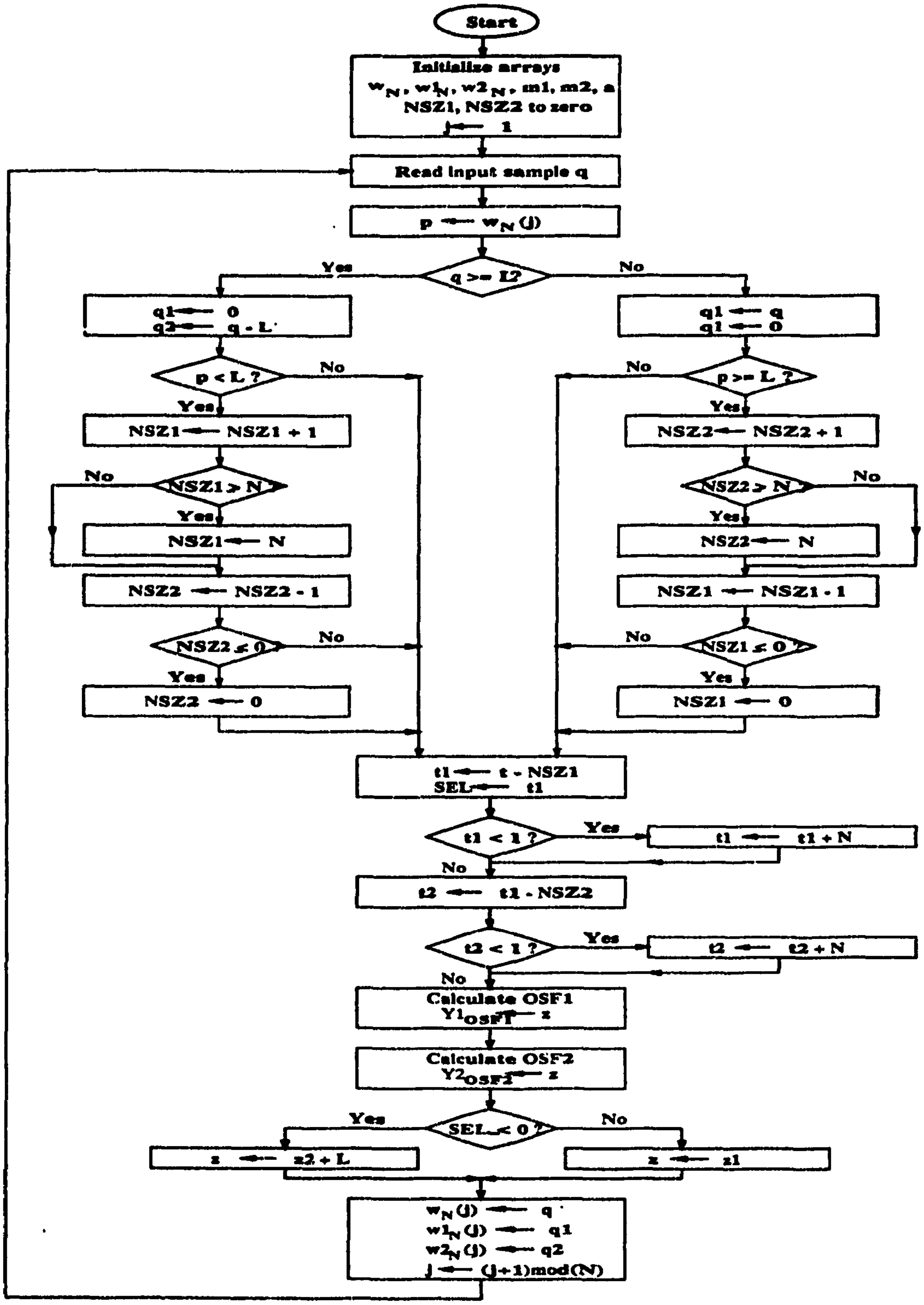


Figure 2.6: Flow diagram of the Expansion algorithm for the computation of the ROS

```
/* C language program for the implementation of the OSF  */
/* This program executes ROS algorithm using two OSFs (Expansion algorithm) */

#include <stdio.h>

/* Definition of OSFs parameters */

#define  r              8    /* r is the resolution of the input samples */
#define  N              8    /* N is the  size of the window */
#define  t              2    /* order of the OSF = 7 so that t=N+1-i  */

int     w[N], w1[N], w2[N], m1[1<<r], m2[1<<r], L, a[r];

main()
{
        int q, q1, q2, p, p1, p2, k, z, z1, z2, inc = 1,j = 0,
            NSZ1 = 0, NSZ2 = N, t1, t2, SEL;

        /* Initialization of elements of m1-array, m2-array and the windows */

        for(k = 0; k < N; k++) w[k] = 0;
        for(k = 0; k < N; k++) w1[k] = 0;
        for(k = 0; k < N; k++) w2[k] = 0;
        L = 1 << r;
        for(k = 0; k < L; k++) m1[k]=0;
        for(k = 0; k < L; k++) m2[k]=0;

        /* Begin Computation of the ROS */

        while(1){
            scanf("%d", &q); /* Read the new input sample q */
            p = w[j];        /* Read p, the oldest sample */

            /* Update  q1, q2 based on the value of q and update NSZ1 and
               NSZ2 based on the value of p */

            if((q >= L)) {
                   q1 = 0;
                   q2 = q - L;
                   if(p < L) {
                          NSZ1 += 1;
                          if(NSZ1 >= N) NSZ1 = N;
                          NSZ2 = NSZ2 - 1;
                          if(NSZ2 <= 0) NSZ2 = 0;
                   }
            }
            if((q < L)) {
                   q1 = q;
                   q2 = 0;
                   if(p >= L){
                          NSZ2 += 1;
                          if(NSZ2 >= N) NSZ2 = N;
                          NSZ1 = NSZ1 - 1;
                          if(NSZ1 <= 0) NSZ1 = 0;
                   }
            }
```

...Cont'd

```
/* Computation of t1 and t2 */

t1 = t - NSZ1 ;
SEL = t1;
if(t1 < 1)
    t1 += N;
t2 = t1 - NSZ2;
if(t2 < 1)
    t2  += N;

p1 = w1[j]; /* p1 is the oldest sample being replaced by q1 */

/* compare p1 and q1 and update m1-array */

if( p1 < q1){
     for(k = (p1+2); k <= (q1+1); k++)
             m1[k-1] += inc;
 }
 if( p1 > q1){
     for(k = (q1+2); k <= (p1+1); k++)
             m1[k-1] -= inc;
 }

 /* Compute i1 th Order Statistic from the elements of m1-array */

 z1 = 0;
 for(k = 0; k < r; k++) {
     a[k] = 1;
     z1 = z1 + (a[k] << (r-k-1));
     z1 = m1[z1] < t1 ? z1 - (a[k] << (r-k-1)) : z1;
 }

 p2 = w2[j];  /* p2 is the oldest sample being replaced by q1 */

 /* compare p2 and q1 and update m2-array */

 if( p2 < q2){
     for(k = (p2+2); k <= (q2+1); k++)
             m2[k-1] += inc;
 }
 if( p2 > q2){
     for(k = (q2+2); k <= (p2+1); k++)
             m2[k-1] -= inc;
 }

 /* Compute i2 th Order Statistic from the elements of m2-array */

 z2 = 0;
 for(k = 0; k < r; k++) {
     a[k] = 1;
     z2 = z2 + (a[k] << (r-k-1));
     z2 = m2[z2] < t2 ? z2 - (a[k] << (r-k-1)) : z2 ;
 }

 /* Compute z based on the following logic */
```

...Cont'd

```
        if(SEL <= 0)  z   = z2 + L;
        else
           z  = z1;

        printf("%d %d\n",q, z);
        w[j] = q;        /* Replace sample p by q */
        w1[j] = q1;      /* Replace sample p1 by q1 */
        w2[j] = q2;      /* Replace sample p2 by q2 */
        j = (++j) > (N-1) ? 0 : j;
      }
}
```

Figure 2.7: C program for the execution of ROS using the Expansion algorithm

2.5 Summary

In this chapter the OSF has been described in detail. Analysis of these filters shows that these filters have good edge preservation properties and are very suitable for the removal of impulsive noise. These two facts make these filters very attractive for various filtering applications. Some important modifications and extensions of the OSF have also been presented.

Section 2.4 discussed the various algorithms for Real-time implementation of OSFs. The C language code implementing the sequence of operations of the algorithms discussed have also been presented.

Chapter 3

Implementation of Order Statistic Filters on a General Purpose Digital Signal Processor

3.1 Introduction

In this chapter we consider the implementation of Order Statistic Filters on a general purpose DSP; the TMS320C30 processor.

General purpose DSP implementations have a number of advantages as compared to VLSI implementations. They are cost effective and fast to implement. VLSI implementations take a longer time for implementation but can be customized and hence for real-time applications have fast processing times.

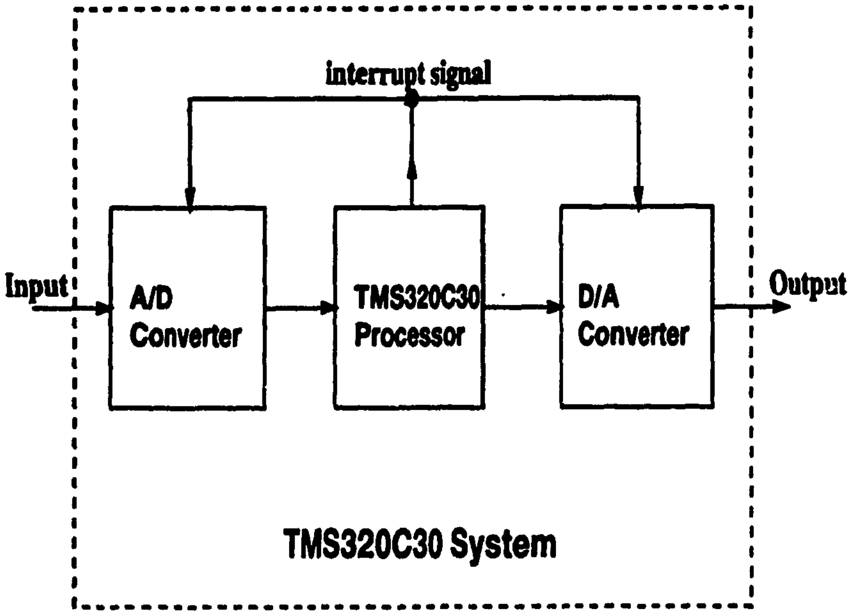


Figure 3.1: TMS320C30 System Board

In the following sections a brief description of the TMS320C30 digital signal processor (DSP) and the Analog interfaces is given. Section 3.4 deals with the implementation of OSFs on the TMS320C30 DSP. The algorithms considered are the sequential mode and expansion algorithm.

3.2 TMS320C30 DSP

The block diagram of the complete TMS320C30 Implementation is shown in Figure 3.1. The TMS320C30 system board [42] is an AT bus compatible board with dual channel 16 bit A/D and D/A systems.

The block diagram of the TMS320C30 DSP is given in Figure 3.2. The

TMS320C30 DSP [43] is a high-performance CMOS 32-bit device capable of executing upto 33 MFLOPS. It consists of integer and floating-point arithmetic units, 2048 x 32 bit words of on-chip RAM, 4096 x 32 bit words of on-chip ROM, control unit and parallel and serial interfaces. A performance of 16.7 million instructions per second is achieved, while operating from a 33.3 MHz clock.

In the next section a brief description of the Central Processing Unit and Memory Organization of the TMS320C30 DSP is given. Other details such as peripherals are not given but can be found in [43].

3.2.1 Central Processing Unit (CPU)

The TMS320C30 has a register-based CPU architecture. The CPU consists of the following components:

- Floating-point/integer multiplier : The multiplier performs single-cycle multiplications on 24-bit integer and 32-bit floating-point values. The TMS320C30 implementation of floating-point arithmetic allows for floating-point operations at fixed-point speeds via a 60-ns instruction cycle and a high degree of parallelism. To gain even higher throughput, a multiply and ALU operation can be performed in a single cycle by using parallel instructions.

  When performing floating-point multiplication the inputs are 32-bit floating-point numbers, and the result is a 40-bit floating-point number. When performing integer multiplication, the input data is 24 bits and yields a 32-bit

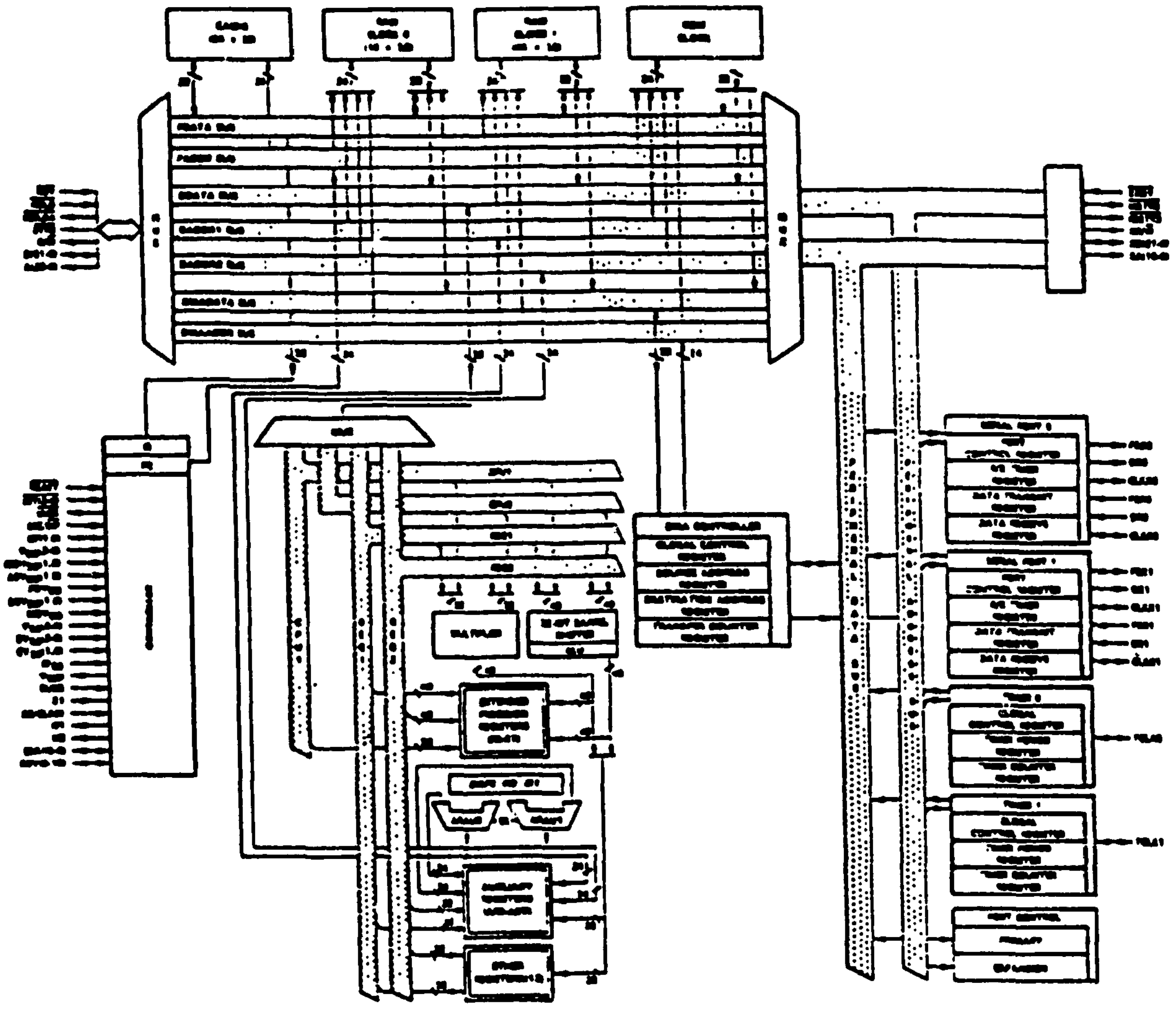


Figure 3.2: TMS320C30 Digital Signal Processor (Adapted from [43])

result.

- Arithmetic Logic Unit (ALU) for performing arithmetic (floating-point, integer) and logical operations : The ALU performs single-cycle operations on 32-bit integer, 32-bit logical, and 40-bit floating-point data, including single-cycle integer and floating-point conversions. Results of the ALU are always maintained in 32-bit integer or 40-bit floating-point formats. The barrel shifter is used to shift up to 32 bits left or right in a single cycle.

  Internal buses, CPU1/CPU2 and REG1/REG2, carry two operands from memory and two operands from the register file, thus allowing parallel multiplies and adds/subtracts on four integer or floating-point operands in a single cycle.

- 32-bit barrel shifter

- Internal buses (CPU1/CPU2 and REG1/REG2)

- Auxiliary register arithmetic units (ARAUs): Two auxiliary register arithmetic units (ARAU0 and ARAU1) can generate two addresses in a single cycle. The ARAUs operate in parallel with the multiplier and ALU. They support addressing with displacements, index registers (IR0 and IR1), and circular and bit-reversed addressing.

- CPU register file: The TMS320C30 provides 28 registers in a multiport register file that is tightly coupled to the CPU. All of these registers can be operated upon by the multiplier and ALU, and can be used as general-purpose registers. However, the registers also have some special functions for which they are more suited than others. For example, the eight extended-precision registers are especially suited for maintaining extended-precision floating-point results. The eight auxiliary registers support a variety of indirect addressing modes and can be used as general-purpose 32-bit integer and logical registers. The remaining registers provide system functions such as addressing, stack ma· --ment, processor statrus interrupts, and block repeat.

The registers names and assigned functions are listed in Figure 3.3.

3.2.2 Memory Organization

The total memory space of the TMS320C30 is 16M (million) 32-bit words. Program, data, and I/O space are contained within this 16M-word address space, thus allowing tables, coefficients, program code, or data to be stored in either RAM or ROM. In this way, memory usage can be maximized and memory space allocated as desired.

Figure 3.4 shows how the memory is organized on the TMS320C30. RAM blocks 0 and 1 are each 1K x 32 bits. The ROM block is 4K x 32 bits. A 64 x 32-bit instruction cache is provided to store often repeated sections of code thus greatly reducing the number of off-chip accesses necessary.

REGISTER NAME	ASSIGNED FUCNTION
R0	Extended-precision register 0
R1	Extended-precision register 1
R2	Extended-precision register 2
R3	Extended-precision register 3
R4	Extended-precision register 4
R5	Extended-precision register 5
R6	Extended-precision register 6
R7	Extended-precision register 7
AR0	Auxiliary register 0
AR1	Auxiliary register 1
AR2	Auxiliary register 2
AR3	Auxiliary register 3
AR4	Auxiliary register 4
AR5	Auxiliary register 5
AR6	Auxiliary register 6
AR7	Auxiliary register 7
DP	Data page pointer
IRO	Index register 0
IR1	Index register 1
BK	Block size
SP	System stack pointer
ST	Status register
IE	CPU/DMA interrupt enable
IF	CPU interrupt flags
IOF	I/O flags
RS	Repeat start address
RE	Repeat end address
RC	Repeat counter
PC	Program counter

Figure 3.3: CPU Registers

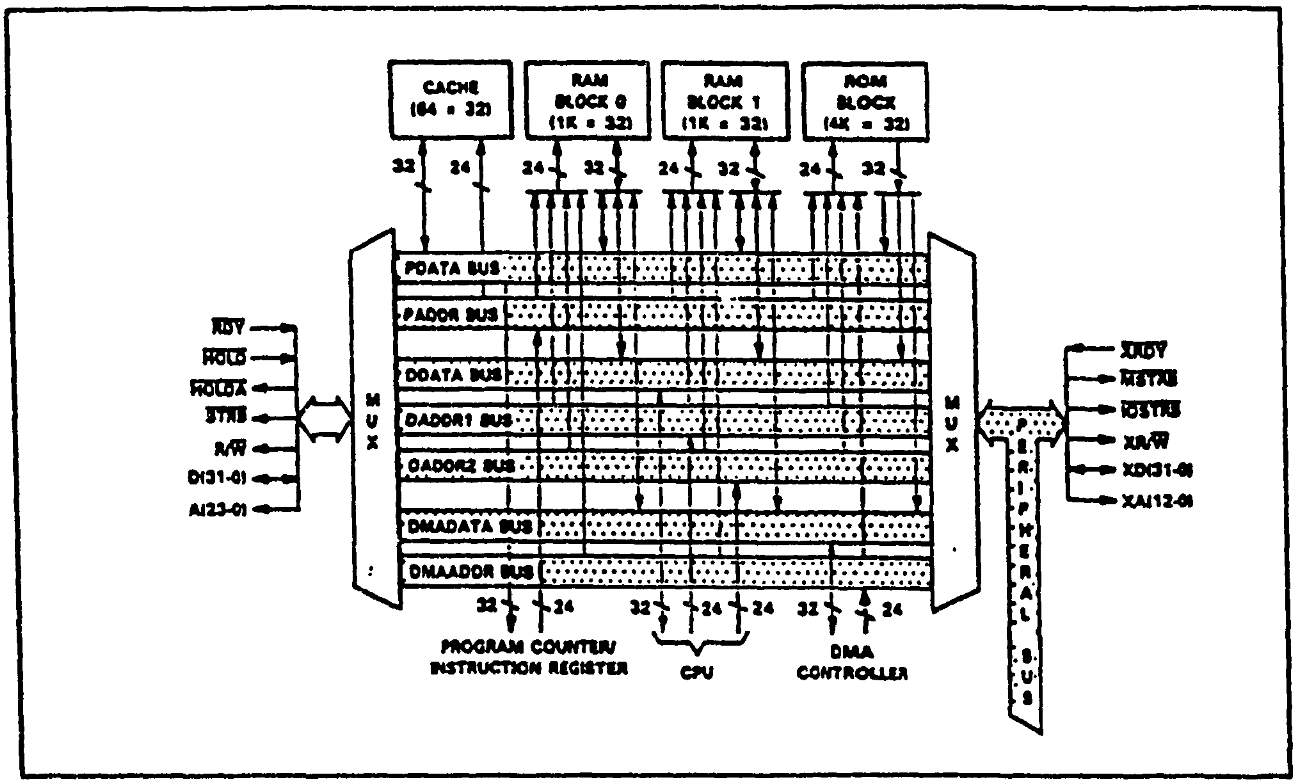


Figure 3.4: Memory Organization (Adapted from [43])

FUNCTION	ADDRESS
Read/write Channel A A/D and D/A	804000
Read/write Channel B A/D and D/A	804001
Generate Software Conversion Trigger	804008

Figure 3.5: Analog I/O Mapping

3.3 Analog Interfaces

The complete analog I/O subsystem [42] consists of two separate channels along with their own sample/hold amplifier, A/D, D/A. The A/D and D/A converters are Burr-Brown PCM78P devices, which offer 16-bit precision and sampling rates of up to 200 Khz.

3.3.1 Addressing

The interface is accessed through three 16-bit secondary I/O bus mapped registers. These registers are mapped in the region between 804000 hex and 804000 hex. The addressing used is shown in Figure 3.5.

The first two registers are used to access the A/D and D/A converters on the two Analog I/O channels. The A/Ds and D/As deal with data in the 16 bit 2's complement format. Data from the A/D converter is received by means of the I/O register located at address 0*x*804000 or 0*x*804001 (on the high half of the data bus). To output data to the D/A at the same addresses, data is moved to the I/O-mapped register (on the high half of the data bus).

Location 804008 hex accesses a register which is used to start a conversion of the A/D's and D/A's. All the registers are accessed with two memory wai. states. This results in an overall time of 180 nsec to access each register.

The A/D's and D/A's actually use serial digital data for output/input. On each channel of I/O, the A/D output is tied through a 16-bit shift register to the D/A input. This means that with no processor intervention, the A/D output will be shifted directly into the shift register, with the previous shift register contents being simultaneously shifted into the D/A input. Thus, by default the anlaog input signal on each channel will be echoed directly to the corresponding analog output channel, with a one-sample-time delay.

3.3.2 Sample Timing

The sampling rates of the A/D and D/A can be controlled by the on-chip counter/timer. A/D and D/A conversions are initiated by an on-chip counter/timer or by an external trigger signal. The timer consists of a 32 bit up-counter and a 32 bit period register. The value of the counter is continuously incremented at an 8.33 Mhz rate (once every 120 nsec). When the counter equals the period register it puts out a pulse that initiates the A/D and D/A conversions and loads a zero starting count value to the 32 bit counter. To set the timer in the correct mode $6C1$ hex is written to address 808030 hex. This enables timer 1 to use the internal clock and to produce a negative going pulse.

The analog I/O sample rate is set by writing a 'counter load value' into the peripheral mapped period register at location 808038 hex.

$$Counter\ \ Load\ \ Value = Period\ \ in\ \ usec/0.12$$

A/D's and D/A's can be dealt through an interrupt service routine. After the timer starts an A/D conversion, the A/D will perform the conversion, then output an end-of-convert signal. This end-of-convert signal can be input to the 'C30' as the *INT1* interrupt request. If *INT1* iss enabled, then the interrupt service routine can immediately read one or both A/D's, and then if desired, write to one or both D/A's. To enable the interrupt, a '1' to bit 1 of the 'C30's IE Register, and a '1' to bit 13 of the 'C30's Status Register.

3.4 TMS320C30 Implementation of OSFs

In this section we consider the implementation of OSFs on the TMS320C30 DSP. Algorithms for the computation of the ROS proposed by Rama Murthy and Swamy [38] which were discussed in Chapter 2 are used.

We consider the sequential mode of execution and expansion algorithm for our implementation. The parallel mode of execution has not been implemented on the TMS320C30 DSP due to the fact that evaluation of equation 2.4 requires too many processor clock cycles. This algorithm is more suitable for VLSI implementation using the architecture described in [38] than on a general purpose DSP.

3.4.1 Sequential Mode of Execution

The flow diagram for the implementation of the OSF by the sequential mode of execution on the TMS320C30 DSP is given in Figure 3.6. Each time the processor is interrupted a new sample is stored in $w_N(n)$ and at the same time the computed ROS sample is written to the D/A converter. $w_N(n)$ is an N location circular buffer $w_N(j), j = 0, 1, ..., N-1$ configured such that a new sample from the A/D converter is written into the memory location where the oldest sample was residing.

The C language code implementing the sequence of opera ion in the Flow diagram given in Figure 3.6 is shown in Figure 3.7.

The total number of cycles taken by the algorithm is approximately 5365 . Each cycle takes 60ns and hence the total time taken by the algorithm is 321us .

The maximum frequency for which the algorithm works efficiently is 1553 Hz .

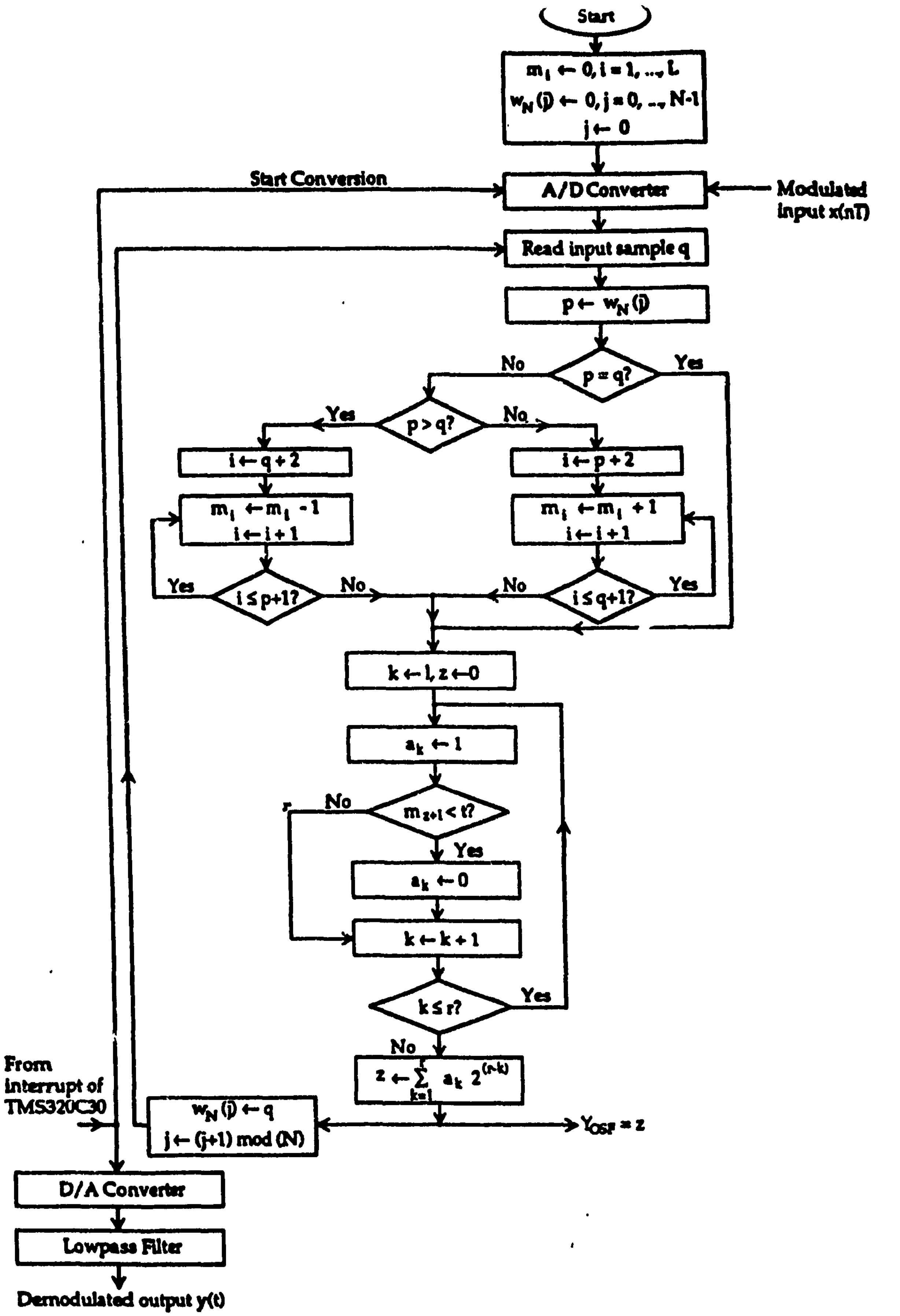


Figure 3.6: Flow diagram for the Sequential Mode of execution of the ROS Algorithm using the TMS320C30 DSP

```
/* C code for the implementation of the OSF on the TMS320C30 DSP */
/* This program executes ROS algorithm in the sequential mode */

/* The following lines are inserted into the C code to enable interrupt
   1 to function */

asm("  .sect  \".int02\"");
asm("  .word  _c_int02");
asm("  .text");

/* The following 5 lines are for intializing the necessary registers */

#define   TIMER_CONTROL_REG  ((unsigned int *)0x808030)
#define   ADC_COUNTER_REG    ((unsigned int *)0x808038)
#define   PRIMCTL_REG        ((unsigned int *)0x808064)
#define   EXPCTL_REG         ((unsigned int *)0x808060)
#define   PRIMWD             0x000800
#define   EXPWD              0x000000

int *ADCaddress; /* pointer to A/D address */
int *DACaddress; /* pointer to D/A address */
int temp1;       /* store output sample of A/D in temporary location temp1 */
int temp2;       /* store input sample to D/A in temporary location temp2 */

/* Definition of OSFs parameters */
#define   r              8    /* r is the resolution of the input samples */
#define   N              21   /* N is the size of the window */
#define   t              2    /* order of the OSF, i = 20 so that t=N+1-i */

int     w[N], m[1<<r], L, a[r];

main()
{
     int p, k, q, z, unity = 1, j = 0;

     /* Initialization of elements of m-array and the window */

     for(k = 0; k < N; k++) w[k] = 0;
     L =  (1 << r) ;
     for(k = 1; k <= L; k++) m[k]=0;

     *PRIMCTL_REG = PRIMWD;  /* Set up primary bus wait states */
     *EXPCTL_REG = EXPWD;    /* Set up expansion bus wait states */

    /* The following 5 lines set up timer1 and analog interfaces */

    *TIMER_CONTROL_REG = 0x601; /* Reset control register */
    *ADC_COUNTER_REG = 204;     /* Set period register */
    *TIMER_CONTROL_REG = 0x6c1; /* Set control register */
    ADCaddress = (int *)0x804000; /* A/D address */
    DACaddress = (int *)0x804001; /* D/A address */

    /* Enable interrupt for timer1 and set global interrupt enable */

    asm(" OR  2h, IE");
    asm(" OR   2000h, ST");
```

..Cont'd

```
  /* Begin Computation of the ROS */

  while(1){

          /* Scaling by L and Biasing by L/2 */

          q = (templ >> r) + (1 << (r-1));

          p = w[j]; /* Read p, the oldest sample */

          /* compare p and q and update m-array */

          if( p < q){
               for(k = (p+2); k <= (q+1); k++)
                      m[k-1] += unity;
          }
          if( p > q){
               for(k = (q+2); k <= (p+1); k++)
                      m[k-1] -= unity;
          }

          /* Compute Order Statistic from the elements of m-array */

          z = 0;  /* Initialize z */
          for(k = 0; k < r; k++) {
                a[k] = 1;
                z = z + (a[k] << (r-k-1));
                z = m[z] < t ? z-(a[k] << (r-k-1)) : z;
          }

          /* Scaling and biasing for D/A conversion */

          temp2 = (z - (1<<(r-1))) << r;

          w[j] = q; /* Replace sample p by q */
          j = (++j) > (N-1) ? 0 : j;
    }
}

/* Interrupt service routine */

c_int02(){
  templ = *ADCaddress >> 16; /* Read scaled output of A/D into templ */
  *DACaddress = temp2 << 16; /* Write scaled input to D/A from temp2 */

}
```

Figure 3.7: C program for the implementation of OSF on TMS320C30

3.4.2 Expansion Algorithm

The flow diagram for the implementation of the OSF by the Expansion algorithm on the TMS320C30 DSP is given in Figure 3.8. Each time the processor is interrupted a new sample is stored in $w_N(n)$, and at the same time the computed ROS sample is written to the D/A converter. $w_N(n)$ is an N location circular buffer $w_N(j), j = 0, 1, ..., N-1$ configured such that a new sample from the A/D converter is written into the memory location where the oldest sample was residing.

The C language code implementing the sequence of operation in the Flow diagram given in Figure 3.8 is shown in Figure 3.9.

The total number of cycles taken by the algorithm is approximately 10669 cycles. Each cycle takes 60ns and hence the total time taken by the algorithm is 640 us . The maximum frequency for which the algorithm works efficiently is 781 Hz.

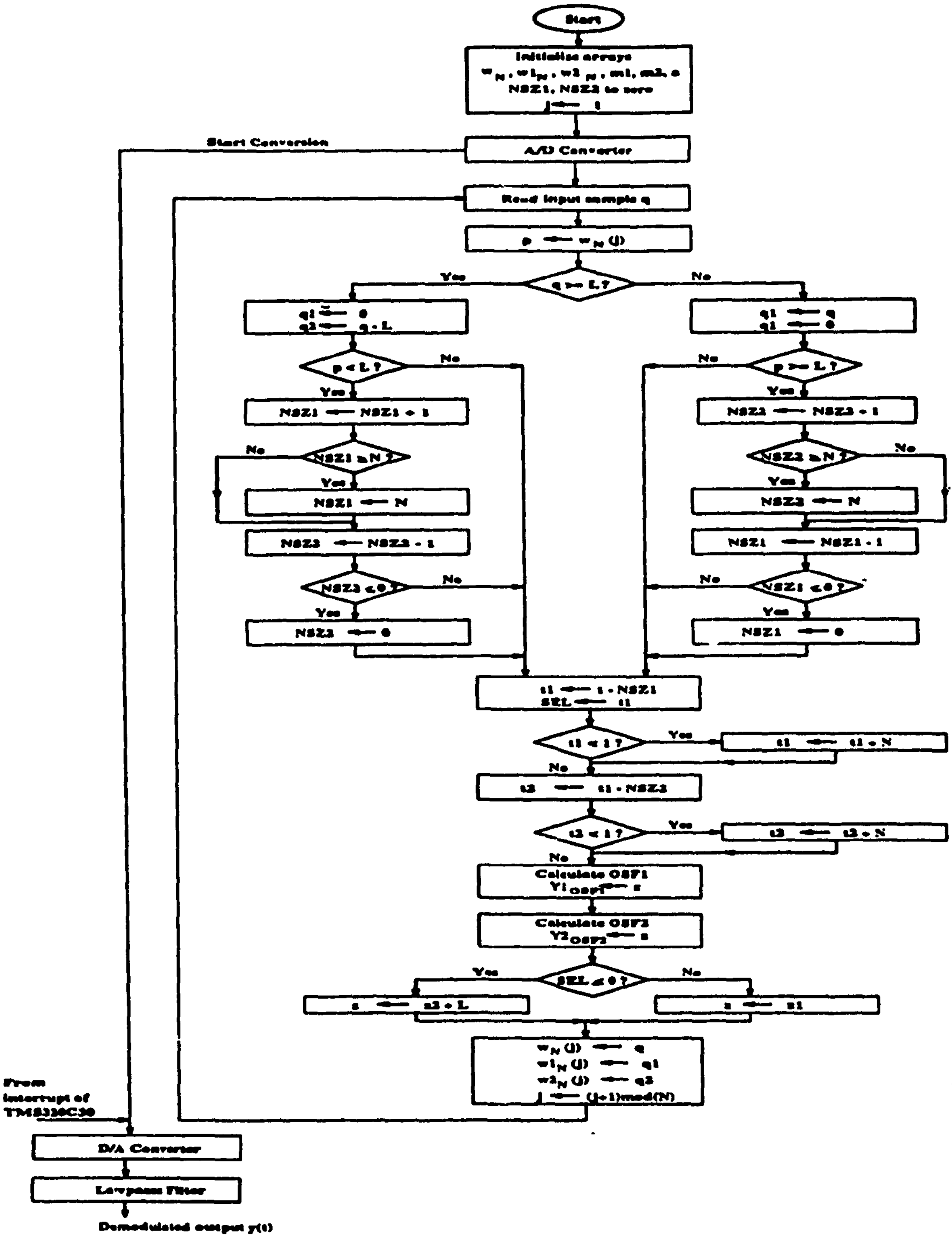


Figure 3.8: Flow diagram of the Expansion algorithm for the computation of the ROS Algorithm using the TMS320C30 DSP

```
/* C code for the implementation of the OSF on the TMS320C30 DSP */
/* This program executes ROS algorithm using two OSFs (Expansion algorithm) */

/* The following lines are inserted into the C code to enable interrupt
   1 to function */

asm("   .sect  \".int02\"");
asm("   .word  _c_int02");
asm("   .text");

/* The following 5 lines define the hardware registers required accessing */

#define   TIMER_CONTROL_REG   ((unsigned int *)0x808030)
#define   ADC_COUNTER_REG     ((unsigned int *)0x808038)
#define   PRIMCTL_REG         ((unsigned int *)0x808064)
#define   EXPCTL_REG          ((unsigned int *)0x808060)
#define   PRIMWD              0x000800
#define   EXPWD               0x000000

int *ADCaddress; /* pointer to A/D address */
int *DACaddress; /* pointer to D/A address */
int temp1;       /* store output sample of A/D in temporary location temp1*/
int temp2;       /* store input sample to D/A in temporary location temp2 */

/* Definition of OSFs parameters */

#define   r              8    /* r is the resolution of the input samples */
#define   N              8    /* N is the size of the window */
#define   t              2    /* order of the OSF = 7 so that t=N+1-i */

int     w[N], w1[N], w2[N], m1[1<<r], m2[1<<r], L, a[r];

main()
{
        int q, q1, q2, p, p1, p2, k, z, z1, z2, inc = 1,j = 0,
            NSZ1 = 0, NSZ2 = N, t1, t2, SEL;

        /* Initialization of elements of m1-array, m2-array and the windows */

        for(k = 0; k < N; k++) w[k] = 0;
        for(k = 0; k < N; k++) w1[k] = 0;
        for(k = 0; k < N; k++) w2[k] = 0;
        L = 1 << r;
        for(k = 0; k < L; k++) m1[k]=0;
        for(k = 0; k < L; k++) m2[k]=0;

        *PRIMCTL_REG = PRIMWD;          /* Set up primary bus wait states */
        *EXPCTL_REG = EXPWD;            /* Set up expansion bus wait states */

        /* The following 5 lines set up timer1 and analog interfaces */

        *TIMER_CONTROL_REG = 0x601;     /* Reset control register */
        *ADC_COUNTER_REG = 204;         /* Set period register */
        *TIMER_CONTROL_REG = 0x6c1;     /* Set control register */
        ADCaddress = (int *)0x804000;   /* A/D address */
        DACaddress = (int *)0x804001;   /* D/A address */
```

...Cont'd

```
/* Enable interrupt for timer1 and set global interrupt enable */

asm(" OR  2h, IE");
asm(" OR   2000h, ST");

/* Begin Computation of the ROS */

while(1){
    q = (templ>>(r-1))+(1<<r); /* Scaling by L/2 and Biasing by L */
    p = w[j],                  /* Read p, the oldest sample */

    /* Update the q1, q2 based on the value of q and update NSZ1 and
       NSZ2 based on the value of p */

    if((q >= L)) {
          q1 = 0;
          q2 = q - L;
          if(p < L) {
               NSZ1 += 1;
               if(NSZ1 >= N) NSZ1 = N;
               NSZ2   NSZ2 - 1;
               if(NSZ2 <= 0) NSZ2 = 0;
          }
    }
    if((q < L)) {
          q1 = q;
          q2 = 0;
          if(p >= L){
               NSZ2 += 1;
               if(NSZ2 >= N) NSZ2 = N;
               NSZ1 = NSZ1 - 1;
               if(NSZ1 <= 0) NSZ1 = 0;
          }
    };

    /* Computation of t1 and t2 */

    t1 = t - NSZ1 ;
    SEL = t1;
    if(t1 < 1)
        t1 += N;
    t2 = t1 - NSZ2;
    if(t2 < 1)
        t2  += N;

    p1 = w1[j]; /* p1 is the oldest sample being replaced by q1 */

    /* compare p1 and q1 and update m1-array */

    if( p1 < q1){
         for(k = (p1+2); k <= (q1+1); k++)
                m1[k-1] += inc;
     }
     if( p1 > q1){
         for(k = (q1+2); k <= (p1+1); k++)
                m1[k-1] -= inc;
     }
```

...Cont'd

```
            /* Compute Order Statistic from the elements of m1-array */

            z1 = 0;
            for(k = 0; k < r; k++) {
                a[k] = 1;
                z1 = z1 + (a[k] << (r-k-1));
                z1 = m1[z1] < t1 ? z1 - (a[k] << (r-k-1)) : z1;
            }

            p2 = w2[j];  /* p2 is the oldest sample being replaced by q1 */

            /* compare p2 and q1 and update m2-array */

            if( p2 < q2){
                for(k = (p2+2); k <= (q2+1); k++)
                        m2[k-1] += inc;
            }
            if( p2 > q2){
                for(k = (q2+2); k <= (p2+1); k++)
                        m2[k-1] -= inc;
            }

            /* Compute Order Statistic from the elements of m2-array */

            z2 = 0;
            for(k = 0; k < r; k++) {
                a[k] = 1;
                z2 = z2 + (a[k] << (r-k-1));
                z2 = m2[z2] < t2 ? z2 - (a[k] << (r-k-1)) : z2 ;
            }

            /* Compute z based on the following logic */

            if(SEL <= 0)  z  = z2 + L;
            else
                z  = z1;

            temp2 = ( z - (1 << r )) << (r - 1); /* Scaling and biasing
                                                    for D/A conversion */

            w[j] = q;       /* Replace sample p by q */
            w1[j] = q1;     /* Replace sample p1 by q1 */
            w2[j] = q2;     /* Replace sample p2 by q2 */
            j = (++j) > (N-1) ? 0 : j;
         }
}

/* Interrupt service routine */

c_int02(){
     templ = *ADCaddress >> 16; /* Read scaled output of A/D into templ */
     *DACaddress = temp2 << 16; /* Write scaled input to D/A from temp2 */
}
```

Figure 3.9: C program for the implementation of OSF on the TMS320C30 DSP using the Expansion Algorithm

3.5 Summary

In this chapter, the implementation of order statistic filters on a general purpose DSP, the TMS320C30 processor was considered. The algorithms considered were the sequential mode and the expansion algorithms. The parallel mode of execution was not implemented on the TMS320C30 DSP due to the fact that evaluation of equation 2.4 requires too many processor clock cycles. This algorithm is more suitable for a VLSI implementation using the architecture described in [38] than on a general purpose DSP.

Chapter 4

Applications of Order Statistic Filters

Order statistic filters have a number of applications that are useful for solving problems encountered in areas such as computational geometry, image processing, computer graphics, pattern classification and communications.

In this chapter we shall consider a few of these applications of OSFs. In addition we consider one such application for our implementation on the TMS320C30 processor.

4.1 Application of Order Statistic Filter to Computational Geoemetry

One of the most basic operations performed on a computer is searching. For this purpose, a number of search problems proposed by Dobkin and Lipton [44] are presented with solutions given by Rama Murthy and Swamy [45].

1. Given a set of m lines in a plane with equation $y_k = a_k x + b_k, k = 1,\cdot \quad ,m$ and a point $P(x_0, y_0)$, determine whether this point P lies on any of the lines.

   The solution for the above problem amounts to finding out if the minimum of the sequence $w_N(n) = \{|y_0 - a_k x_0 - b_k|, k = 1, ..., m\}$, i.e., $x_{(1)}(n) = 0$. The algorithm presented in Chapter 2 are capable of computing the ROS, i.e., they can be used to repetitively answer the above problem whenever a new line replaces one amongst the initial set of m lines. Further, since the ROS algorithm can compute any desired ith order statistic, it can also be used to solve a modified form of the above problem which might require one to find out, say, the third farthest line (based on the distance) from $P(x_0, y_0)$.

2. Given a set of m points $(x_k, y_k), k = 1.2, ..., m$ in a plane and a straight line $y = ax + b$, determine if this line passes through any of the points.

   This problem can be considered as the dual of the previous problem. The solution for this problem amounts to finding out if $x_{(1)}(n)$ of the sequence $w_N(n) = \{d_k, k = 1, ..., m\}$ where $d_k = |y_k - ax_k - b|$ is zero. The algorithms presented in Chapter 2 are capable of answering the above problem under the condition that a new point replaces one amongst the set of m points $(x_k, y_k), k = 1, ..., m$. The modified form of the above problem which might require one to find out, say, the third closest point to the straight line can be solved using the ROS algorithms.

3. Given a set of points $(x_k, y_k), k = 1, 2, ..., m$ in a plane and a new point $P(x_0, y_0)$, determine to which one amongst the original points it is closest to. This query amounts to finding out if zero is the minimum of sequence $w_N(n) = \{d'_K, k = 1, ..., m\}$ where d'_k is the Euclidean distance between (x_k, y_k) and $P(x_0, y_0)$, i.e., $d'_k = |\sqrt{(x_k - x_0)^2 + (y_k - y_0)^2}|$. Again, the ROS algorithms can be used for repetitive answering under the condition that a new point replaces one amongst the set of m points $(x_k, y_k), k = 1, ..., m$. To answer the modified form which requires one to find out say, the farthest point from P , one can use the ROS algorithms.

4.2 Application of Order Statistic Filter to Planar Search

We consider here the "post office" problem which can be reduced to the planar search problem. Given m cities or "post offices", the problem is to determine which post office is nearest to a given point.

This problem may be solved [45] by considering each post office as a coordinate set (x_k, y_k) on a plane. The problem then amounts to finding out if $x_{(1)}(n)$ of the sequence $w_N(n) = \{d_k, k = 1, ..., m\}$ *where* $d_k = |y_k - ax_k - b|$ is zero .

4.3 Application of Order Statistic Filter to Speech Recognition

In this section we consider the problem of finding the closest points in spaces of small dimensions. An example of such a problem occurs in the area of speech recognition. Sounds can be classified according to a set of less than 8 characteristics, and thus a database for speech recognition system to consist of a set of points is considered. When such a system is used to understand a speaker, the method used is to find for each sound uttered the closest sound in the database.

Let a sound be thought of an N-dimensional vector $(p_1, p_2, ..., p_N)$. Given a fixed set of K sounds $Q = \{q^{(1)}, q^{(2)}, ..., q^{(K)}\}$ each of which is assumed to be representative of some class of patterns, and a given new pattern p, a problem that often arises is that of determining to which of the classes, p "most likely" belongs. One way to choose the most likely class is to apply the nearest neighbor rule [46]. This means that if ρ is a metric on R^n, find that $q^{(s)} \epsilon Q$ for which $\rho(q^{(s)}, p)$ is minimal and classify p as belonging to the class which $q^{(s)}$ represents.

ρ could be chosen to be L_∞ or maximum norm. The problem of pattern classification reduces to finding $||q^{(\cdot)} - p||_\infty$ i.e., maximum of the sequence $x^{(1)} = \{|p_k - q_k^{(\cdot)}|, k = 1, ..., N\}$ for every $q^{(\cdot)}$ and then finding the minimum of $\underline{x}^{(1)} = \{\rho(q^{k)}, p), k = 1, ..., K\}$.

4.4 Application of Order Statistic Filter to Signal Processing

Order statistic filters can be used to eliminate impulse noise. This is due to their property of eliminating impulse-like structures while passing edges unperturbed. In the following section this property is used for the digital AM detection with and without corruption by impulse noise.

4.4.1 An Application of the TMS320C30 Based OSF

As a practical application, the OSF was implemented on the TMS320C30 board for the real-time demodulation of the AM wave [40]. The configuration of the hardware set-up is shown in Figure 4.1.

The modulator was implemented using a Motorola 1495L. The resulting analog AM-modulated doublε side band signal $x(t)$ is given by the equation

$$x(t) = A_c(1 + m cos(2\pi f_m t) cos(2\pi f_c t)$$

where m is the modulation index, A_c is the amplitude of the carrier signal, f_m is the frequency of the modulated signal and f_c is the frequency of the carrier signal.

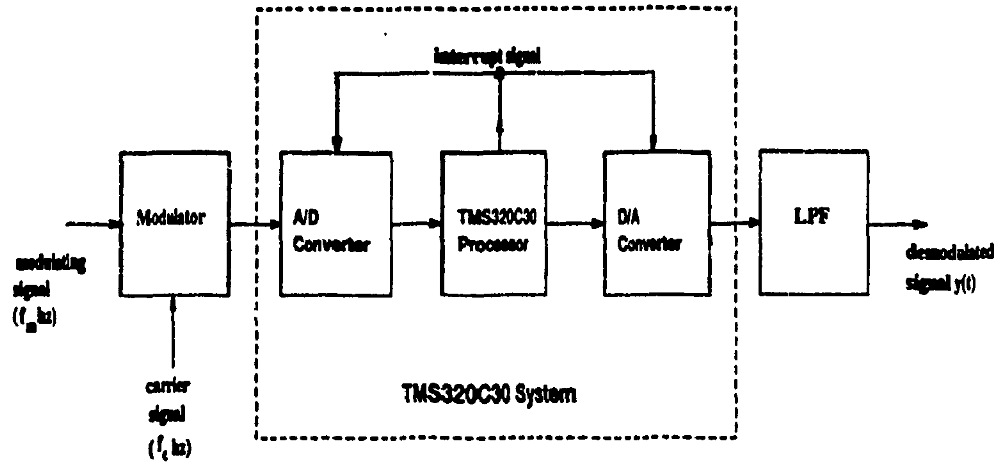


Figure 4.1: A TMS320C30 Implementation

The ciruit diagram is given in Figure 4.2. The resulting AM modulated double side band signal was sampled at 40 KHz using the A/D converter on the system board to get $x(nT)$, the digital signal corresponding to $x(t)$. In equation 4.4.1 A_c was chosen to be 6 volts and the modulation index m was taken to be 0.66 without loss of any generality.

For the sequential mode of execution, the input carrier frequency f_c was taken to be 1.14 Khz and the input sinusoidal modulating frequency f_m was taken as 60 hz. The sampled AM modulated signal was then fed to a twentieth order OSF with a window size of 21. The resulting output of the OSF fed to the D/A results in a staircase output that is subsequently filtered by a 4th order, 4 Khz cut-off LPF with a Butterworth response. It must be noted that the LPF will not destroy the edges as the cut-off frequency is only 4 Khz. The circuit used is shown in Figure 4.3.

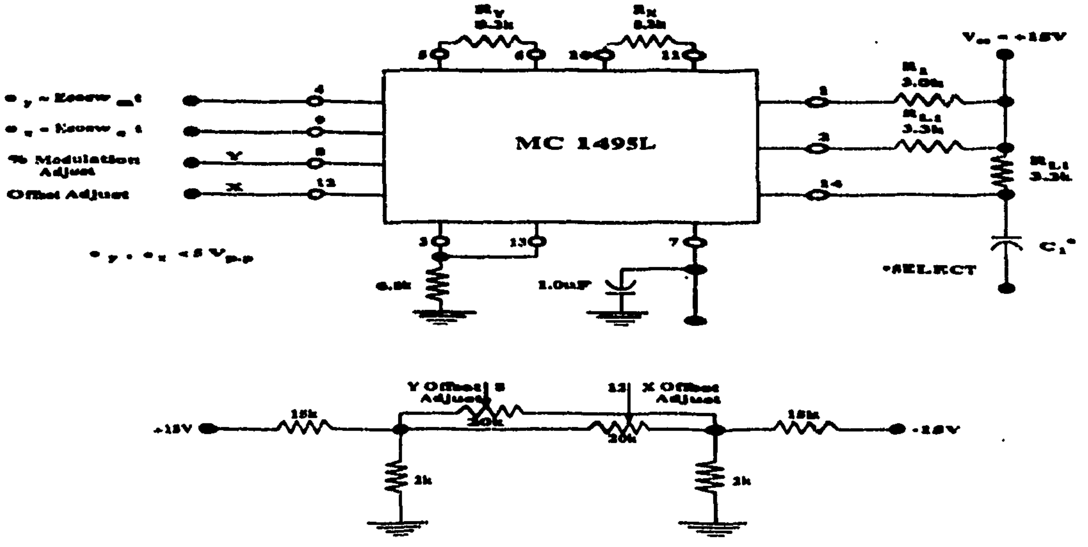


Figure 4.2: Modulator Circuit

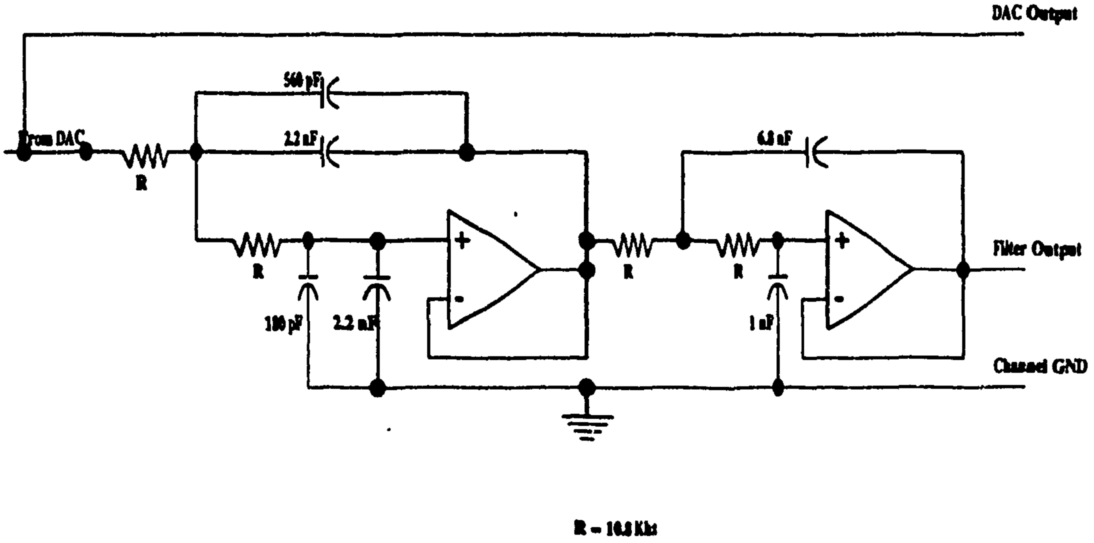


Figure 4.3: A Fourth Order Lowpass Filter

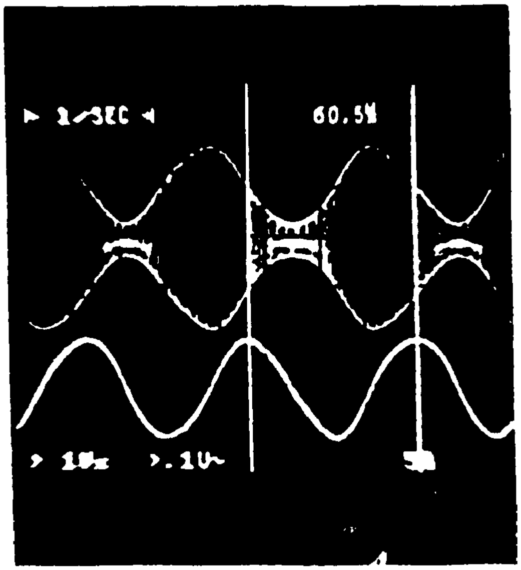


Figure 4.4: Photograph showing AM Wave along with the Recovered Modulating Signal

The modulated AM wave $x(t)$ and the demodulated wave $y(t) = cos(2\pi f_m t)$ are shown in Figure 4.4.

4.5 Summary

A number of applications encountered in areas such as computational geometry, image processing, computer graphics, pattern classification and communications were considered. It is seen that the OSF provides efficient solutions for the set of problems considered. As a practical application, the OSF was implemented on the TMS320C30 DSP for the Real-time demodulation of a double side-band amplitude modulated (AM) signal.

Chapter 5

Conclusions

Order statistic filters are a class of non-linear filters whose output is a linear combination of the order statistics of the input. Analysis of these filters shows that these filters have good edge preservation properties and are very suitable for the removal of impulsive noise.

In this thesis, Order statistic filters were discussed in detail. Real-time implementations of OSFs using the TMS320C30 digital signal processor was considered. The implementations made use of the running order statistic (ROS) computation algorithms given in [38].

In Chapter 2, the properties of ranked order filters, followed by some modifications of order statistic filters was presented. Various algorithms for the real-time implementation of OSFs was also considered.

In Chapter 3, the implementation of order statistic filters on a general purpose

DSP, the TMS320C30 processor was considered. The algorithms considered were the sequential mode and the expansion algorithms. It was seen that the implementation of these filters on general purpose DSP boards are limited to low frequencies. The parallel mode of execution was not implemented on the TMS320C30 DSP due to the fact that evaluation of equation 2.4 requires too many processor clock cycles. This algorithm is more suitable for a VLSI implementation using the architecture described in [38] than on a general purpose DSP.

In Chapter 4, a number of applications encountered in areas such as computational geometry, image processing, computer graphics, pattern classification and communications was presented. It is seen that the OSF provides provides efficient solutions for the set of problems considered. As a practical application, the OSF was implemented on the TMS320C30 DSP for the real-time demodulation of a double side-band amplitude modulated (AM) signal.

5.1 Future Work

Several promising avenues exist to extend the current research. These are as follows:

The performance of order statistic filters on a general purpose DSP, the TMS320C30 processor can be improved by implementing the order statistic filters in assembler.

In addition to the above, VLSI implementations of the algorithms considered in Chapter 2 would allow the filters to be operated at larger frequencies.

Bibliography

[1] N.Wiener, *Nonlinear problems in random theory.* New York, The Technology Press: John Wiley and Sons, Inc., 1958.

[2] J.W.Tukey, "Nonlinear (nonsuperposable) methods for smoothing data," in *Conf. Rec.*, p. 673, 1974 EASCON.

[3] W.K.Pratt, *Digital Image Processing.* Wiley, 1978.

[4] H.C.Andrews and B.R.Hunt, *Digital Image Restoration.* Prentice-Hall, 1977.

[5] B.I.Justusson, *Two-Dimensional Digital Signal Processing II: Transforms and Median Filters.* New York: Springer-Verlag: T.S.Huang, Ed., 1981.

[6] G.J.Yong and T.S.Huang, "The effect of median filtering in edge location estimation," *Computer Vision, Graphics and Image Processing*, vol. 15, pp. 224–245, 1981.

[7] S.S.Perlman, S.Eisenhandler, P.W.Lyons, and M.J.Shumila, "Adaptive median filtering for impulse noise elimination in real-time tv signals," *IEEE Trans. Communications*, vol. 35, pp. 646–652, June 1987.

[8] L.R.Rabiner, M.R.Sambur, and C.E.Schmidt, "Applications of a nonlinear smoothing algorithm to speech processing," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 23, pp. 552–557, December 1975.

[9] G.Arce and N.C.Gallagher, "State description for the root-signal set of median filters," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 30, pp. 894–902, December 1982.

[10] N.C.Gallagher, Jr., and G.L.Wise, "A theoretical analysis of the properties of median filters," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 29, pp. 1136–1141, December 1981.

[11] S.G.Tyan, *Two-Dimensional Digital Signal Processing II: Transforms and Median Filters.* New York: Springer-Verlag: T.S.Huang, Ed., 1981.

[12] F.Kuhlmann and G.L.Wise, "On second moment properties of median filtered sequences of independent data," *IEEE Trans. Commun.*, vol. 29, pp. 1374–1379, September 1981.

[13] E.Ataman, V.K.Aatre, and K.M.Wong, "Some statistical properties of median filters," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 29, pp. 1073–1075, October 1981.

[14] I.Pitas and A.N.Venetsanopoulos, *Nonlinear Digital Filters.* Massachusetts: Kluwer Academic Publishers, 1990.

[15] T.A.Nodes and N.C.Gallagher, "Median filters: some modifications and their properties," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 30, pp. 739–746, October 1982.

[16] J.P.Fitch, E.J.Coyle, and N.C.Gallagher, "Median filtering by threshold decomposition," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 32, pp. 1183–1188, December 1984.

[17] G.R.Arce, "Statistical threshold decomposition for recursive and nonrecursive median filters," *IEEE Trans. Inform. Theory.* vol. 32, pp. 243–253, March 1986.

[18] M.P.McLoughlin and G.R.Arce, "Deterministic properties of the recursive separable median filter," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 35, pp. 98–106, January 1987.

[19] G.R.Arce and M.P.McLoughlin, "Theoretical analysis of the max/median filter," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 35, pp. 60–69, January 1987.

[20] A.C.Bovik, T.S.Huang, and D.C.Munson, Jr., "A generalization of median filtering using linear combinations of order statistics," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 31, pp. 1342–1350, December 1983.

[21] H.G.Longbotham and A.C.Bovik, "Theory of order statistic filters and their relationship to linear FIR filters," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 37, pp. 275-287, February 1989.

[22] J.B.Bednar and T.L.Watt, "Alpha-trimmed means and their relationship to median filters," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 32, pp. 145-153, February 1984.

[23] H.-M.Lin and A.N.Wilson, Jr., "Median filters with adaptive length," *IEEE Trans. Circuits and Systems*, vol. 35, pp. 675-690, June 1988.

[24] A.Nieminen, P.Heinonen, and Y.Neuvo, "A new class of detail-preserving filters for image processing," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 9, pp. 74-90, January 1987.

[25] T.I.Haweel and P.M.Clarkson, "A class of Order Statistic LMS algorithms," *IEEE Trans. Signal Processing*, vol. 40, pp. 44-53, January 1992.

[26] P.D.Wendt, E.J.Coyle, and N.C.Gallagher, Jr., "Stack filters," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 34, pp. 898-911, August 1986.

[27] E.J.Coyle, "Ranked order operators and the mean absolute error criterion," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 36, pp. 63-76, January 1988.

[28] J.H.Lin and E.J.Coyle, "Stack filters and the mean absolute error criterion," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 36, pp. 1244-1254, August 1988.

[29] M.Gabbouj and E.J.Coyle, "Minimum mean absolute error stack filtering with structural constraints and goals," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 38, pp. 955–968, June 1990.

[30] J.H.Lin, T.M.Sellke, and E.J.Coyle, "Adaptive stack filtering under the mean absolute error criterion," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 38, pp. 935-954, June 1990.

[31] P.D.Wendt, "Nonrecursive and recursive stack filters and their filtering behaviour," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 31, pp. 2099–2107, December 1990.

[32] J.P.Fitch, "Software and VLSI algorithms for generalized rank order filtering," *IEEE Trans. Circuits and Systems*, vol. 34, pp. 553–559, May 1987.

[33] K.Oflazer, "Design and implementation of a single chip 1-D median filter," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 31, pp. 1164-1168, October 1983.

[34] T.S.Huang, G.J.Yang, and G.Y.Tang, "A fast two-dimensional median filtering algorithm," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 27, pp. 13–18, February 1979.

[35] E.Ataman, V.K.Aatre, and K.M.Wong, "A fast method for real-time median filtering," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 28, pp. 415–421, August 1980.

[36] V.V.B.Rao and K.S.Rao, "A new algorithm for real-time median filtering," *IEEE Trans. Acoust. Speech Signal Proc.*, vol. 34, pp. 1674–1675, December 1986.

[37] K.Chen, "Bit-Serial realizations of a class of nonlinear filters based on positive boolean functions," *IEEE Trans. Circuits and Systems*, vol. 36, pp. 785–794, June 1989.

[38] N.Rama Murthy and M.N.S.Swamy, "On the VLSI implementation of real-time order statistic filters," *IEEE Trans. Signal Processing*, vol. 40, pp. 1241–1252, May 1992.

[39] H.A.David, *Order Statistics*. New York: Wiley, 1970.

[40] N. Jr., "Median filters: A tutorial," *in Proc. ISCAS*, pp. 1737–1744, 1988.

[41] Woo-Jin Song and W.A.Pearlman, "Edge-preserving noise filtering based on adaptive windowing," *IEEE Trans. Circuits and Systems*, vol. 35, pp. 1048–1055, August 1988.

[42] SPECTRUM Signal Processing Inc, *TMS320C30 System Board*. SPECTRUM Signal Processing Inc., 1990.

[43] Texas Instruments, *Third Generation TMS320 User's Guide*. Texas Instruments, 1988.

[44] D.Dobkin and R.J.Lipton, "Multidimensional searching problems," *SIAM J. Comput.*, vol. 5, pp. 181–186, June 1976.

[45] N.Rama Murthy and M.N.Swamy, "On the computation of the running order statistic of a sequence and its applications," *submitted to IEEE Trans. Signal Processing*, 1993.

[46] R.O.Duda and P.E.Hart, *Pattern Classification and Scene Analysis.* New York: John Wiley and Sons, Inc., 1978.