ABSTRACT

HARDWARE IMPLEMENTATION OF DIGITAL FILTERS

Saleem G. Zoughbi

This major technical report attempts to put together methods, approaches and techniques which can be used to implement recursive and non-recursive digital filters. In other words, it is the analysis and design of dedicated digital hardware that can be used to perform real-time filtering tasks like the many required in communication systems.

Chapter I introduces the digital filter, its operation and its realizations. It starts with basic definitions of the digital filter, its operation, then the different methods of realization are described.

In Chapter II, the different hardware implementations are described. These implementations include the conventional and combinatorial approaches. Other implementations that use counters, RAMs and residue number systems are described.

Chapter III describes problems, variations and other associated aspects of the approaches used in the hardware implementation of filters. Basic considerations of digital signal processor hardware architecture are described also.

Chapter IV, which concludes the report, gives a summary and an evaluation of the different hardware implementation approaches described in the report.

ACKNOWLEDGEMENT

I wish to express my gratitude to Professor A.Antoniou for suggesting numerous improvements and guiding me through writing this report.

I wish to thank Miss Margaret Stredder for typing the final manuscript. I also wish to thank Miss Luna Shomali and Miss Amal Shomali for helping in typing the first draft. Last, but not least, I wish to thank my wife Hiyam for helping and understanding.

TABLE OF CONTENTS

PAGE

LIST OF FIGURES

FIGURE PAGE

FIGURE PAGE

LIST OF TABLES

NOTATIONS

a_i	The ith coefficient of a filter (for input data
b_i	The ith coefficient of a filter (for delayed output in a recursive filter)
B	The word-length in bits
B_a	The word-length of a_i
B_x	The word-length of the input data word
B_t	The total word-length for a shift register in the counting implementation, $= B_a + B_x - 1$
b_{jk}	The kth bit of the jth coefficient
$d_i, e_i, f_i, F_{i,j}, g_i, h_i$	Logic functions of i,j
F_t	The tth Fermat number
$F(p)$	A Galois field of order p
$H(z)$	The transfer function in the z-domain
m_i	The ith modulo in a residue number system
m_i^{-1}	The multiplicative inverse of m_i
$\overline{m}_i$	The dynamic range of the modulo m_i in a residue number system
$\overline{m}_i^{-1}$	The multiplicative inverse of $\overline{m}_i$
R	The response of a filter
T	The sampling period
$X_n, X(nT)$	The nth sample of input data
$\overline{X}$	The Discrete Fourier transform
$X_i(n), \lvert X_n \rvert_{m_i}$	The input data modulo m_i in a residue number system

$X(z)$	The Z-transform of the input
$Y_i(n)$, $\lvert Y_n \rvert_{m_i}$	The output modulo m_i in a residue number system
$Y(z)$	The Z-transform of the output
$Y_{i,n}$	The ith partial result of the nth output sample
α_{ij}	
β_{ij}	
γ_{ij}	
δ_{ij}	Binary bits for discrete numbers in binary representation
μ_{ij}	
ν_{ij}	
ρ_{ij}	

CHAPTER I
BASIC CONCEPTS

1.1 INTRODUCTION

Filtering is a process by which the frequency spectrum of a signal can be reshaped, modified or manipulated according to some desired specification. It may include amplifying, attenuating, rejecting or isolating frequency components. Such a process can be used to eliminate signal contamination such as noise, to remove signal distortion brought about by an imperfect transmission channel, to separate two or more distinct signals and many other applications.

The digital filter is a digital system that is used to filter discrete time signals. It is implemented by means of software (computer programs) or by means of dedicated hardware. The hardware implementation is very useful for real-time data, although real-time and non-real-time data can be filtered by both a hardware or a software-implemented digital filter.

Digital filters have tremendous applications in digital signal processing. These include digital communications, acoustics, biomedical engineering, radar systems, moving-target indicators, and so forth. The development in digital-filter hardware is progressing rapidly. This will lead to the production of efficient and low-cost digital filters that are suitable for a variety of applications.

1.2 THE DIGITAL FILTER AS A SYSTEM

A digital filter can be viewed as a system. The input is a discrete time quantized function which would excite the filter to produce a discrete time output. Figure 1.1 depicts such a representation. The response is related to the excitation by some rule of correspondence. If X(nT) and Y(nT) are the excitation and the response, respectively, then the filter can be characterized by

$$Y(nT) = R\ X(nT) \qquad (1.1)$$

where

R is an operator.

If the internal parameters of the filter do not change with time, the digital filter is said to be time-invarient. Usually, the response of a filter depends on the current and some previous values of the excitation. Such filters are said to be causal. On the other hand, filters in which the response depends on future values of the excitation, are said to be noncausal.

If the response of a digital filter is a linear function of the excitation, the filter is linear. For example, a linear digital filter would have its response doubled if

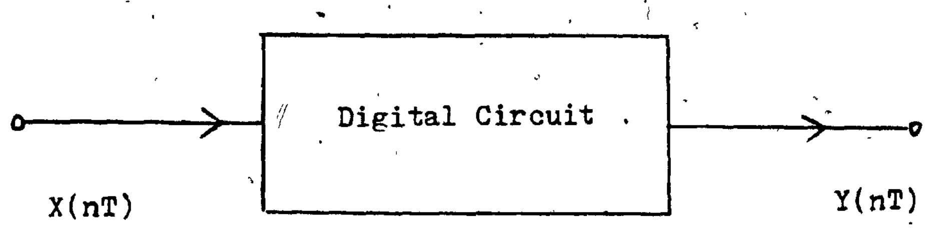


Figure 1.1 The digital filter as a system.

its excitation is doubled. In addition, in a linear filter, the response to a sum of different excitations is the sum of the individual responses.

Digital filters are of two basic types: recursive and non-recursive. A non-recursive digital filter is one in which the response depends only on the values of the excitation, that is

$$Y(nT) = f\{\ldots, X(nT-T), X(nT), X(nT+T) \ldots\}$$

Assuming the filter to be linear and time-invarient, Y(nT) can be expressed as

$$Y(nT) = \sum_{i=-\infty}^{\infty} a_i X(nT-iT)$$

where

a_i represent constants.

Furthermore, if the filter is causal

$$a_{-1} = a_{-2} = \ldots = a_{-\infty} = 0$$

Hence

$$Y(nT) = \sum_{i=o}^{\infty} a_i X(nT-iT)$$

If $X(nT) = 0$ for $n < 0$ and $a_i = 0$ for $i > N$, then

$$Y(nT) = \sum_{i=o}^{N} a_i X(nT-iT) \qquad (1.2)$$

Therefore, any linear, time-invarient, causal, non-recursive filter can be represented by an N^{th} - order difference equation. N is the order of the filter.

The recursive digital filter, on the other hand, has a response which depends on current and previous values of the excitation and the previous values of the responses. So, for a linear, time-invarient causal filter

$$Y(nT) = \sum_{i=o}^{N} a_i \; X(nT-iT) - \sum_{i=1}^{N} b_i \; Y(nT-iT) \qquad (1.3)$$

1.3 DIGITAL FILTER REALIZATIONS

There are three types of elements involved in the digital filter realization: the unit delay, the adder and the multiplier. The proper inter-connection of these elements would produce a network which would realize the digital filter.

It is convenient and useful to analyze the digital filter characteristics by examining the Z-transform of its response. Taking the one-sided Z-transform of Equation (1.3), we obtain

$$Z[Y(nT)] = Z[\sum_{i=o}^{N} a_i \; X(nT-iT)] - Z[\sum_{i=1}^{N} b_i \; Y(nT-iT)]$$

or

$$Y(Z) = \frac{\sum_{i=o}^{N} a_i \, Z^{-i}}{1 + \sum_{i=1}^{N} b_i \, Z^{-i}} X(Z) \qquad (1.4)$$

where

$$Y(Z) = Z[Y(nT)]$$

$$X(Z) = Z[X(nT)]$$

Let us define N(z), D(z), D'(z), H(z) as follows:

$$N(z) = \sum_{i=o}^{N} a_i \, Z^{-i} \qquad (1.5)$$

$$D'(z) = \sum_{i=1}^{N} b_i \, Z^{-i} \qquad (1.6)$$

$$D(z) = 1 + D'(z) \qquad (1.7)$$

$$H(z) = \frac{N(z)}{D(z)} \qquad (1.8)$$

With these definitions, Eq. (1.4) reduces to

$$Y(z) = H(z)\ X(z)$$

where

H(z) is the transfer function of the filter.

There are many methods for realizing a digital filter. The basic and common principle to all of them,to break down the filter transfer function, H(Z), into blocks and to re-arrange them into different configurations. Four configurations for realizing digital filters are described here [1].

1.3.1 Direct Realization

With $H(Z)$ defined as in Eq. (1.8), we can write $Y(Z)$ as

$$Y(Z) = U_1(z) + U_2(z) \tag{1.9}$$

where

$$U_1(Z) = N(Z)\ X(Z) \tag{1.10}$$

$$U_2(Z) = -D'(Z)\ Y(Z) \tag{1.11}$$

The realization can be obtained by realizing $N(Z)$ and $-D'(Z)$ separately, and then connecting them, as shown in Fig. 1.2. The realization of polynomial $N(Z)$ is easy and can be obtained in several ways. One possibility is depicted in Fig. 1.3. The same approach can be used to realize $-D'(Z)$. The direct realization of a second-order filter, $N = 2$, is shown in Fig. 1.4.

1.3.2 Direct Canonic Realization

If Eq. (1.4) is rewritten as

$$Y(Z) = N(Z)\ Y'(Z) \tag{1.12}$$

where

$$Y'(Z) = X(Z) - D'(Z)\ Y'(Z) \tag{1.13}$$

we can deduce a direct realization which has the structure shown in Fig. 1.5. Notice that the number of unit delays is N, which is the order of the filter. Because of this property the realization is said to be canonic with respect

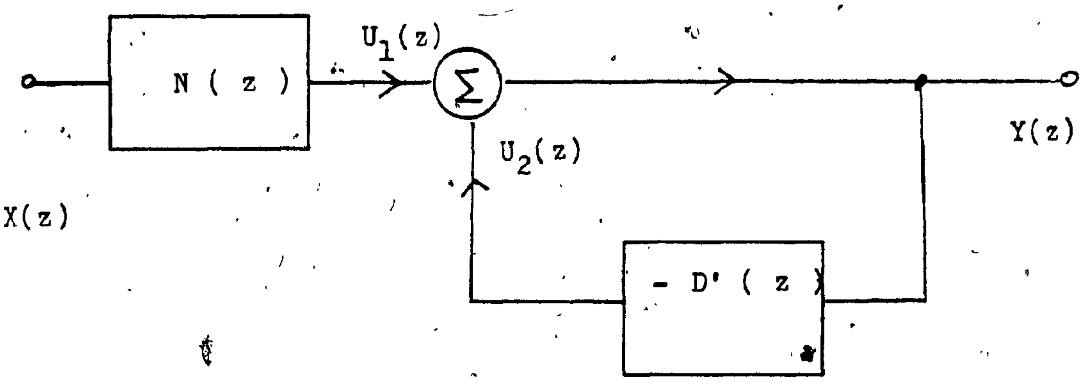


Figure 1.2 Direct realization of H(z).

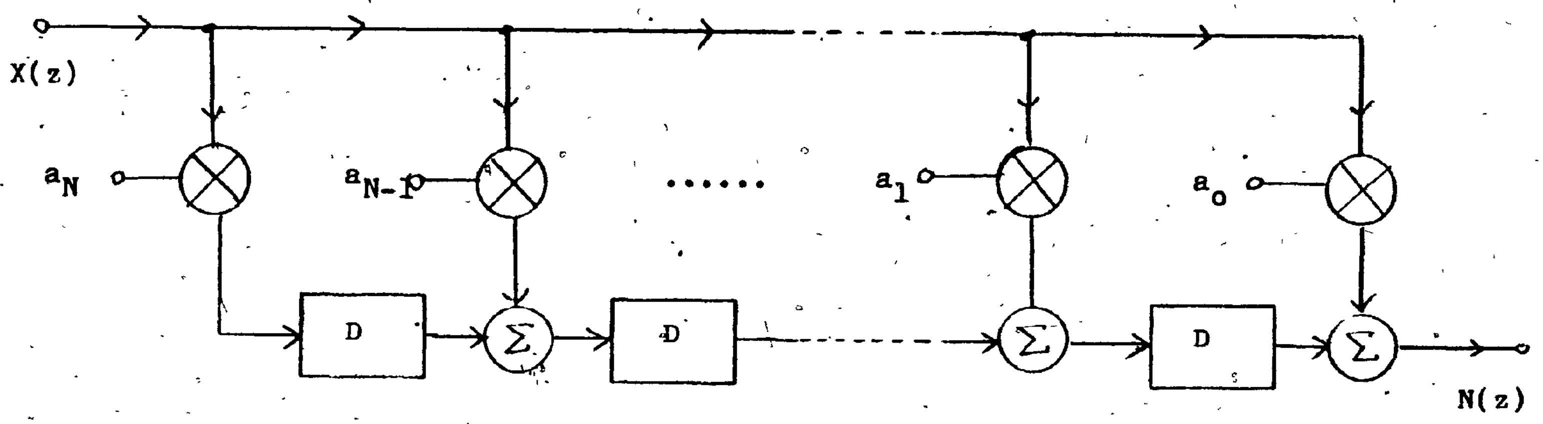


Figure 1.3 The realization of N(z).

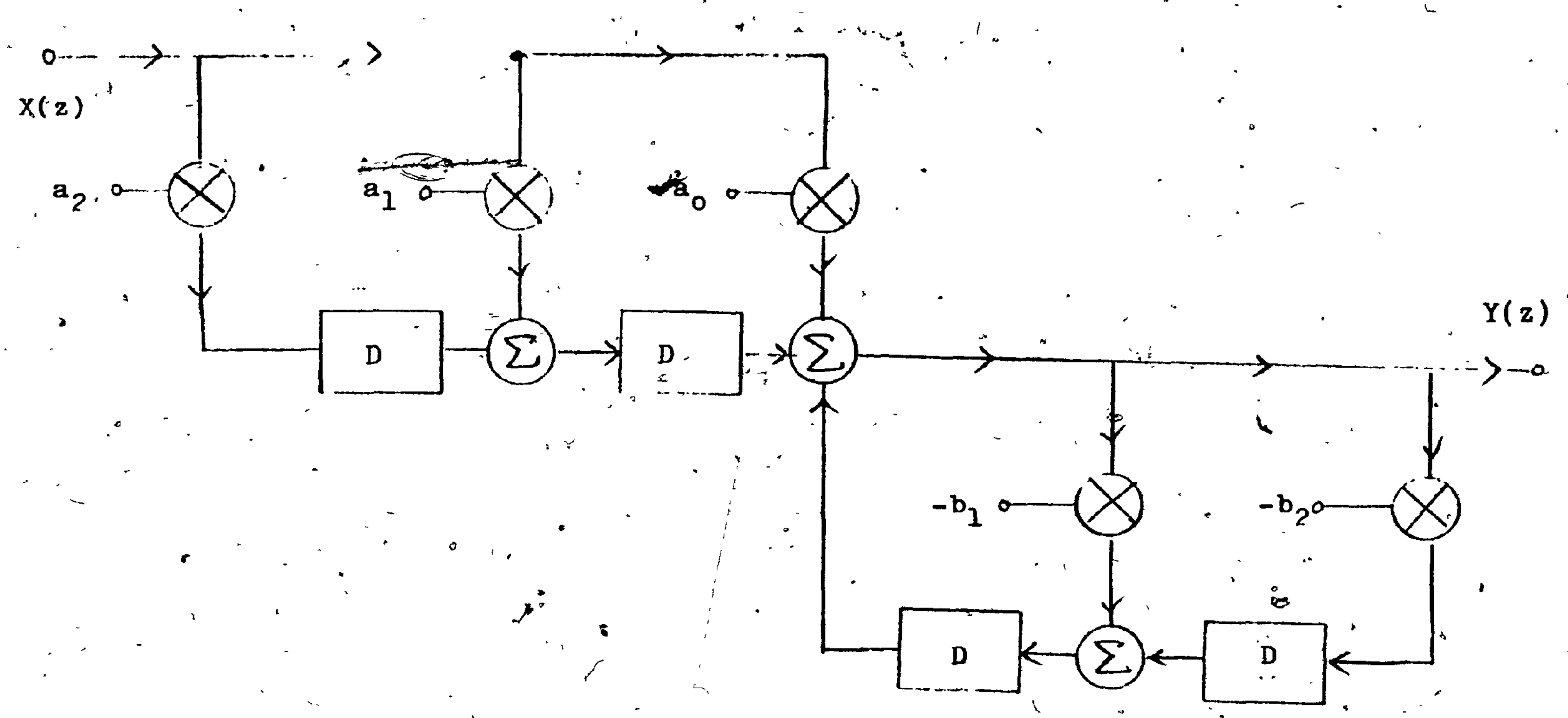


Figure 1.4 Direct realization of a second-order filter.

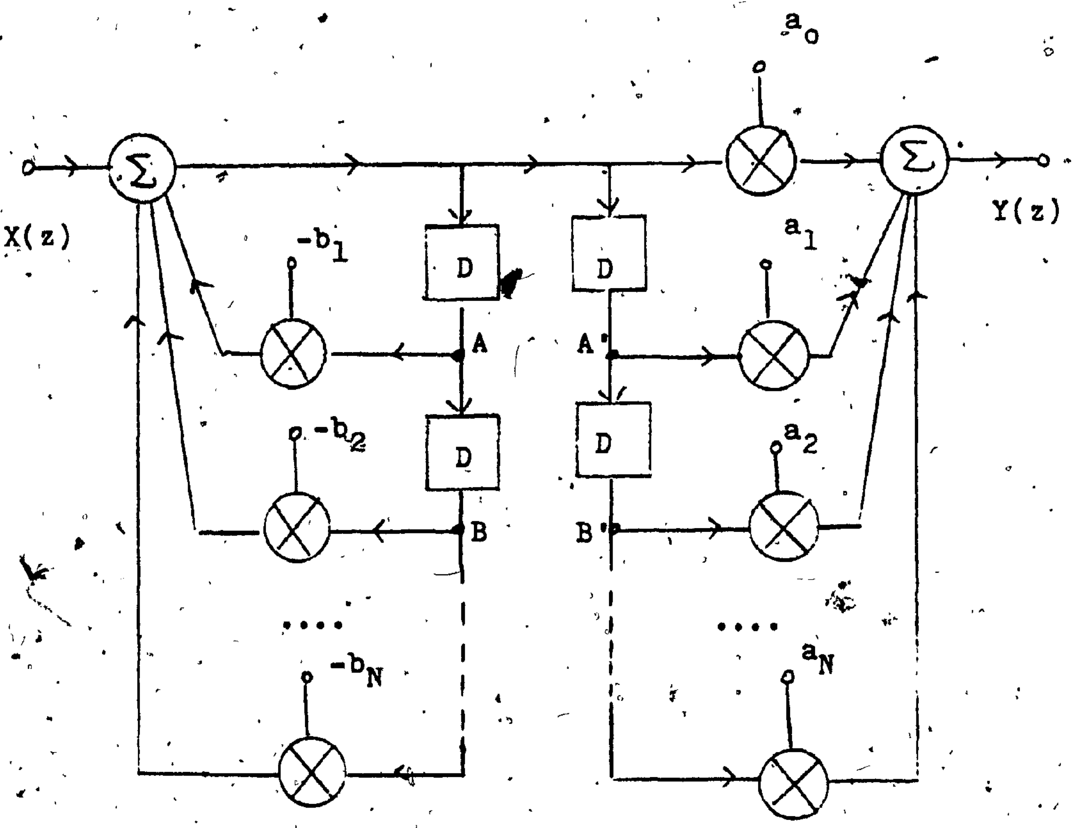


Figure 1.5 The Canonic realization of H(z).

Notice that points A & A' are the same point indeed. They can be combined together to eliminate the excess usage of delay units.

The same thing applies to points B & B',..etc.

to the number of unit delays.

1.3.3 Cascade Realization

The transfer function H(Z) defined in Equation (1.8) can be split into a product of second-order sections, i.e., we can rewrite Equation (1.8) as

$$H(Z) = \prod_{i=1}^{M} H_i(Z) \qquad (1.14)$$

where

$$H_i(Z) = \frac{a_{oi} + a_{1i}Z^{-1} + a_{2i}Z^{-2}}{1 + b_{1i}Z^{-1} + b_{2i}Z^{-2}} \qquad (1.15)$$

Each $H_i(Z)$ can be realized by using the 2nd - Order canonic structure of Figure 1.6. The cascade realization can be obtained by cascading the realizations of $H_i(Z)$ as in Figure 1.7.

1.3.4 Parallel Realization

In this approach, Equation (1.8) is rewritten as

$$H(Z) = \sum_{i=1}^{M} H_i(Z) \qquad (1.16)$$

where

$$H_i(Z) = \frac{a_{oi} + a_{1i}Z^{-1}}{1 + b_{1i}Z^{-1} + b_{2i}Z^{-2}} \qquad (1.17)$$

Each $H_i(Z)$ is realized as a separate block, as in the cascade realization. The blocks obtained are then connected

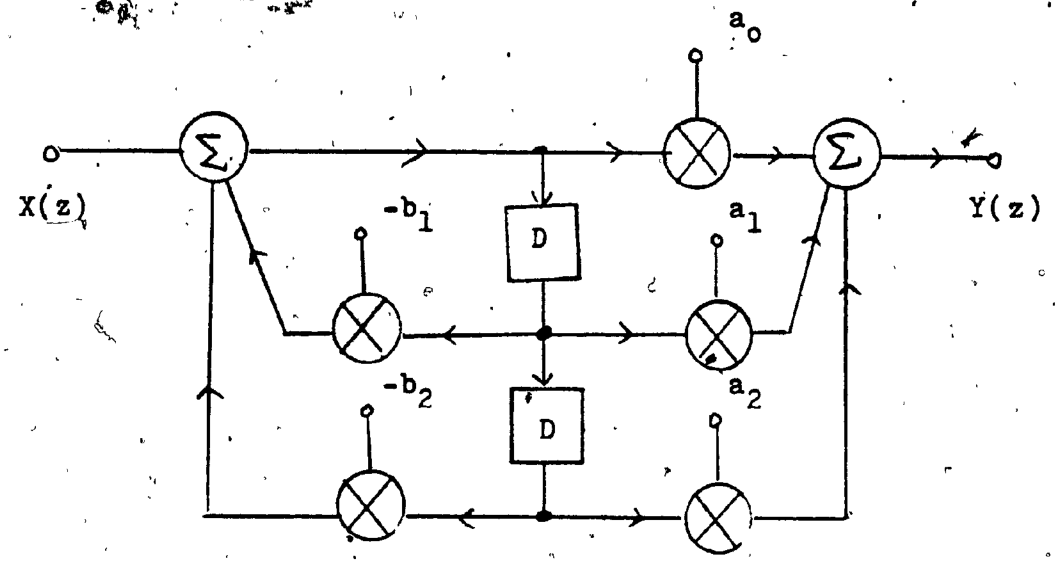


Figure 1.6 A canonic second-order section.

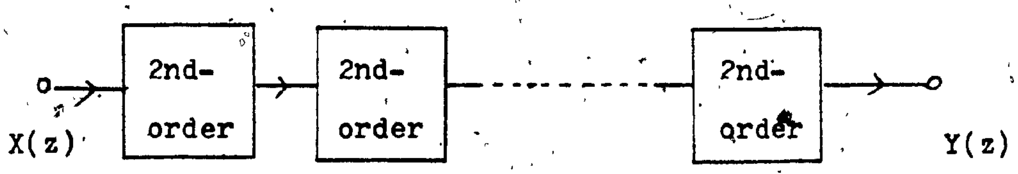


Figure 1.7 A cascade realization of H(z).

in parallel as in Figure 1.8.

1.4 DIGITAL FILTER IMPLEMENTATION

Certain arithmetic operations are required very often in digital signal processing. This includes bit operations which are the elementary operations dealt with in implementation.

Recalling the characteristic equations of the non-recursive digital filter (Equation (1.2)), and that of the recursive one (Equation (1.3)), the filter operation involved is a "multiply-and-add" one. Sometimes, the initial input and final output of the filter are continuous time analog signals. Hence, to be able to use a digital filter in such a case, another process is required, namely, sampling to obtain discrete data, and smoothing to recover the continuous output.

The "multiplication-and-addition" process is the thing that identifies the different methods of implementation. There are many types of implementations because of the different ways one can multiply and add binary numbers. One possibility is to use a multiplier and an accumulator. One other way is to precompute all possible products of the coefficients of the filter and all possible inputs, and store them in a table, then use the input to look into this table and find the partial products and accumulate them.

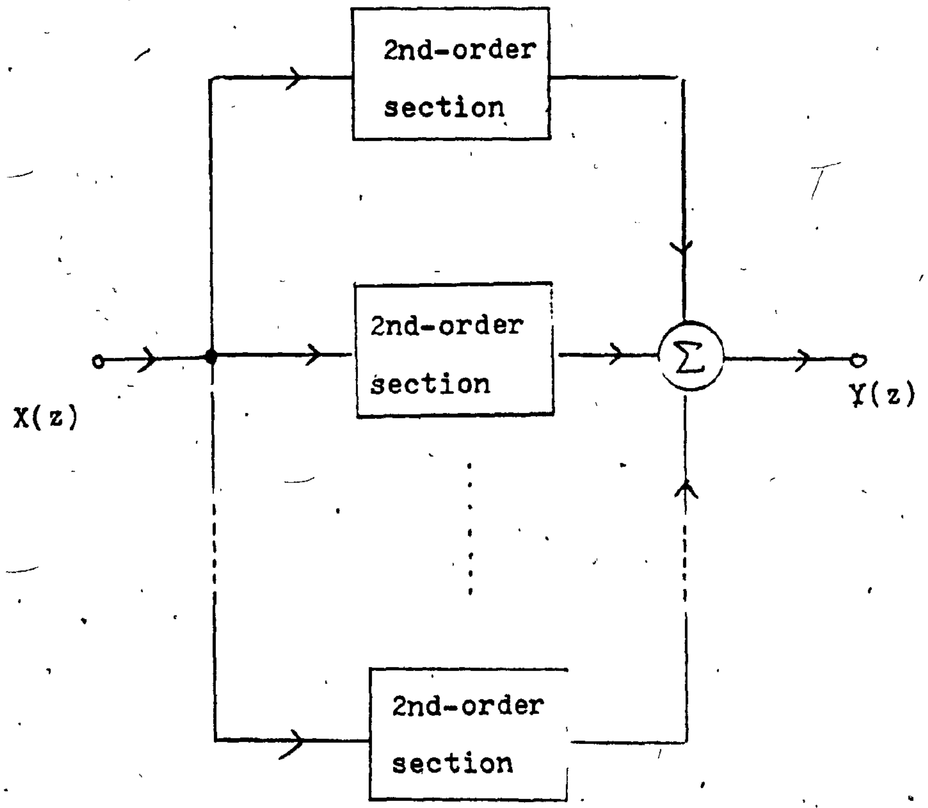


Figure 1.8 Parallel realization of H(z).

Many other ways are suggested, and described in Chapter II. However, these methods require one or more of the following digital operations:

(a) Addition or accumulation: This is the process of adding a computed value to a register. This is done by an accumulator/register, using full adders and combinational circuitry.

(b) Multiplication by 2^n: This is an easy operation realized by a shift register. To multiply a register value by 2^n, we simply shift the n bits toward the most significant bit in the register. Similarly, to divide by 2^n the bits are shifted towards the least significant bit in the register.

(c) Multiplication of two binary numbers: There are many algorithms to multiply two binary numbers. The most common one is the serial multiplication. Let $X_1X_2 \ldots X_N$ and $Y_1Y_2 \ldots Y_N$ be two N-bit binary numbers, (i.e., $X_i, Y_i \epsilon [1,0]$), then $Z_1Z_2 \ldots Z_N$ can be obtained as follows [1]:

(i) Initialize an accumulator (a shift register of 2^n bits.)

(ii) Form A_{ij} by gating (AND) Y_i and X_i for j=1,2, ..., N, and accumulate the result in the accumulator/shift register.

(iii) Shift the contents of the shift register one bit towards the most significant bit (A_{i1}), i.e., multiplying by 2.

(iv) Add the contents of the shift register to the accumulator.

(v) Repeat steps (ii), (iii), (iv) for i=1,2, ..., N.

The final result in the accumulator is the product $Z_1Z_2 \ldots Z_{2N}$. Another approach for the multiplication of two binary numbers is to use the serial magnitude multiplier. This multiplier receives the multiplicand bit by bit, serially, and the N bits of the multiplier in parallel and produces the product serially, bit by bit. It can be realized by delay flip-flops, gates and full adders.

(d) Residue arithmetic encoding and decoding: Another pair of digital operations that is sometimes used in the implementation of digital filters, is residue arithmetic encoding and decoding. If, for example, $X_1X_2 \ldots X_n$ is a binary number, the encoding process is to find M binary numbers, each represented by K bits as

$$R = \{X_{11}X_{12} \;\ldots\; X_{1K};\; X_{21}X_{22} \;\ldots\; X_{2K};\; \ldots;\; X_{M1} \;\ldots\; X_{MK}\} \qquad (1.18)$$

where those M binary numbers are the original binary number $X_1\ X_2\ \ldots\ X_n$ modulo p_i, and

p_i's is a set of M relatively prime integers.

The decoding process is the reverse of this operation, i.e., given R as in Equation (1.18), $X_1X_2\ \ldots\ X_n$ is obtained.

(e) Counting: The "multiply-and-add" operation can be implemented by adding all the possible products of all the bits of the data and the coefficients. These bit products are gated bits, hence by counting the number of 1's obtained at the gates, and shifting and accumulating, we achieve the filter operation. This requires binary counters which are implemented by using flip-flops and gates.

Any implementation method uses two or more of the above operations. The choice of operations affects the efficiency, accuracy and speed of the implemented digital filter.

CHAPTER II

HARDWARE IMPLEMENTATION OF DIGITAL FILTERS

2.1 INTRODUCTION

Hardware implementations of digital filters have different characteristics. These characteristics are determined by the implementation approaches used. These approaches use mostly logic circuitry that are basically combinatorial and use shift registers, ROMs, RAMs, counters, encoders and decoders. The basic approaches discussed in this Chapter are widely different from each other, and hence their hardware characteristics differ with respect to speed, accuracy, reliability and other aspects. Six different approaches are discussed here, and the basic concepts are analyzed and compared.

2.2 CONVENTIONAL IMPLEMENTATIONS

One of the standard approaches towards the implementation of digital filters is to replace the multipliers and adders in the filter realization by arithmetic units that would multiply and add. A specific multiplier unit was proposed by Jackson, Kaiser and McDonald [2] as early as 1968. This multiplier implements the filter using serial arithmetic, and performs addition and subtraction, as well. The filter can be constructed from a small set of relatively simple digital circuits, primarily shift registers and adders. This configuration is highly modular in form, and is well suited to large-scale integration. In fact, such a filter can be easily multiplexed to process a number of distinct signals.

2.2.1 The Arithmetic Unit

There are three operations involved in the filter operation: delay, addition and multiplication. Each one of these operations can be implemented by a simple digital circuit. A delay is achieved by a delay flip-flop, and the addition is done by using an accumulator of a set of full adders. The multiplication, on the other hand, is not easy to implement by a simple digital circuit. However, if the multiplication is examined on the bit level, it turns out to be a simple process of gating. This is because the product of two bits is the same as their gated value.

Consider the multiplication of two B-bit words, x and a, such that

$$x = \sum_{i=0}^{B} x_i(2^i) = X_B X_{B-1} \;\; \dots \;\; X_1 X_0$$

$$a = \sum_{i=0}^{B} a_i(2)^i = A_B A_{B-1} \;\; \dots \;\; A_1 A_0 \qquad (2.1)$$

where

$$X_i,\ A_i\ \varepsilon\ \{0,1\}.$$

If y is the product of x and a, then

$$y = xa = \sum_{i=0}^{B} X_i 2^i \sum_{i=0}^{B} A_i 2^i$$

$$= \sum_{i=0}^{B} 2^i \left(\sum_{j=0}^{B} X_i A_j 2^j \right) \qquad (2.2)$$

This can be rewritten as

$$y = (X_B X_{B-1},\ \ldots,\ X_1 X_0)(A_B A_{B-1},\ \ldots,\ A_1 A_0)$$

$$= S_{2B}\ S_{2B-1},\ \ldots,\ S_1 S_0 \qquad (2.3)$$

where $S_i\ \varepsilon\ \{0,1\}$. Bits S_i's can be obtained from the following algorithm.

$$\begin{array}{r}
X_B X_{B-1} \cdots\cdots\cdots X_2 X_1 X_0 \\
\underline{A_B A_{B-1} \cdots\cdots\cdots A_2 A_1 A_0} \\
q_{0,B} \cdots\cdots\cdots q_0, q_{0,0} \\
\underline{q_{1,B}\ q_{1,B-1} \cdots\cdots q_{1,1}\ q_{1,0}} \qquad\quad \\
q_{2,B}\ q_{2,B-1} \cdots\cdots\cdots q_{2,0} \qquad\qquad\qquad \\
\underline{q_{3,B}\ q_{3,B-1} \cdots\cdots\cdots\cdots q_{3,0}} \qquad\qquad\qquad\qquad
\end{array}$$

```
         ...     ...     ...
       ...     ...     ...
     ...     ...     ...
   ...     ...     ...
---------------------------------
```

$$S_{2B}S_{2B-1} \cdots\cdots\cdots\cdots\cdots\cdots\cdots S_2 S_1 S_0$$

Here, q_{ij} is given by

$$q_{i,j} = A_i X_j \tag{2.4}$$

for $i,j = 0,1, \ldots, B$

The partial products are used to compute the final product bits

$$S_i = Z_i \bmod 2 \tag{2.5}$$

where

$$Z_i = \sum_{j=0}^{i} q_{j,i-j} + C_{i-1}$$

and

$$C_i = Z_i - S_i$$

Notice that

$$C_0 = 0, \; S_0 = q_{0,0}$$

$$Z_0 = S_0$$

Notice that for every input, X_i, a process of gating takes place with every other bit of A_i. Adding these gated results, a partial sum is found. This partial sum would be controlled by a timing sequence that realizes the shifting process to account for the relative position of X_i and the i^{th} partial sum. Therefore, a 2B bit multiplier can be implemented by using gates, full adders and delay Flip-Flop. Figure 2.1 shows a serial multiplier. It is composed of multiplier bit sections which are gated full-adders. X_j is fed in to be gated with A_i, then the result is accumulated, i.e., added to a partial sum which was calculated in the previous step. This partial sum is gated with a timing sequence, r_{i+1} so as to be sure that the proper bit gating is added to the old partial sum, starting at the proper bit in this sum.

Negative numbers can be represented in the two's complement form or the signed magnitude form. A sign bit can be fed to a sign detector or multiplier. In fact, an exclusive-OR gate is what is needed. It takes two signs as input and produces their product sign. Timing is required to prevent the sign bit from being taken as a magnitude bit, thus resulting in an erroneous output. After the sign multiplication is done, and the final output is computed, the sign adjustment is trivial. The timing will cause the sign bit to be taken to the sign detector (the exclusive-OR gate). The timing sequence ensures that there is synchronization of this shift-

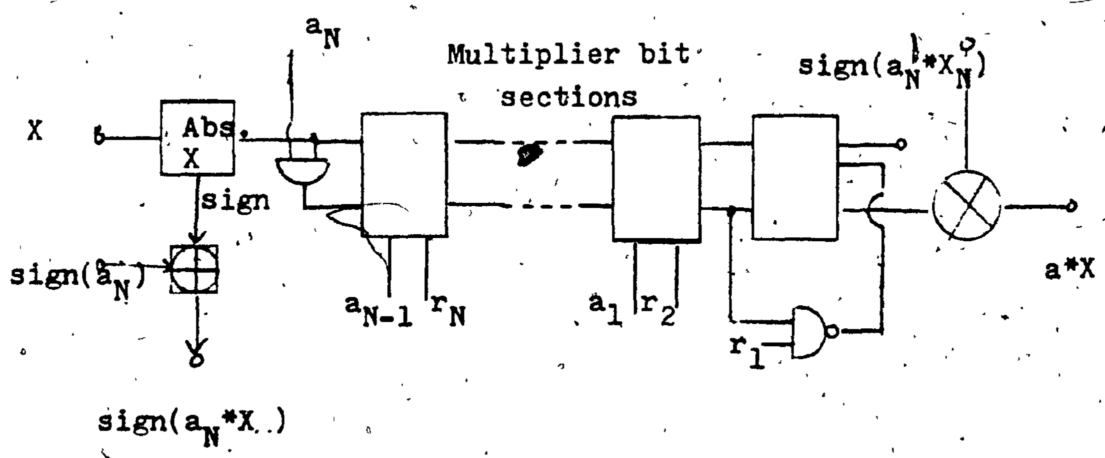


Figure 2.1(a) Serial multiplier for signed magnitude $X*a$ where $a = a_{N-1}a_{N-2}\dots a_1a_o$. a_N is the sign bit.

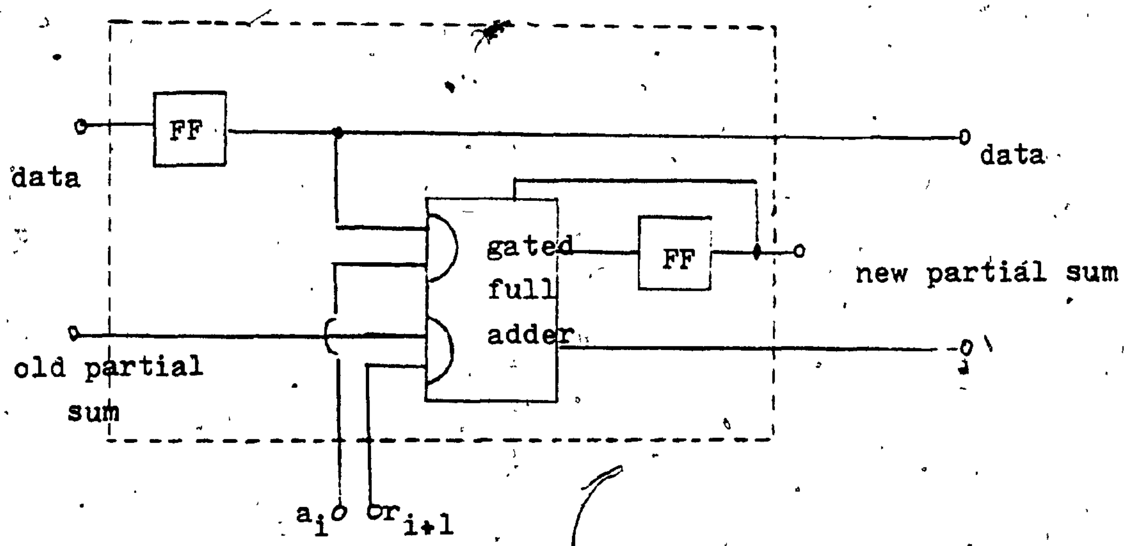


Figure 2.1(b) A multiplier bit section.

ing process, and the addition of this product to the old partial sum. Finally, there are two things to do: change the product sign, as mentioned earlier, and secondly, to truncate or round the final product to a word of the required number of bits.

2.2.2 Implementation of Digital Filters

Equations (1.2) - (1.3) can be expressed as

$$Y_n = \sum_{i=0}^{N} a_i X_{n-i} \qquad (2.6)$$

and

$$Y_n = \sum_{i=0}^{N} a_i X_{n-i} - \sum_{i=1}^{N} b_i Y_{n-i} \qquad (2.7)$$

where

$X_{n-i} \equiv X(nT-iT)$

$Y_{n-i} \equiv Y(nT-iT)$

for non-recursive filters.

The z-transform gives

$$Y(z) = \sum_{i=0}^{N} a_i z^{-i} X(z) \qquad (2.8)$$

and proper scaling, Y(z) can be expressed as

$$Y(z) = K_0(1+a_1'z^{-1}+a_2'z^{-2} + \ldots + a_n'z^{-N})\ X(z)$$

$$= Y'(z)\ K_0$$

that is

$$Y'(z) = X(z)(1+\sum_{i=1}^{N} a_i'\ z^{-i}) \qquad (2.9)$$

Alternatively, by factorizing the polynomial $(1+\sum_{i=1}^{N} a_i'\ z^{-i})$ we obtain

$$Y'(z) = \sum_{i=1}^{N_o} (1+\alpha_{1i}\ z^{-1}+\alpha_{2i}\ z^{-2})\ X(z) \qquad (2.10)$$

where

$$N_o = \begin{cases} N/2 & N\text{:even} \\ (N+1)/2 & N\text{:odd} \end{cases}$$

Each second-order polynomial in Equation (2.10) can be realized by a second-order filter section. By proper scaling again, the input data can be scaled so as to satisfy the following condition:

$$-1 < x_n < 1 \qquad (2.11)$$

The complete filter can be realized by using N_o second-order sections in cascade.

Likewise, for the recursive filter, Equation (2.7) gives

$$Y(z) = \frac{\sum_{i=0}^{N} a_i' \, Z^{-i}}{1 + \sum_{i=1}^{N} b_i' \, Z^{-i}} X(z) \qquad (2.12)$$

This equation can be written as

$$Y'(z) = \prod_{i=1}^{N_o} \frac{(1 + \alpha_{1i}z^{-1} + \alpha_{2i}z^{-2})}{(1 + \beta_{1i}z^{-1} + \beta_{2i}z^{-2})} X(z) \qquad (2.13)$$

and hence, a recursive filter can be realized in cascaded form of second-order sections. Equation (1.12) can also be rewritten as

$$Y'(z) = G_o + \sum_{i=1}^{N_o} \frac{\gamma_{0i} + \gamma_{1i} \, z^{-1}}{(1 + \beta_{1i}z^{-1} + \beta_{2i}z^{-2})} \qquad (2.14)$$

This equation results in a parallel realization of second-order sections.

The implementation of the filter therefore can be accomplished by cascade or parallel connection of second-order simple filters characterized by

$$Y_i(z) = \frac{(1 + \alpha_{1i}z^{-1} + \alpha_{2i}z^{-2})}{(1 + \beta_{1i}z^{-1} + \beta_{2i}z^{-2})} X(z) \qquad (2.15)$$

To implement a simple section, we require four multipliers, two delay flip flops, three intermediate accumulators, and another accumulator to hold the final output. The implementation of Equation (2.15) is depicted in Fig. 2.2 . The cascade implementation of Equation (2.13) is slower than the parallel one of Equation (2.14). These implementations are shown in Figures 2.3 and 2.4 , respectively.

The multiplier, itself, causes a delay of B bits in going through to compute the product serially. This delay has to be deducted from the unit delay that is provided by the delay flip-flops. Also, on addition, the operation of checking the sign bit, a delay of B bits occurs which is the time needed to pass the absolute value of the input data and the arrival of the input sign bit. This has to be considered,too, in determining the delay time of the input data.

2.2.3 Multiplexing

If the input rate bit is far below the speed capability of the digital circuit, the digital filter can be multiplexed to utilize the circuit more efficiently. The filter can be multiplexed to operate on several input signals simultaneously, or to affect a number of different filters for a single input signal.

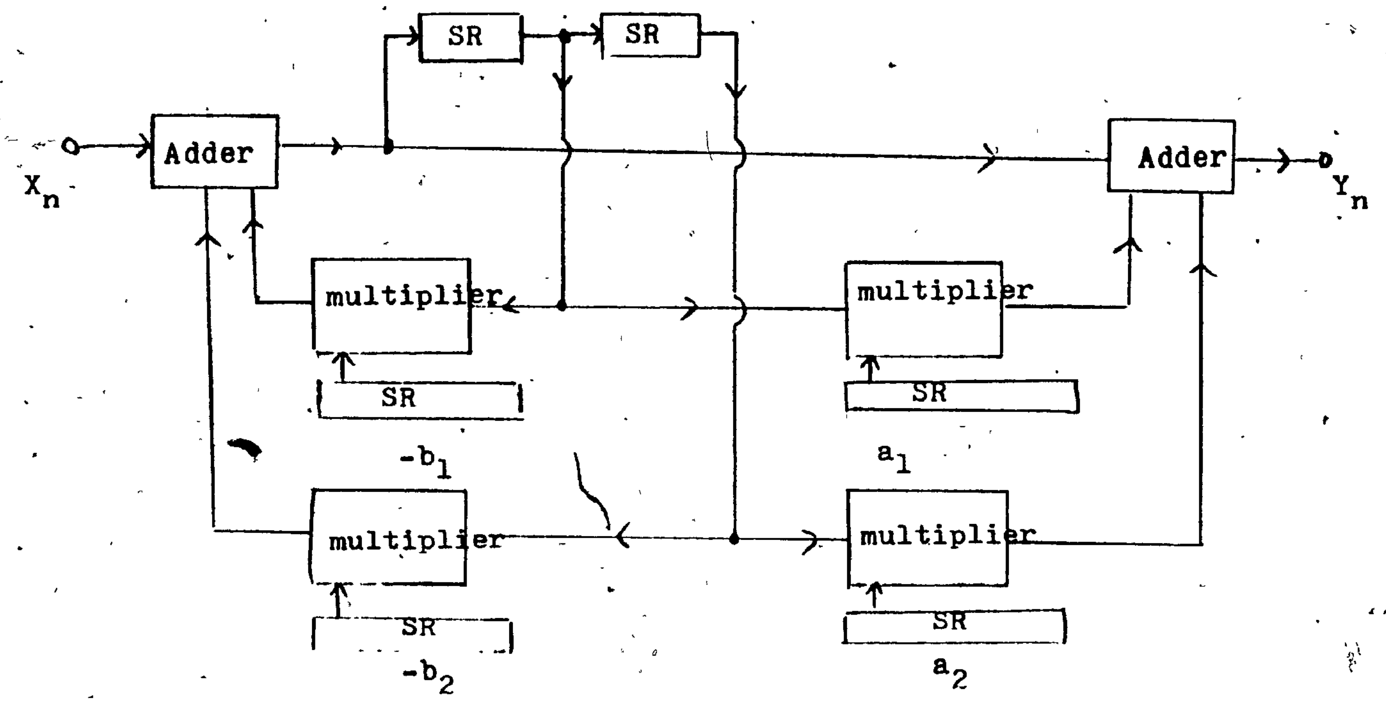


Figure 2.2 The basic conventional second-order section.

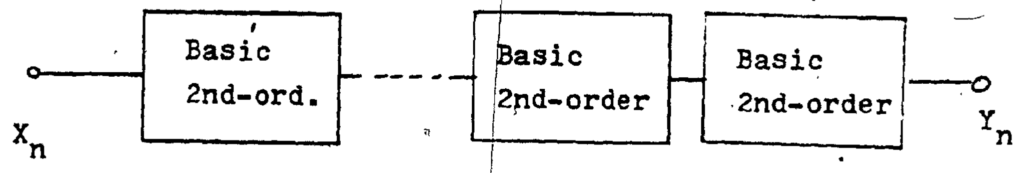


Figure 2.3 Cascade implementation using 2nd-order sections.

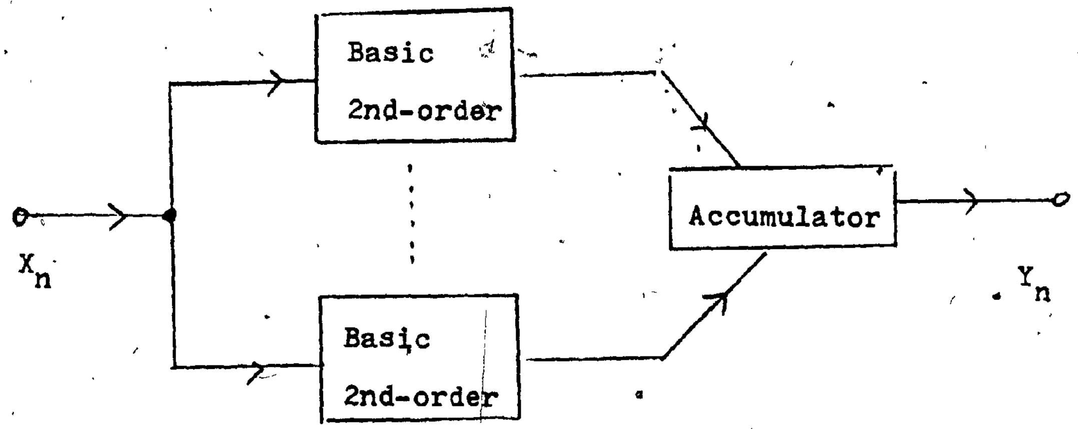


Figure 2.4 Parallel implementation using 2nd-order sections.

If there are M input signals, a multiplexer would cause the M input signals to be interleaved and fed serially into the arithmetic unit. The use of multiplexing is illustrated in Figure 2.5 . The bit rate in this scheme is increased by a factor of M. This means that the filter is operating M times faster.

The other possibility, is to use the same unit to compute, or participate in computing the outputs of several filters for given inputs. This implies that the filter section has different coefficients to use for every distinct input signal. These coefficients can be stored in ROMs. It is also possible to multiplex one arithmetic unit, so as to implement a higher order filter. This can be achieved by routing the output back to the input of the unit, and at the same time, changing the coefficients in the unit by reading a new set from the ROMs. Figure (2.6) illustrates this multiplexing scheme.

2.2.4 Multiplexed Hardware Structures

A more elaborate hardware structure for recursive digital filters was suggested by Gabel [3]. High speed operation and low hardware cost are achieved in this structure. Its operation is based upon a multiplier and a set of ROMs that store the coefficients. However, the control and the timing circuitry are different from that of the implementation discussed, so far.

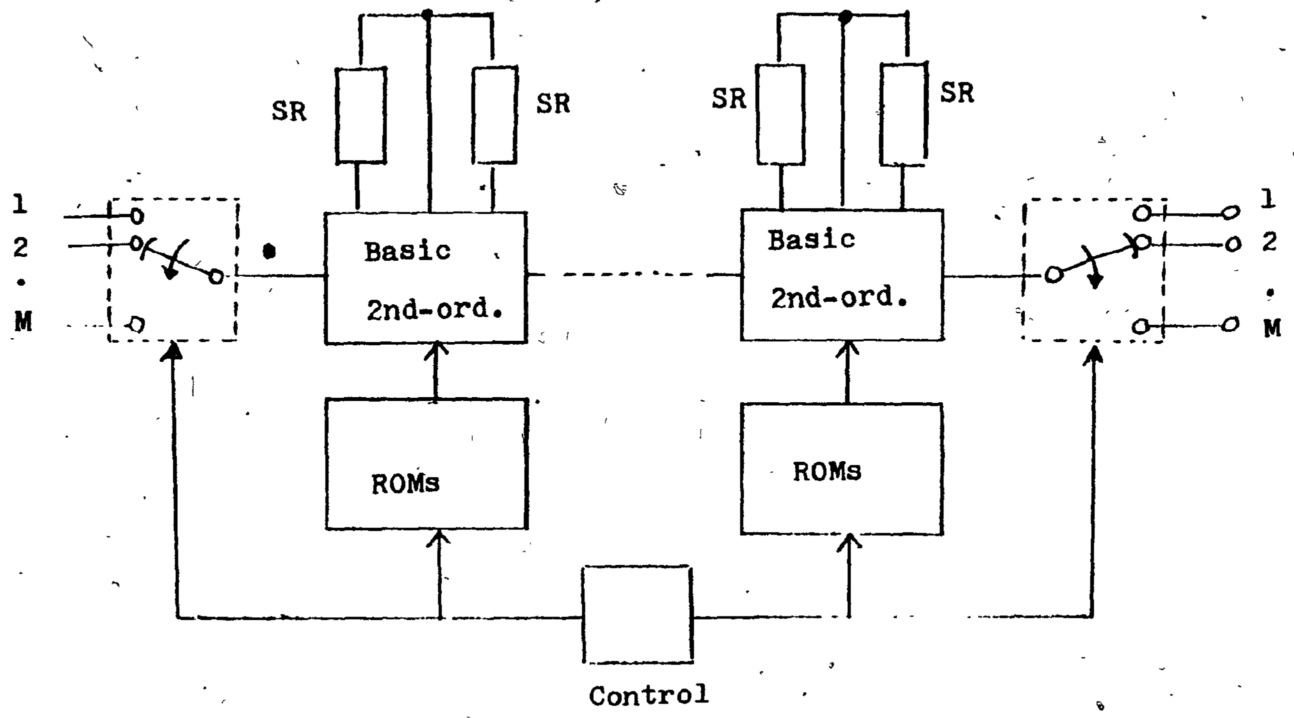


Figure 2.5 Type I multiplexing in conventional implementation.

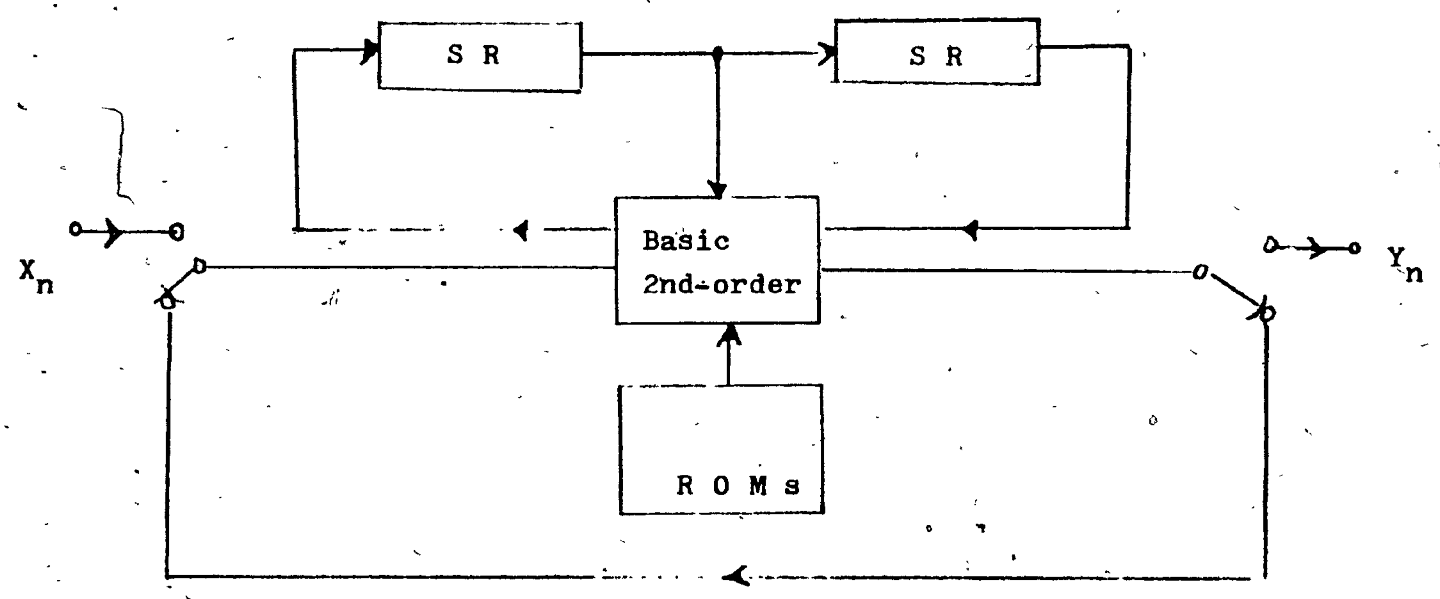


Figure 2.6 Type II multiplexing in conventional implementation.

Cascade or parallel implementation is possible. The cascade structure is shown in Figures(2.7) and (2.8). The input x(t) is digitized and fed to the multiplexer MUX1, together with the previous values of the input, X_{n-i}, where i=1,2, ..., N. X_n will be passed to the adder which accumulates in the AC register. This partial output is fed back to a hold register for the next Nyquist interval, then shifted to the shift register REG1 and then to REG2. Both of these registers are used to generate the old or delayed values of the partial outputs. Meanwhile, the counter would control the ROMs to provide the proper coefficients which are fed to the multiplier. The product is transmitted to the adder through a regulating multiplexer and the new partial output is accumulated again, in the AC register. Feeding the output back to MUX1 the cycle can be repeated over and over. The counter is all the control circuitry needed. It controls the data flow. Each count triggers a certain event in the system to happen. For example, an 8th - order filter is to be implemented. The control and the timing of the circuit must be of the form described in Table 2.1.

For a parallel implementation, the same hardware is used, but all sections now have the same input x_n and their outputs have to be accumulated, rather than to be fed back as inputs to the other stages. This structure is shown in Figure 2.9.

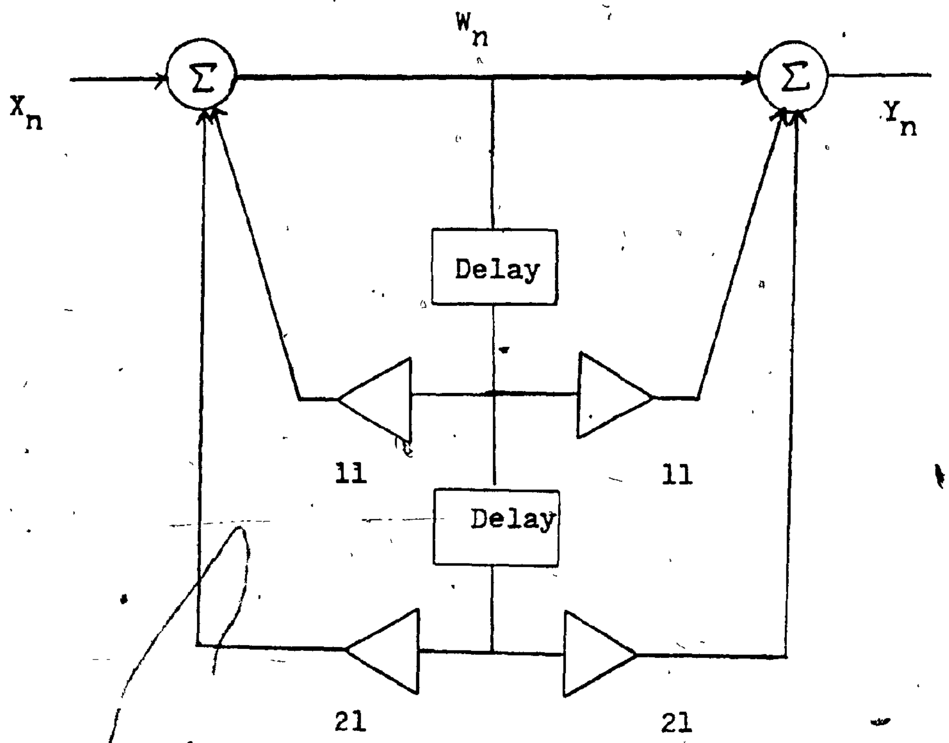


Figure 2.7 A Realization of a Second-Order Section.

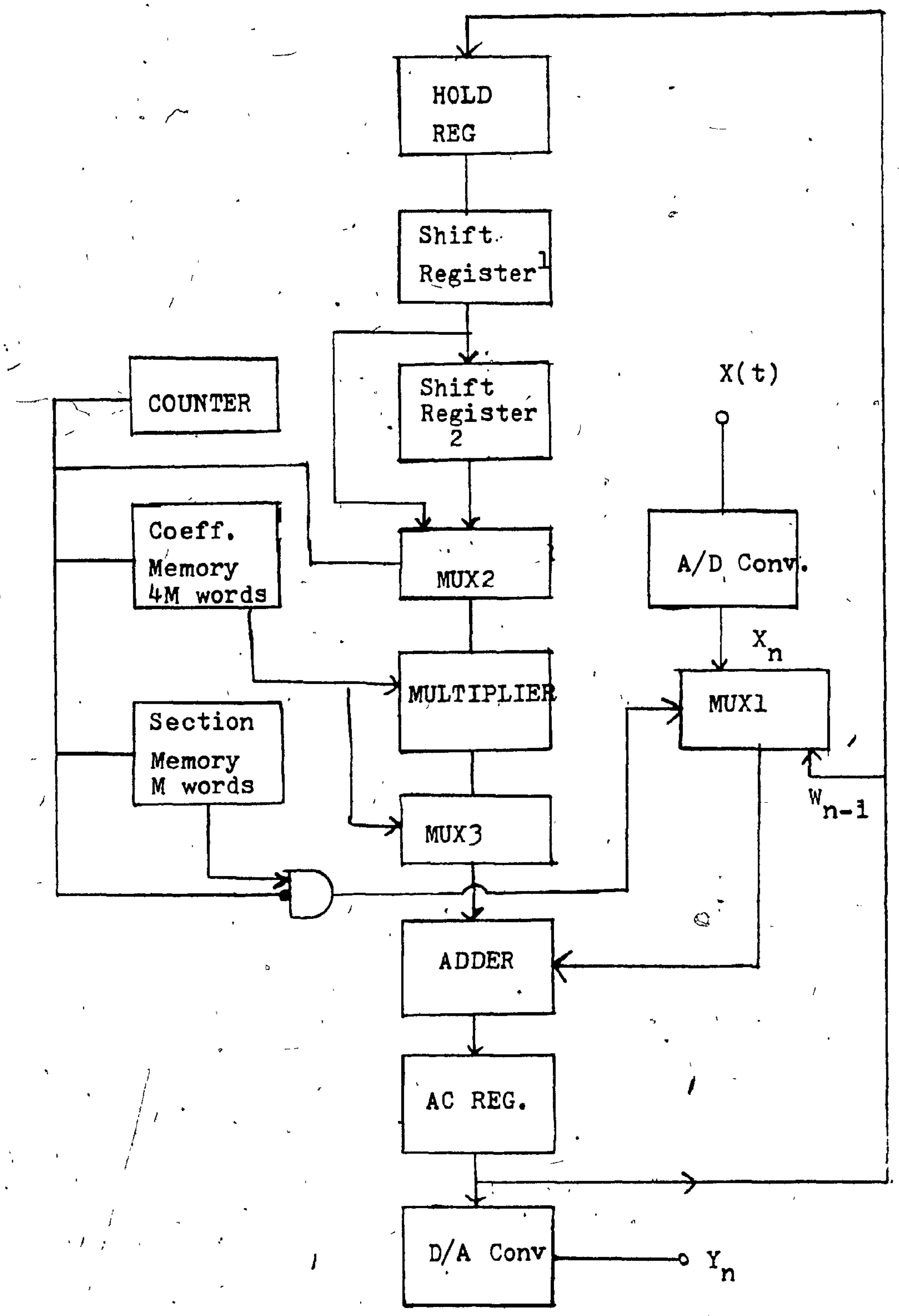


Figure 2.8 Gabel`s hardware structure for cascade sections

TABLE 2.1 CONTROL AND TIMING IN GABEL'S MULTIPLEXED FILTER

Count	Event
0000	Pass previous output to D/A
	Coefficient ROM selects $-\beta_{11}$
	MUX2 selects w_{n-1}
	MUX1 selects x_1
	AC enters sum
0001	Coefficient ROM selects $-\beta_{21}$
	MUX2 selects w_{n-2}
	MUX1 selects x_1
	AC accumulates sum
0010	Coefficient ROM selects $-\alpha_{11}$
	MUX2 selects w_{n-1}
	MUX1 selects x_2
	AC contents ($=w_n$) are passed to the hold register
	AC accumulates sum
0011	Coefficient ROM selects $-\alpha_{21}$
	MUX2 selects w_{n-2}
	MUX1 selects x_2
	SR shifts right, entering contents of hold register
	AC accumulates sum

TABLE 2.1 CONTROL AND TIMING IN GABEL'S MULTIPLEXED FILTER (continued)

Count	Event
0100	Coefficient ROM selects $-\beta_{12}$ MUX2 selects w_{n-1} MUX1 selects $2^s x_2$ where 2^s is a scaling factor used between sections, and x_2 is the output of the previous section AC accumulates sum ... ... and so for a total of 16 counts.

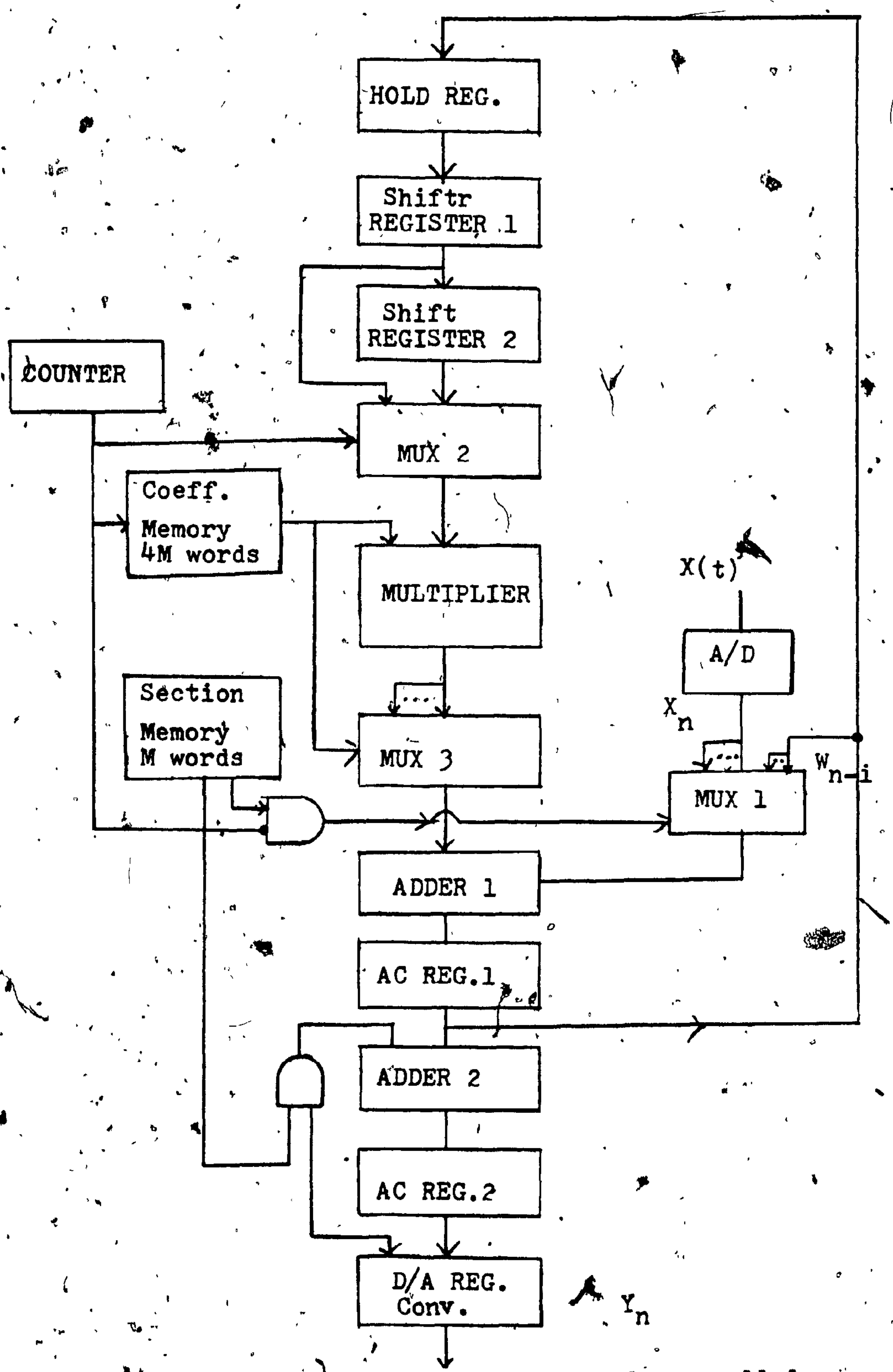


Figure 2.9 Gabel's hardware structure for parallel or cascade sections.

The cascade and parallel implementations offer advantages such as low-cost hardware, fast operation, simple control circuitry and easy programming. No bound is imposed on the number of bits used to represent external or internal variables. However, overflow should be studied carefully because it may drive the filter easily into incorrect computation.

2.3 COMBINATORIAL IMPLEMENTATIONS

Croisier et al, in a U.S. patent [3], and later discussed by Liu and Peled, [4,5,6] have introduced an implementation method which is completely different from that of Jackson, Kaiser and McDonald [2]. The sum of the products can be computed digitally, without multiplication at all. This is possible because if the multiplier is one bit only, then the multiplication is reduced to a process of gating. This partial result, the bit multiplication, can be precomputed for all possible combinations of input bits, thus these products can be stored in memory, such as a ROM. In fact, these input bits are nothing but each single bit of the input, and its corresponding bits in delayed-input values. Hence, the implementation reduces to a process of read, shift and add operations.

2.3.1 Non-Recursive Digital Filters

As was shown in Section 2.1.1, a non-recursive digital filter is represented by

$$Y_n = \sum_{i=0}^{N-1} a_i X_{n-i} \qquad (2.6)$$

Let the input samples be quantized and represented in the 2's complement form in words of B bits. Also, assume that

$$-1 < X_n < +1 \qquad (2.16)$$

This condition can be satisfied by proper scaling procedures. The n^{th} input sample is given by

$$X_n = \sum_{j=1}^{B-1} 2^{-j} x_{n,j} - x_{n,0} \qquad (2.17)$$

where

$$x_{n,j} \in \{0,1\}.$$

Substitution of Equation (2.17) into Equation (2.6) yields

$$Y_n = \sum_{i=0}^{N-1} \{a_i \{ \sum_{j=1}^{B-1} 2^{-j} x_{n-i,j} - x_{n-i,0} \}\} \qquad (2.18)$$

interchanging the order of summation,

$$Y_n = \sum_{j=1}^{B-1} 2^{-j} F_{n,j} - F_{n,0} \qquad (2.19)$$

where

$F_{i,j}$ is given by

$$F_{i,j} = \sum_{i=0}^{N-1} a_i \, x_{n-i,j} \qquad (2.20)$$

The above analysis leads to the following implementation. The jth bits of the input and its delayed versions are used to form a word. The resultant word is then used to address a ROM which contains all the possible values of F_i; the resulting value obtained from the ROM is a partial result that has to be added or accumulated in a shift register. The contents of this register have to be shifted towards the least significant bit, thus dividing by 2(or multiplying by 2^{-1}). This process is repeated N-1 times, where at the last bit, the addressed value has to be subtracted, rather than added, as Equation (2.19) indicates. Figure 2.10 depicts this implementation.

2.3.2 Memory Optimization

The ROM used is $2^N \times B$, and contains all the possible values of $F_{i,j}$. Notice that the size of the ROM increases rapidly as N increases, or as B is increased. This will make this implementation impractical for a filter of order 12 or more, or for large word lengths. Hence, high-

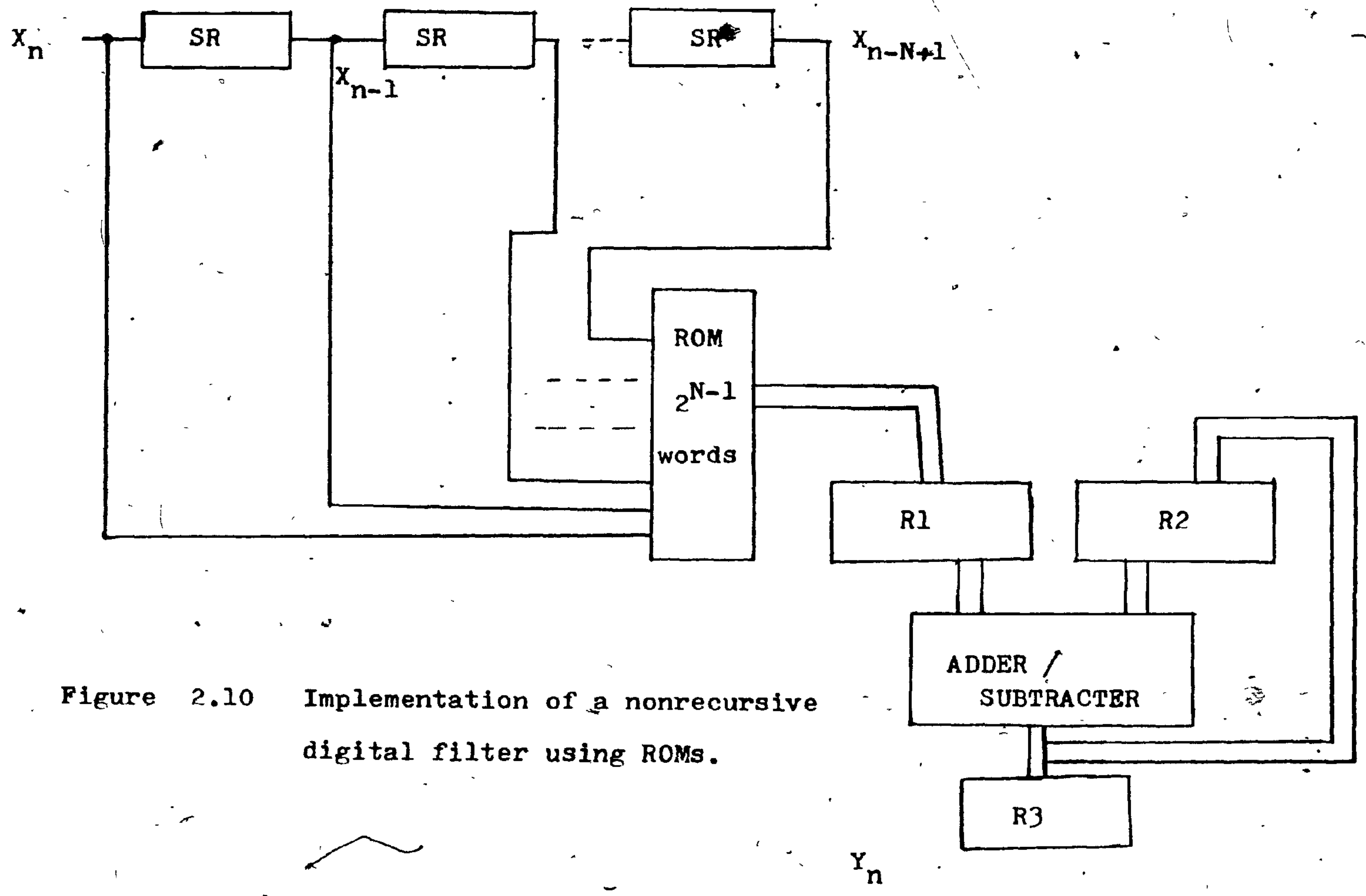


Figure 2.10 Implementation of a nonrecursive digital filter using ROMs.

order or high-precision filters are difficult to implement.

To overcome the above problems, the computation can be done through separate steps. The objective is to reduce as much as possible the size of memory needed. This can be done by decreasing the memory size and increasing the number of accumulators or by multiplexing one filter section.

The memory size can be decreased exponentionally by increasing linearly the number of accumulators used. This is due to the fact that if the filter is realized by using parallel low-order sections, the memory size will decrease. Consider a filter of order N-1. The transfer function can be expressed as a sum of M transfer functions, each of order K such that

$$N-1 = KM \tag{2.21}$$

Under these circumstances, Equation (2.6) becomes

$$Y_n = \sum_{i=0}^{K} a_i X_{n-i} + \sum_{i=K+1}^{2K} a_i X_{n-i} + \dots + \sum_{i=(M-1)K+1}^{MK} a_i X_{n-i} \tag{2.22}$$

or

$$Y_n = \sum_{i=1}^{M} V_i \tag{2.23}$$

where

V_i is the $i^{\underline{th}}$ partial sum

$$V_i = \sum_{j=(i-1)K+1}^{iK} a_j X_{n-j} \qquad (2.24)$$

There are M such small filters, and if those operate in parallel, we get M partial sums $(V_1, V_2, \ldots, V_n)$. Therefore, the final results are the sum of all those results. Such an implementation is depicted in Figure 2.11. Notice that the memory requirement is reduced from 2^{MK} to $M2^K$ since M ROMS of 2^K words are needed instead of one ROM of 2^{MK} words. This represents a significant reduction in the required ROM size. For example, if the filter is of order 15, N-1=15, then 2^{15}=32,798 words of storage are required if the filter is implemented directly with one ROM, while if it is implemented as three small $5^{\underline{th}}$ - order filters in parallel, the memory requirements would be three ROMS of 32 words each, amounting up to 96 words only, compared with 32,798 words. The modified implementation, however, requires two additional accumulators. This variation is shown in Figure 2.11.

One other way to decrease the memory requirement is by trading memory size with the speed of operation, by splitting the filter into M smaller filters of order K. The filter can be implemented by using one ROM of $M2^K$ words in the direct implementation, as in Figure 2.10. This

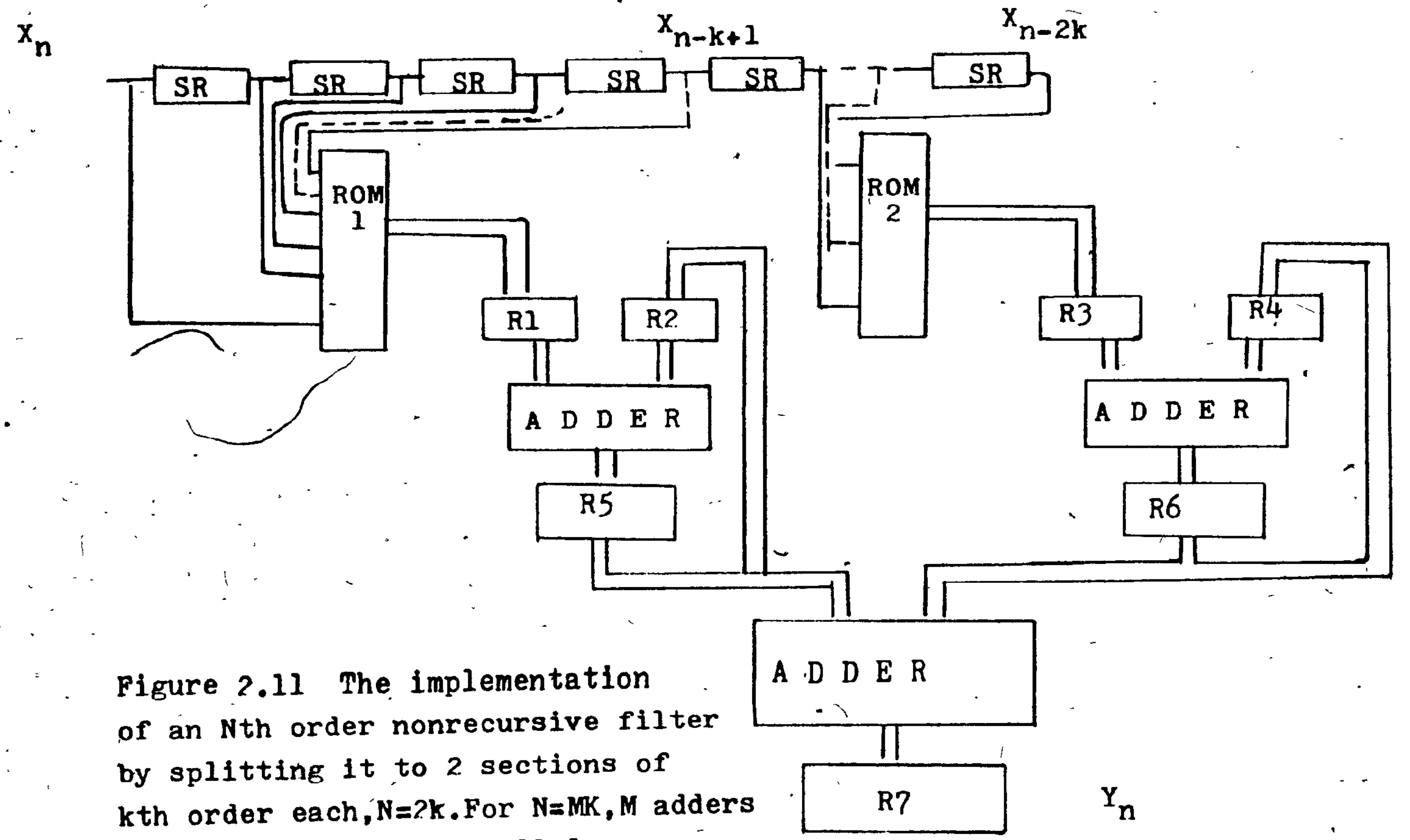


Figure 2.11 The implementation of an Nth order nonrecursive filter by splitting it to 2 sections of kth order each, N=2k. For N=MK, M adders have to operate in parallel.

would be the only small filter needed, because by multiplexing it for M channels, the partial results V_i, will be accumulated, and hence, the final result would be obtained after M sub-final additions, or BM clock cycles. Figure 2.12 illustrates this structure.

One important fact that can be used to implement a high order linear-phase non-recursive digital filter is the symmetry of the filter. This fact reduces the order of the filter by a factor of 2, because

$$a_i = a_{N-1-i} \qquad (2.25)$$

If this relation is used in Equation (2.6), the difference equation characterizing the filter would be reduced to

$$Y_n = \sum_{i=0}^{N-1} a_i X_{n-1} = \sum_{i=0}^{(N-3)/2} a_i (X_{n-i} + X_{n-N+1+i}) \qquad (2.26)$$

Furthermore, from Equation (2.17)

$$X_n = \sum_{j=0}^{B-1} X_{n,j}\, 2^{-j}$$

and hence, Equation (2.26) reduces to the following equation after the order of summation is interchanged:

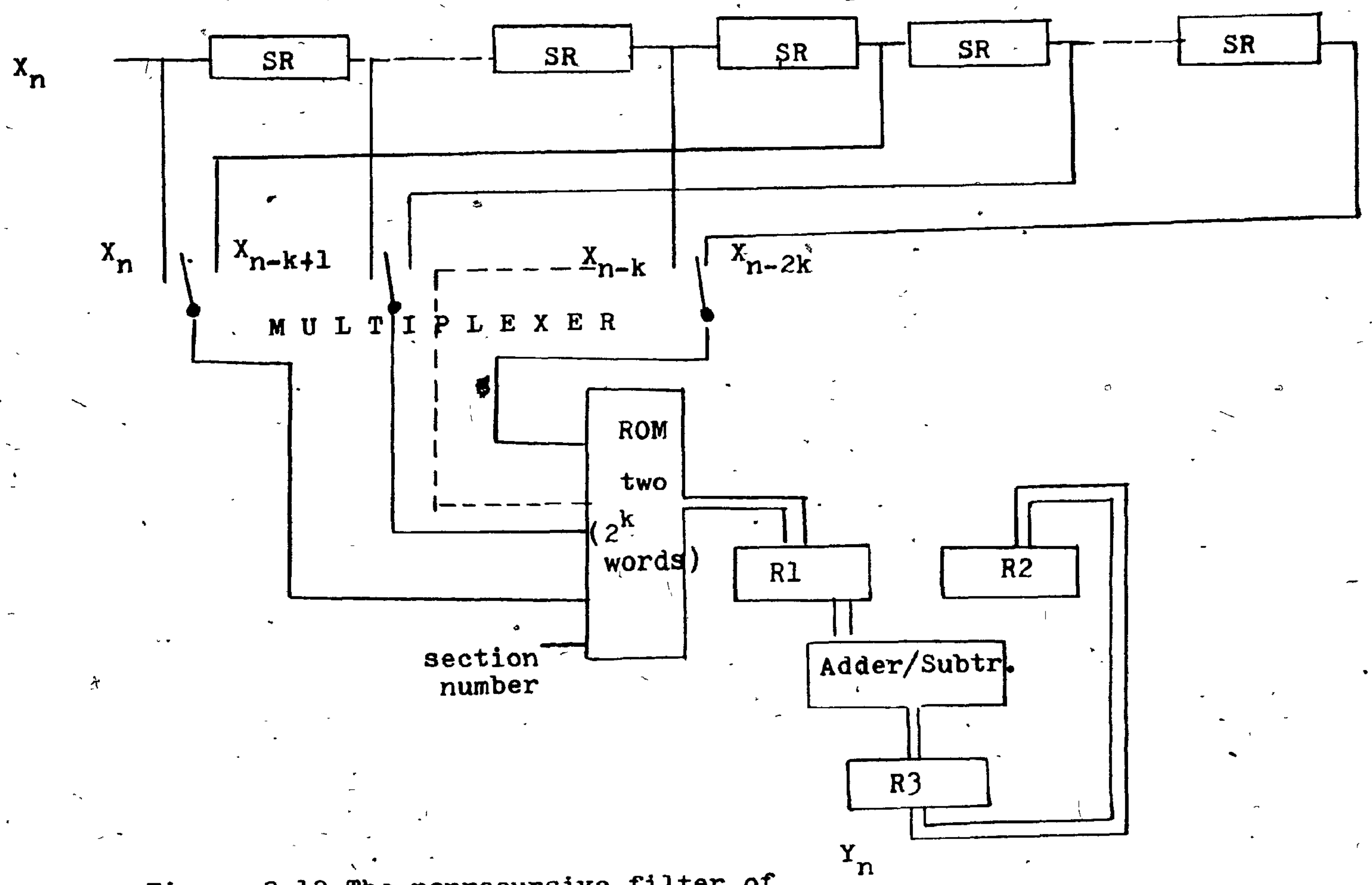


Figure 2.12 The nonrecursive filter of order N 2k in a serial fashion.

$$Y_n = \sum_{j=1}^{B-1} 2^{-j} \{ \sum_{i=0}^{(N-3)/2} a_i (X_{n-i,j} +$$

$$+ X_{n-N+1+i}) \} \qquad (2.27)$$

The sum of the two input bits would yield a zero or a one with a zero or a one carry to the next summation. (N-1)/2 full adders can be used to generate (N-1)/2 output bits which can be used to address a $2^{(N-1)/2} \times B$ ROM which contains all the possible combinations, assuming that the (N-1)/2 coefficients are unique. Figure 2.13 shows this implementation clearly for an (N-1)-order symmetric filter. The memory size is reduced from 2^{N-1} to $2^{(N-1)/2}$ words. For example, a 10th - order filter requires a ROM of $2^5=32$ words rather than $2^{10}=1{,}024$ words, but it requires five more full adders and five delays.

2.3.3 Recursive Digital Filters

As was shown in Section 2.2.2, a recursive digital filter has been represented by the equation

$$Y_n = \sum_{i=0}^{N-1} a_i \, X_{n-i} - \sum_{i=1}^{L} b_i \, Y_{n-i} \qquad (2.28)$$

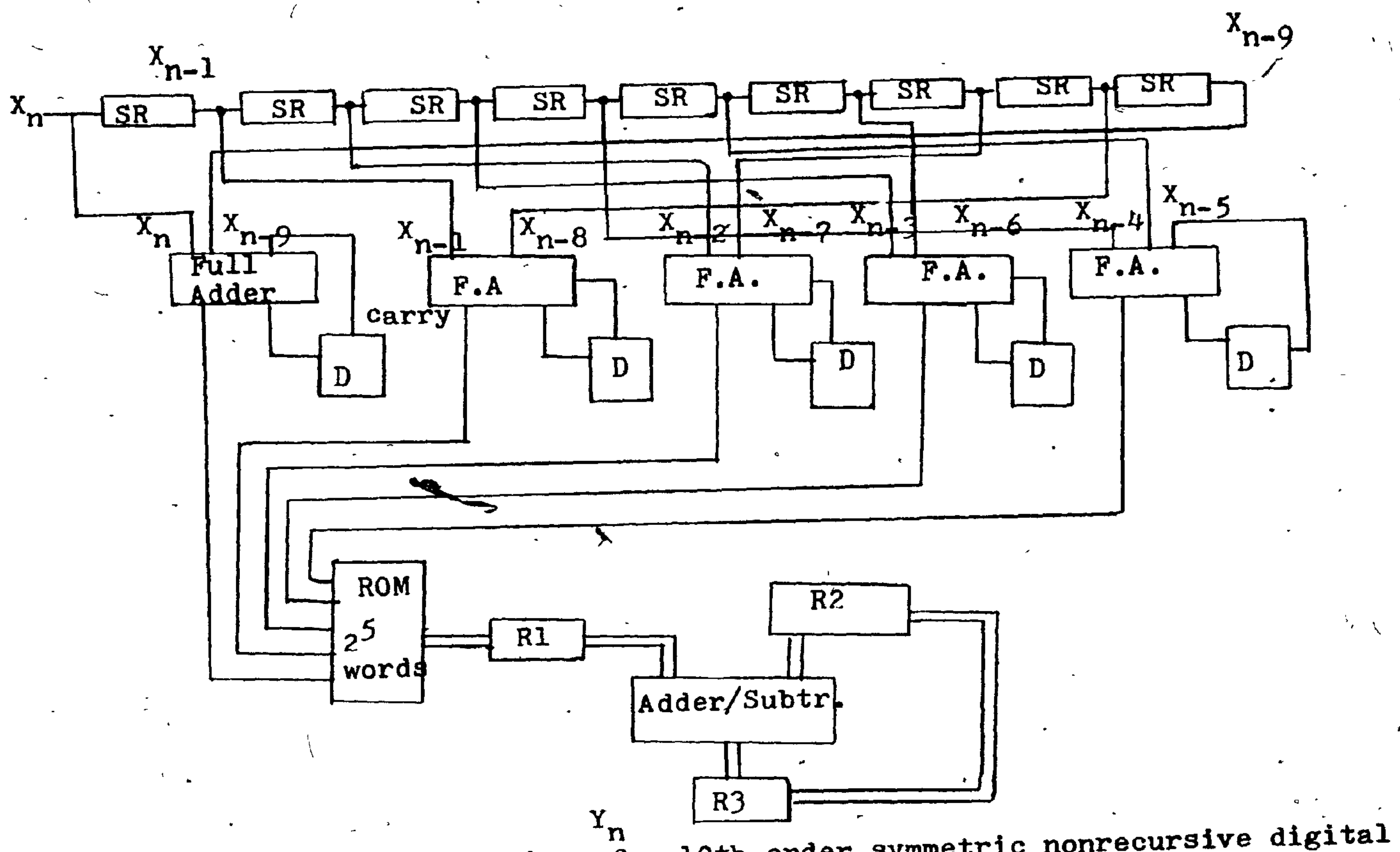


Figure 2.13 An implementation of a 10th order symmetric nonrecursive digital filter using 32 words of memory.

Here, a_i and b_i are the filter coefficients, and N-1 is the order of the filter and L is an integer. Using the binary representation, the input and output can be expressed as

$$X_n = \sum_{j=1}^{B-1} 2^{-j} x_{n,j} - x_{n,0}$$

$$Y_n = \sum_{j=1}^{B-1} 2^{-j} y_{n,j} - y_{n,0} \qquad (2.29)$$

By substitution and interchanging the order of summation, Equation (2.28) becomes

$$Y_n = \sum_{j=1}^{B-1} 2^{-j} \Big(\sum_{i=0}^{N-1} a_i x_{n-i,j} +$$

$$+ \sum_{i=1}^{L} (-b_i) y_{n-i,j} - x_{n-i,0} +$$

$$+ y_{n-i,0} \Big) = \sum_{j=1}^{B-1} 2^{-j} F_{n,j} \qquad (2.30)$$

where

$$F_{n,j} = \sum_{i=0}^{N-1} a_i x_{n-i,j} + \sum_{i=N}^{N+L} (-b_{i-N} y_{n-i+N,j}) \qquad (2.31)$$

The direct implementations of Figure 2.10 can be used again, except that L bits are coming from the output to be combined with the N input bits to form the address for the ROM which contains the values of $F_{i,j}$. This implementation is shown in Figure 2.14.

High-order filters can be implemented by breaking the filter into many second-order sections that can be connected in parallel. Equation (2.28) can be rewritten as

$$Y_n = \sum_{m=0}^{(N-3)/2} C_m V_{n,m} \qquad (2.32)$$

where

$$V_{n,m} = X_{n-m+2}\, a_{m+2} + a_{m+1} X_{n-m-1} + A_m X_{n-m} - \\ - b_{m+2} Y_{n-m-2} - b_{m+1} Y_{n-m-1} \qquad (2.33)$$

and C_m are constants used for scaling purposes. Each $V_{n,m}$ is a partial sum and can be summed directly as in Figure 2.15a or in parallel, as in Figure 2.15b. These partial sums can now be used to implement Equation (2.32) by addressing an ROM which contains all the possible values of $F_{n,j}$, according to Equation (2.31), and then accumulating the result to obtain Y_n. This implementation is shown in Figure 2.16 for a 6th-order filter.

Another way of implementing high-order filters is by operating serially, as in the implementation of Figure 2.12. The modification necessary to take care of the recursive nature of the filter is illustrated in Figure 2.17. The filter can be implemented with one second-order section multi-

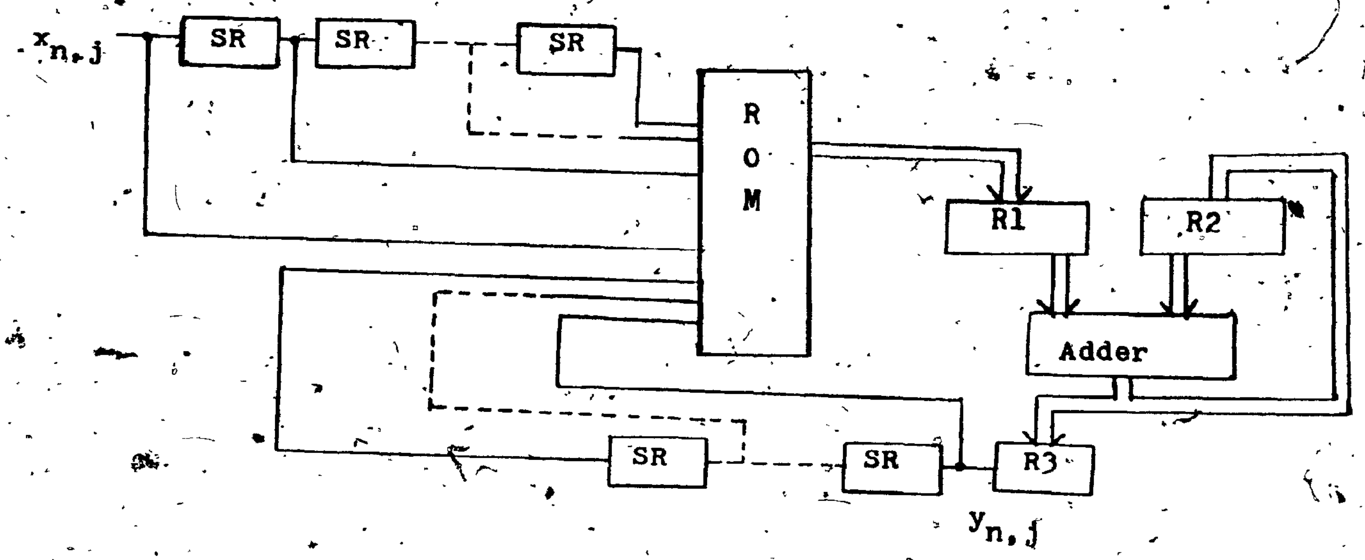


Figure 2.14 A bit serial implementation using ROM for a recursive digital filter.

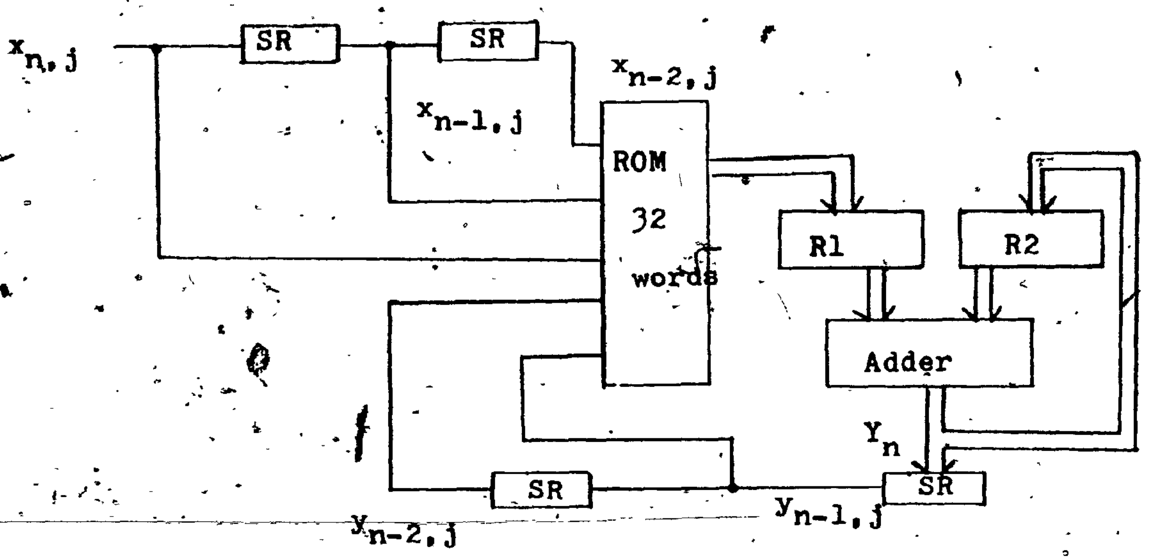


Figure 2.15(a) A second-order recursive filter using a ROM.

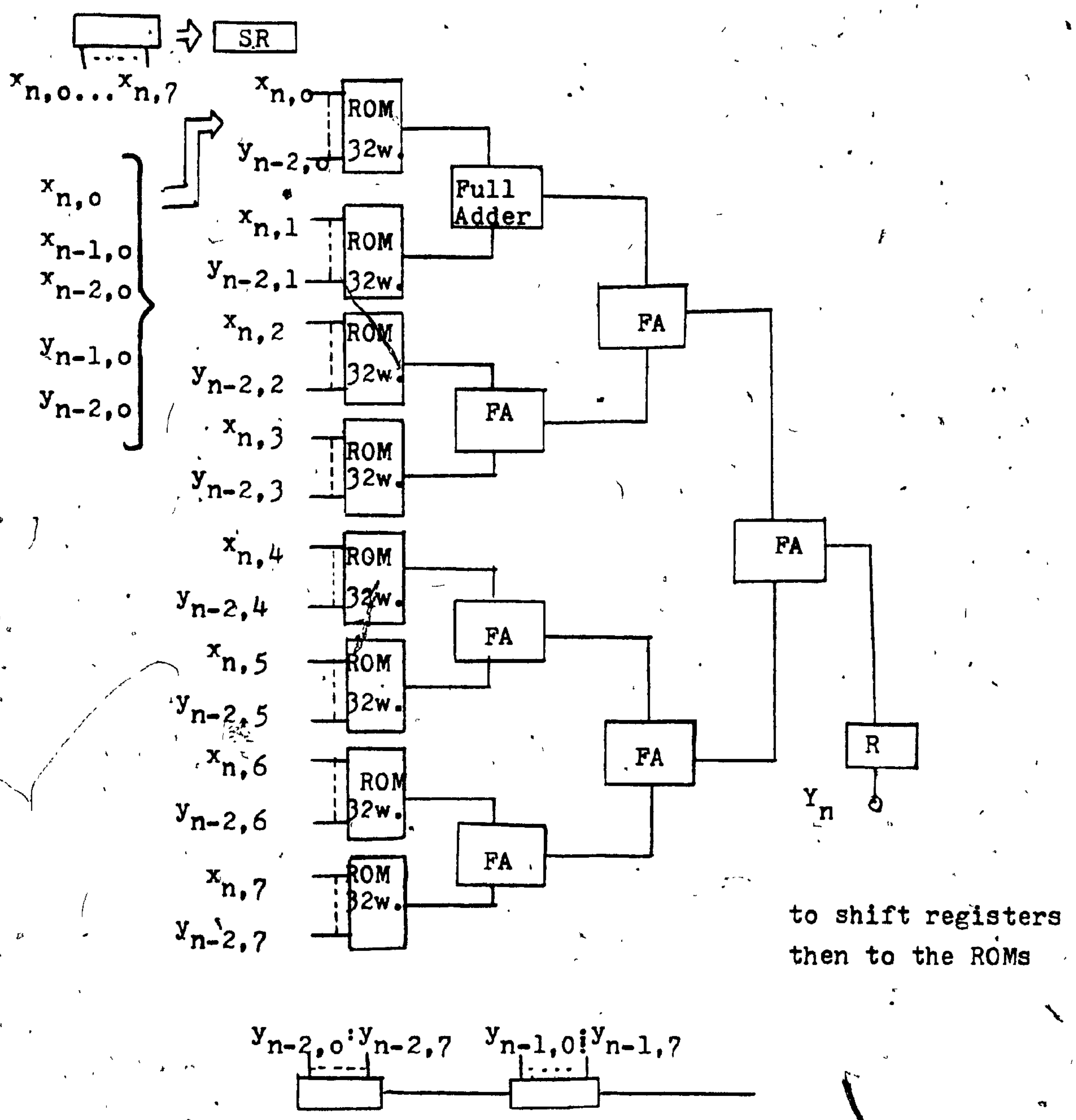


Figure 2.15(b) A fully parallel implementation of a second-order section using ROMs.

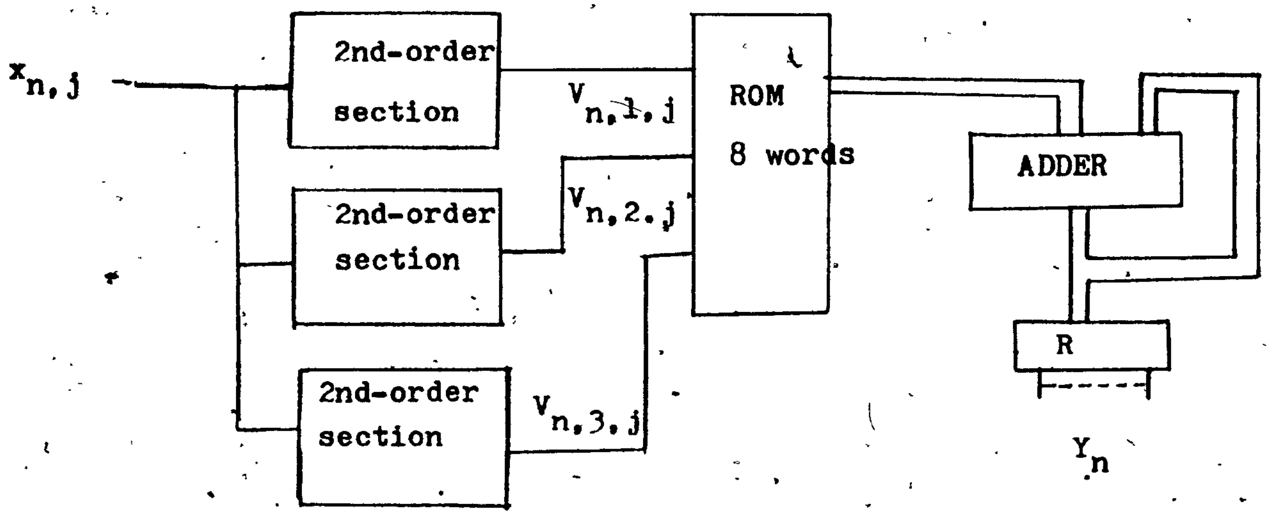


Figure 2.16 A parallel 6th order ROM implementation.

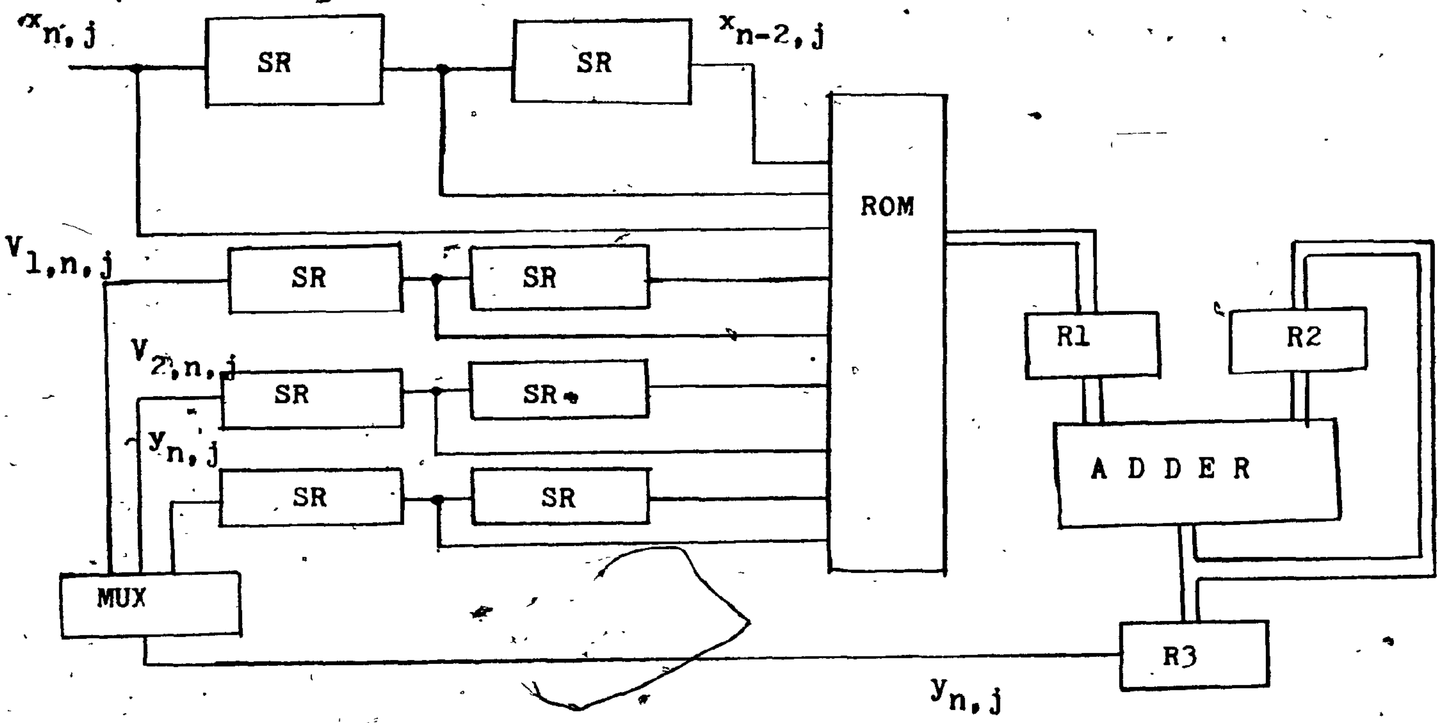


Figure 2.17 A multiplexed recursive digital filter using a ROM.

plexed (N-1)/2 times, using an ROM of $(N-1)2^{N-2}$ words.

2.3.4 Monkewich-Steenart Implementation

Monkewich and Steenart have adopted another way of implementating a second-order section of the recursive digital filter, [7]. They basically use ROMS to eliminate the need for multipliers. Five ROMS are used: three to contain all possible input signal products with each coefficient separately, and the other two for the output signal. Reading those five products that correspond to the input samples with each coefficient separately, the final output is obtained by summing up all the partial results as a common accumulator. This method is depicted in Fig. 2,18. Notice that it can be used to implement filters of any order through the use of 2nd - order sections.

2.4 IMPLEMENTATION USING COUNTERS

A completely different hardware realization of digital filters was introduced by Zohar [8,9] early in 1972. This method is different in the sense that the conventional multipliers or ROMs are not needed, at all. Instead, counters are used to calculate the output of the digital filter. The principle behind this method is very simple. Consider a filter in which the required output is the sum of the products of the corresponding coefficients and the delayed

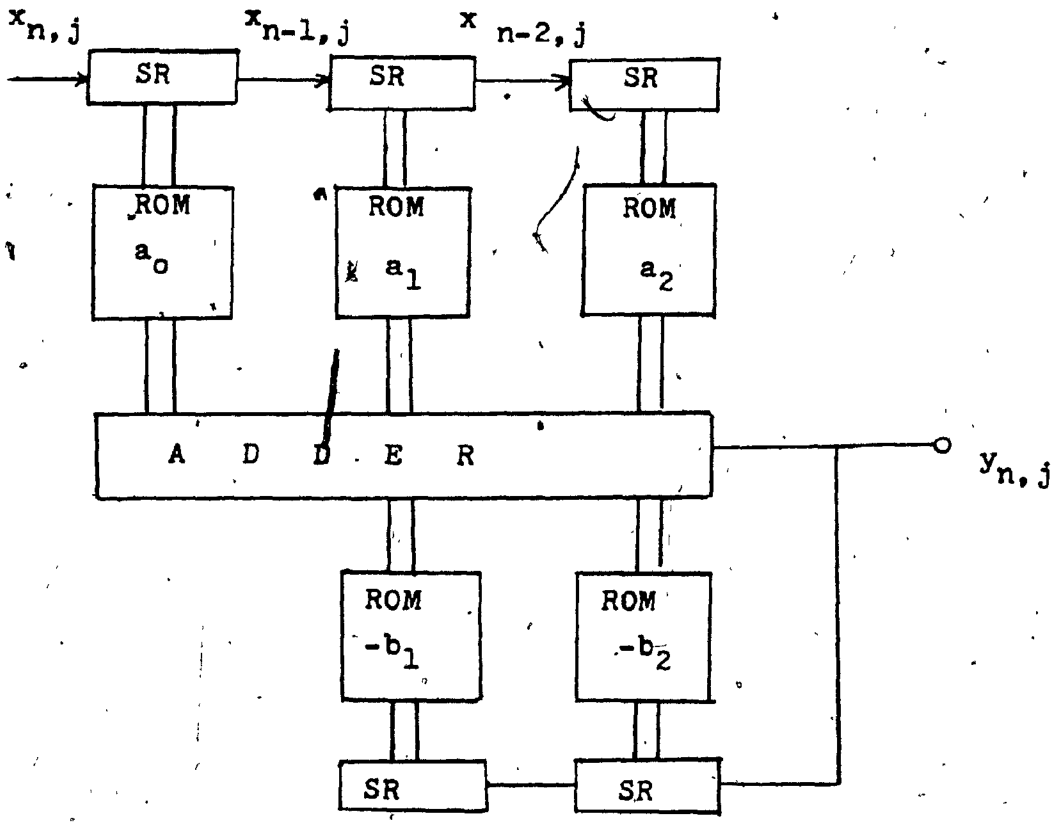


Figure 2.18 Steenart-Monkewich implementation of a second-order section.

inputs. Instead of considering all the i bits of a single product, where $0 \leq i \leq B - 1$, and B is the number of bits in each product, the i^{th} bit of each possible product is considered, first. The number of ones in these products (counted by a counter), will be the coefficients of the power-of-2 polynomial, which when evaluated, will give the final result. In other words, instead of the "multiply-then-add" or "shift-then-look-in-a-ROM-then-add" strategies, the "gate-then-count-then-shift-then-add" strategy is implied in this method.

2.4.1 Non-Recursive Digital Filters

If Equation (2.6) that represents the non-recursive digital filter, is scaled to integers, it can be rewritten as

$$Y_n = \sum_{K=0}^{N-1} a_K X_{n-K} \tag{2.34}$$

Since X_n and a_K are integers, they can be coded as binary numbers of word lengths B_x and B_a bits, respectively, as follows:

$$X_k = \sum_{j=0}^{B_x-1} \nu_{k,j} 2^j , \quad a_k = \sum_{j=0}^{B_a-1} \mu_{k,j} 2^j \tag{2.35}$$

where

$$\mu_{k,j}, \nu_{k,j} \in [0,1].$$

With these descriptions of X_k and a_k, Equation (2.34) becomes

$$Y_n = \sum_{k=0}^{N-1} \left[\sum_{i=0}^{B_a-1} \mu_{k,i} \, 2^i \right] \left[\sum_{j=0}^{B_x-1} \nu_{n-k,j} \, 2^j \right]$$

or

$$Y_n = \sum_{k=0}^{N-1} \sum_{j=0}^{B_t-1} \left(\sum_{i=0}^{B_a-1} \mu_{k,i} \, \nu_{n-k,j-i} \right) 2^j \qquad (2.36)$$

where

$$B_t = B_a + B_x - 1$$

Equation (2.36) can be rewritten as the summation over i, then k,j, by interchanging the order of summation, that is

$$Y_n = \sum_{j=0}^{B_t-1} \left(\sum_{k=0}^{N-1} \sum_{i=0}^{B_a-1} \mu_{k,i} \, \nu_{n-k,j-i} \right) 2^j \qquad (2.37)$$

Define the function h_j as

$$h_j = \sum_{k=0}^{N-1} \sum_{i=0}^{B_a-1} \mu_{k,i} \, \nu_{n-k,j-i} \qquad (2.38)$$

then, Equation (2.37) reduces to

$$Y_n = \sum_{j=0}^{B_t-1} h_j \, 2^j$$

The function h_j can be evaluated therefore by gating the all possible combinations of the bits of X_n and a_k and then counting the number of 1's that result. With this definition of h_j, Equation (2.37) can be rewritten as

$$Y_n = \sum_{j=0}^{B_t-1} h_j 2^j = (\ldots(((2h_{B_t-1}+h_{B_t-2})2 + \ldots h_1)2 + + h_0 \qquad (2.39)$$

This is simply a shift (i.e., multiply by 2) and add operation. Equations (2.38) - (2.39) are the basis of this implementation. It can be summarized as follows.

1) Pass the input bits in parallel to the bits of the corresponding coefficient in reverse order(because of the j-i subscript in Equation (2.38)) and gate them every time. Do this each time you feed in another bit of the input.

2) When gated, use a counter, to count how many TRUE or 1 gates result.

3) Use the output of the counter to calculate the output of the filter by accumulating all the results in an accumulator, a shift register, that will add and shift, hence multiplying by 2, each time a bit of the input causes an evaluation of a part of the output.

4) Repeat this for all the bits of the input.

This realization needs two registers, one to hold the coefficients of the filter a_k, $k=0,1, \ldots, N$, and the other to hold momentarily, the bits of the current input and the previous values of it in a sequential way. There should be a matching between the cells of these two registers such that the paired bits of the input and the corresponding coefficient are gated by usual AND gates, and the results fed to a binary counter. Figure 2.19 illustrates this process. The output of the counter is then fed to a shift register which is used to multiply by 2, and add these partial products. After passing all the bits of the input into the two registers, the final result y_n is obtained. The case of a $3^{\underline{rd}}$ - order non-recursive filter, using 5 bits for the coefficients and 3 bits for the input word, is shown in Figure 2.20.

2.4.2 Extension To Negative Numbers

The most common methods of representing negative numbers in binary arithmetic are either by the signed magnitude (bit 0 for +ve and bit 1 for -ve) or by using the two's-complement form. The number 6 in base 10 is (0110) in base 2, assuming a 4-bit word-length. If a fifth bit is added as a sign bit, thus having a word length of 5 bits, then $(-6)_{10}$ is $(10110)_2$. Using the 2's-complement form, we get for $(-6)_{10}$ the binary form of $(1010)_2$. This is achieved by complementing every bit and adding one to the overall result.

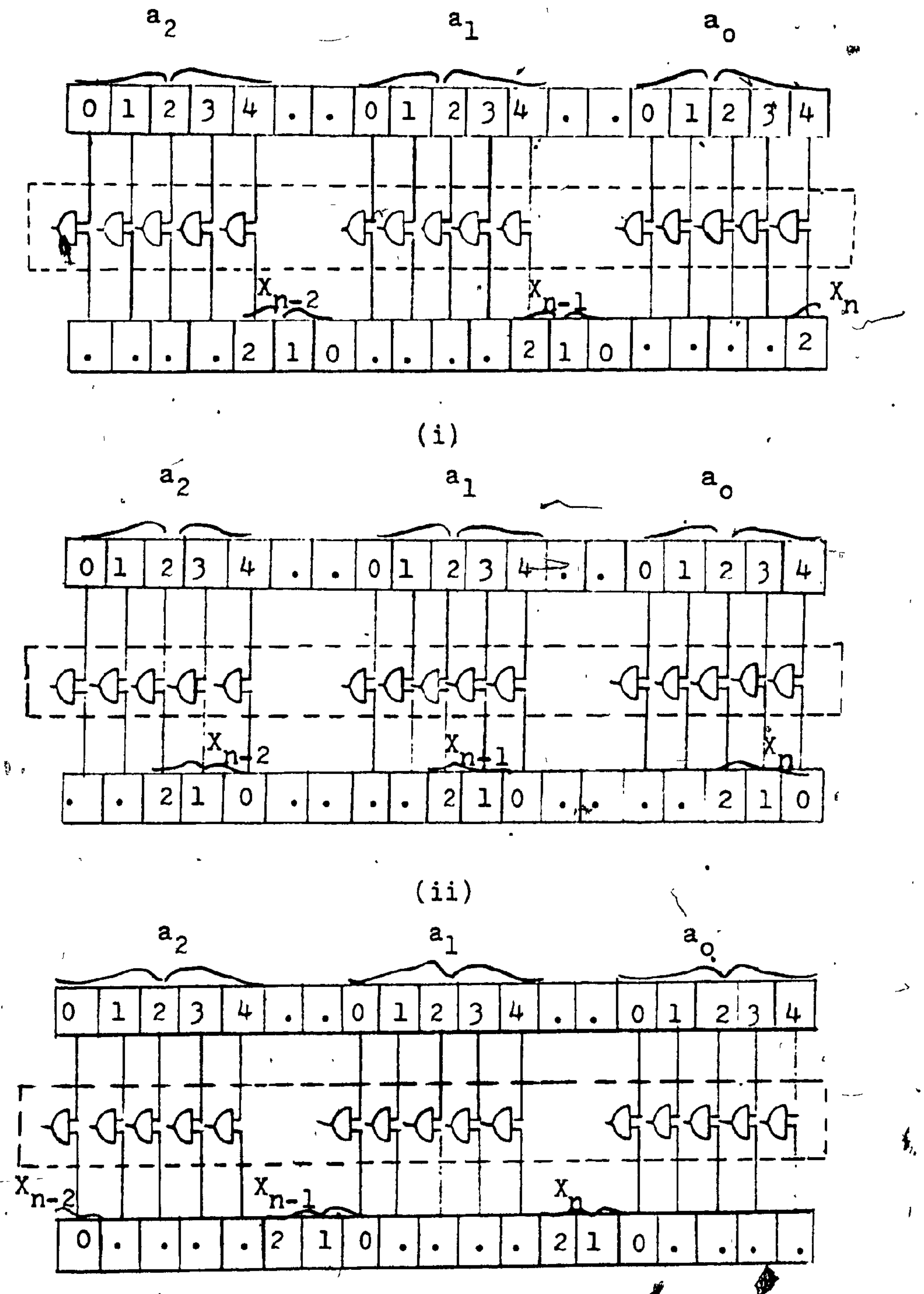
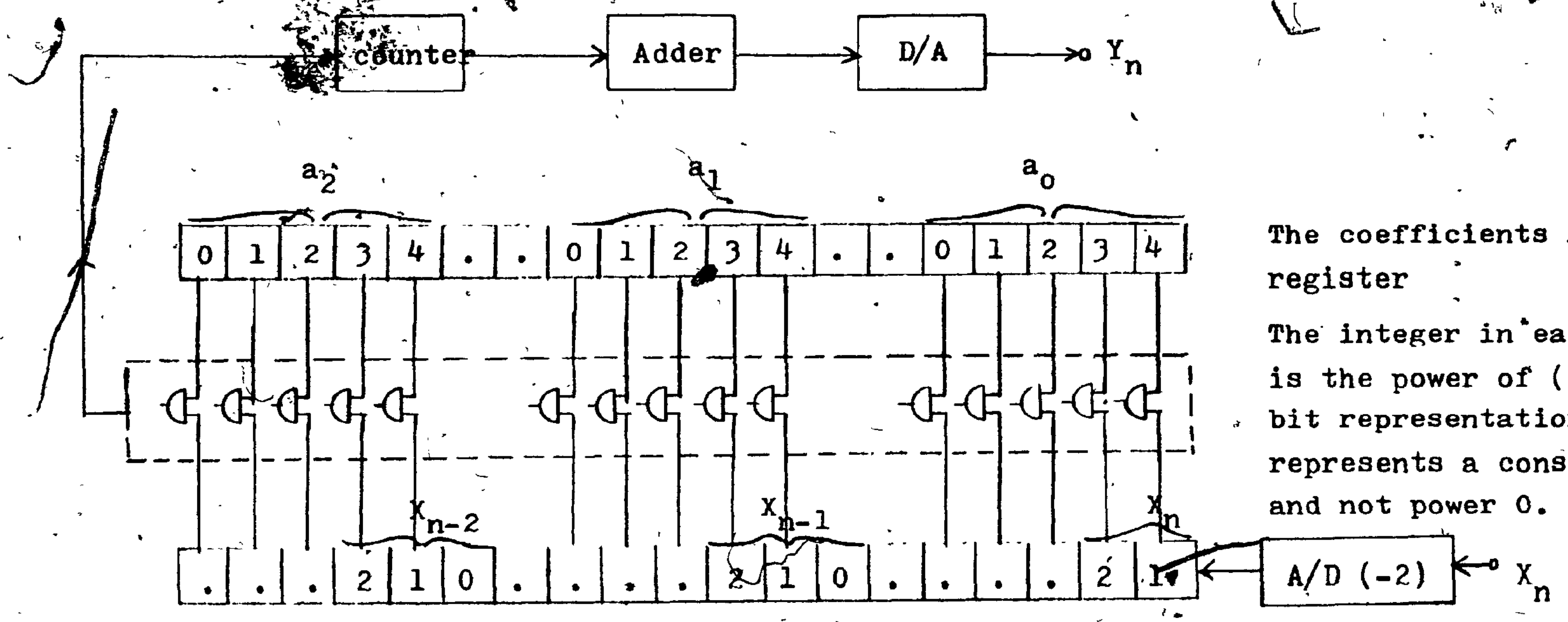


Figure 2.19 The shifting and counting process.
(Integers in the cells of the shift registers indicate the power of the -2 radix for that particular bit.A(.) represent a zero bit.)



The coefficients A ~~shift~~ shift register

The integer in each cell is the power of (-2) in bit representation. A (.) represents a constant 0, and not power 0.

Figure 2.20 The nonrecursive counting digital filter.

The alternative to these two ways of handling negative numbers is the negative radix representation. If z is an integer, it can be represented as a set of B binary bits with a radix of $+2$ as

$$z = \sum_{i=0}^{B} z_i (+2)^i, \quad \text{where } z_i \text{ is either 0 or 1} \qquad (2.40)$$

If the radix used is (-2), Equation (2.40) becomes

$$z = \sum_{i=0}^{B} z_i (-2)^i \qquad (2.41)$$

The representation of Equation (2.41) can be used to represent negative integers. For example, our $(+6)_{10}$ becomes $(11010)_{-2}$ and $(-6)_{10}$ becomes $(01110)_{-2}$. This representation provides a much simpler method for the handling of negative numbers and hence, makes this implementation much simpler than that which uses the 2's-complement or the signed magnitude number representations. The fact that the increase in word length for negative radix form is eventual, the trade-off between the increase in word-length and the complexity of the hardware for the 2's-complement or the signed magnitude is worth it.

Equation (2.35) can be redefined by using a negative radix such as

$$X_K = \sum_{j=0}^{B_x-1} \nu_{k,j}(-2)^j, \quad a_k = \sum_{j=0}^{B_a-1} \mu_{k,j}(-2)^j$$

Hence, from Equation (2.34), we get

$$Y_n = \sum_{k=0}^{N} \sum_{j=0}^{B_t-1} \sum_{i=0}^{B_a-1} \mu_{k,i}\, \nu_{n-k,j-i}(-2)^j = \sum_{j=0}^{B_t-1} h_j(-2)^j$$

where

h_j is as defined in Equation (2.38).

The implementation, using the negative radix, is the same as that described before, basically because the counter counts the TRUE or 1's irrespective of the form of representation used. Of course, the input bits should be in the negative radix form, therefore, a negative radix converter has to be used, but the output is obtained directly in the standard binary form. Notice that h_j is exactly the same as before, i.e., it is independent of the number system used, although now its arguments are in the negative radix system. Also, the shift in the accumulator has to be modified, such that it will multiply by (-2) and not by (+2). This can be done by adopting the signed-magnitude representation for the accumulator. This is of particular interest, because the output of the counter will be added to the accumulator, a shift (multiplication by 2) then follows, and then the sign bit will be reversed immediately, (multiplication by -1).

The coefficients have to be stored in the hold register in the negative radix form. They can be computed on a digital computer by using a program. However, the input values have to be converted too, to base (-2) and that has to be done in the filter instantaneously. An analog input has to be fed into an A/D converter, i.e., to be converted from decimal to base +2. Here, the digitized input, in binary +2, can be fed to a converter which would convert from base +2 to base -2. This radix converter was suggested also by Zohar, [10]. Of special interest is a converter that will convert directly from analog to base (-2). This can be manufactured as a combined circuit from an A/D and a negative radix converter.

In summary, the mode of operation of this implementation is as follows:

1) Accept the analog input and quantize it to an approximation which gives scaled integers.

2) Pass this input into an A/D converter, then convert it through a negative radix converter to base (-2).

3) Feed these input bits to a shift register which is cross-linked to a hold register, which contains the bit (-2) representation of the filter coefficients. Each time an input bit is added, gate the cross-linked bits of the input and the coefficients and count the

number of TRUE's that result from the gating process.

4) Feed the count produced to an accumulator/shift register that will add to its contents the count received, then shift the contents of the input register to the left, thus multiplying by (+2) and then complement the sign bit of the accumulator to account for the fact that base (-2) is being used. Do this each time a new bit of the input word is fed to the input shift register.

5) When this is done for one complete word for the input, the value present in the output accumulator/shift register is the required output. Pass this to a D/A converter (normal D/A, i.e., from base +2, not from base - 2), then rescale the output of the D/A.

2.4.3 Recursive Digital Filters

Consider the recursive digital filter characterized by Equation (2.28), as

$$Y_n = \sum_{i=0}^{N-1} a_i X_{n-i} - \sum_{i=1}^{L} b_i Y_{n-i} \qquad (2.28)$$

If an integer variable, Z is defined as

$$Z_{n-1} = X_{n+1} + \sum_{k=1}^{L-1} Z_{n-k}(-b_{k+1}) \qquad (2.42)$$

then it can be shown [9] that Equation (2.28) becomes

$$Y_n = \sum_{i=0}^{N-1} Z_{n-i}\, a_i \qquad (2.43)$$

The input data and the filter coefficients now can be represented by bits in base (-2) so as to be able to represent negative numbers, as well. These can be expressed as follows:

$$a_k = \sum_{j=0}^{B_a-1} \mu_{k,j}(-2)^j$$

$$b_k = \sum_{j=0}^{B_a-1} \nu_{k,j}(-2)^j \qquad (2.44)$$

$$z_k = \sum_{j=0}^{B_a-1} \rho_{k,j}(-2)^j$$

where

B_a is the coefficient word-length (a_k and b_k are assumed to have the same word-length, otherwise B_a is the maximum of both lengths),

B_z is the word-length for z_k) and

$$(\mu_{k,j}, \nu_{k,j}, \rho_{k,j}) \in [0,1].$$

Substituting for a_k and z_{m-k} in Equation (2.43):

$$Y_n = \sum_{k=0}^{N-1} \left(\sum_{j=0}^{B_z-1} \rho_{n-k,j}(-2)^j\right)\left(\sum_{j=0}^{B_a-1} \mu_{k,j}(-2)^j\right)$$

$$= \sum_{k=0}^{N-1} \sum_{j=0}^{B_t-1} \left(\sum_{i=0}^{B_a-1} \mu_{k,i}\rho_{n-k,j-i}\right)(-2)^j$$

Here, B_t is used for the total word-length,

$$B_t = B_a + B_z - 1$$

Again, by interchanging the order of summation, we get

$$Y_n = \sum_{j=0}^{B_t-1} \left(\sum_{k=0}^{N-1} \sum_{i=0}^{B_a-1} \mu_{k,i}\rho_{n-k,j-i}\right)(-2)^j$$

$$= \sum_{j=0}^{B_t-1} h_j(-2)^j \qquad (2.45)$$

where

$$h_j = \sum_{k=0}^{N-1} \sum_{i=0}^{B_a-1} \mu_{k,i}\rho_{n-k,j-i} \qquad (2.46)$$

Thus, if Z_n' is known, we can gate its bits with the bits of the a_k coefficients, then the resulting count, h_j, can be used to compute the power-of-minus-2 polynomial (Equation (2.45)), and hence, obtaining Y_n. The polynomial evaluation is nothing but add-and-shift-and-complement-sign-bit operation, which can be done in an ordinary accumulator/shift register.

To calculate Z_n, we substitute Equation (2.44) in Equation (2.42):

$$Z_{n-1} = [X_{n+1} + \sum_{k=0}^{L} (\sum_{j=0}^{B_z-1} \rho_{n-k,j}(-2)^j)(\sum_{i=0}^{B_a-1} \nu_{k,i}(-2)^i)]$$

and after some manipulation and arrangement, as done before, we get

$$Z_{n-1} = [X_{n+1} + \sum_{j=0}^{B_t-1} g_j(-2)^j] \quad (2.47)$$

where

$$g_j = \sum_{k=0}^{L} \sum_{i=0}^{B_a-1} \nu_{k,i}\rho_{n-k,j-i} \quad (2.48)$$

According to Equation (2.47), Z_n can be constructed by evaluating a power-of-(-2) polynomial whose coefficients are the counts resulting from gating all the bits of the b_k coefficients and the previous values of Z. This results in a count which should be added to X_{n+i}.

Therefore, the complete realization of the recursive digital filter described by Equation (2.28) can be carried out as follows:

1) Accept the analog input to the filter, quantize and approximate to give scaled intergers.

2) Pass this input integer into an A/D converter, thus converting it to base (+2), standard binary.

3) Feed this to an accumulator, then pass it into a negative radix converter to convert it to base (-2).

4) Pass the bits out of the negative converter into a shift register which is cross-linked to a general register which contains the bits of the a_k coefficients on one side and to another register which contains the bits of the b_k coefficients on the other side. Notice that the stored coefficients must all be in base (-2). Each bit coming from the input negative radix converter is gated with the bits of b_k, thus calculating the next value of the input to the negative radix converter, and simultaneously is gated to the bits of a_k, hence calculating the partial value of the required output.

5) Take the output of the gates of Z_n and a_k to a counter, then to an accumulator, where the output will be calculated by summing the outputs of that counter, shifting to the left and flipping the sign bit. Take the other output of the gates (Z_n and b_k), to a counter, then add the count to the input accumulator, thus evaluating, after adding the new input from the A/D, the value of Z_{n+1}.

6) Repeat (4) and (5) until all the bits of the input are fed to the shift register, which is cross-linked to the coefficient-registers. Then, take whatever is in the output accumulator and convert it to analog through a D/A which becomes the output of the filter, then take the contents of the input accumulator, add to it the incoming new input from the A/D converter, and then repeat the whole process for a new value of the output.

Figure 2.21 shows this implementation, considering a simple filter as an example.

The input bits of Z_n can be fed to the shift register in two ways: the most significant bit leading (decreasing index sequence DIS) or the least significant bit leading (increasing index sequence IIS). In the DIS case, y_m is calculated, starting with the function H_{B_t-1}, which, in turn, is calculated by gating all the bits of the coefficients a_k with the (B_t-1)th bit of Z_n, according to Equation (2.47) and Equation (2.48). This process requires that the value of Z_{m-1} be computed completely before starting to compute the value of Z_n, i.e., before starting to feed in the bits of Z_{n-1} to the negative radix converter, and the shift register. This is because the evaluation of the Z_n is done by addition in the accumulator, which produces the bits of the new value in IIS, hence, there should be a

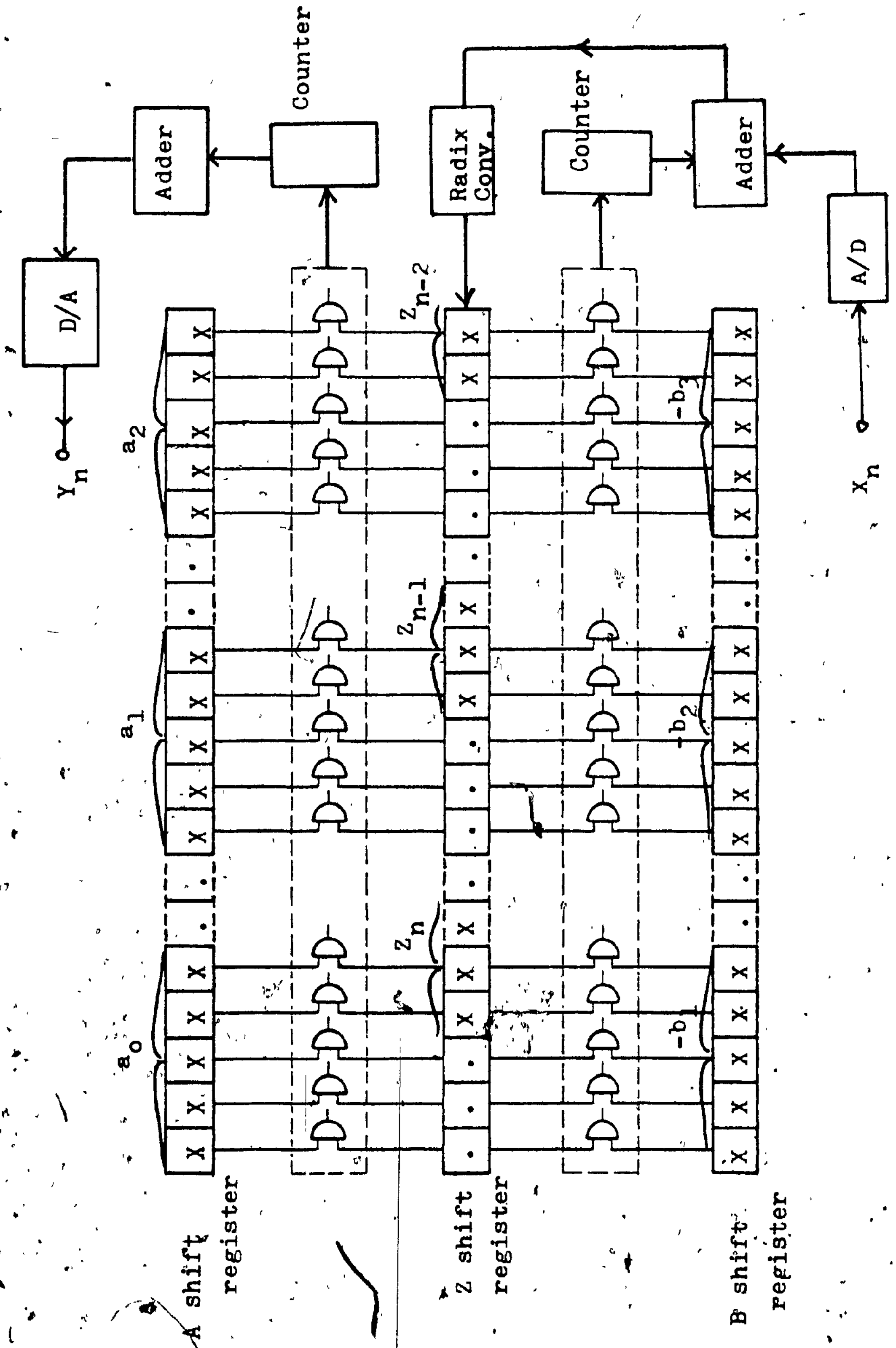


Figure 2.21 The recursive filter implemented with counters.

storage device to store all the bits of the sum until the last one, i.e., the most significant bit which is supposed to be first of all other bits.

On the other hand, feeding in using IIS, requires that the least significant bit be leading. Since this least significant bit is the first to be evaluated in the addition process, we do not have to wait until all the bits of Z_m are calculated, but rather, feed immediately the bit that is produced by the addition to the next step for evaluating the value Z_{n+1}. This means that h_o is calculated first, rather than h_{B_t-1}, as is in the case of DIS. No auxiliary storage is needed, either, hence, it is faster and simpler in terms of hardware components. The important feature of the IIS case is that the shift has to be to the right, multiplying by 1/2, which means that there is an assumed binary point for (B-1) bits from the left of the register.

The way the input bits are fed affects the coefficient registers, since always the bits of both registers, the z_m and a_k should be in opposite order. Figure 2.22, and Figure 2.23 show both IIS and DIS operations.

2.4.4 Other Variations

Other implementations using counters, can be derived by simply varying the general form of the filter equation. In fact, better implementations can be obtained, once few

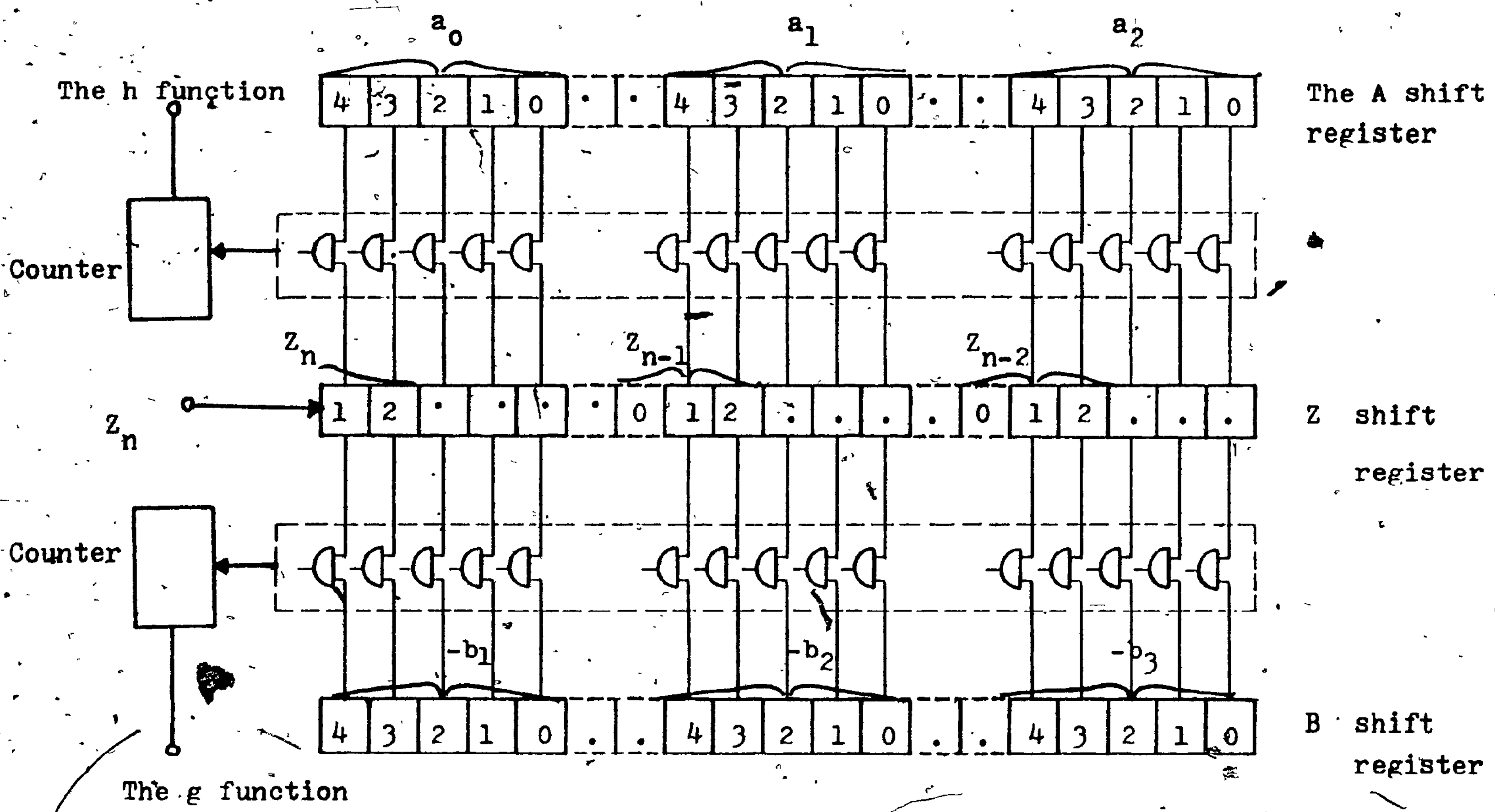


Figure 2.22 Shifting in an increasing index (IIS) sequence.

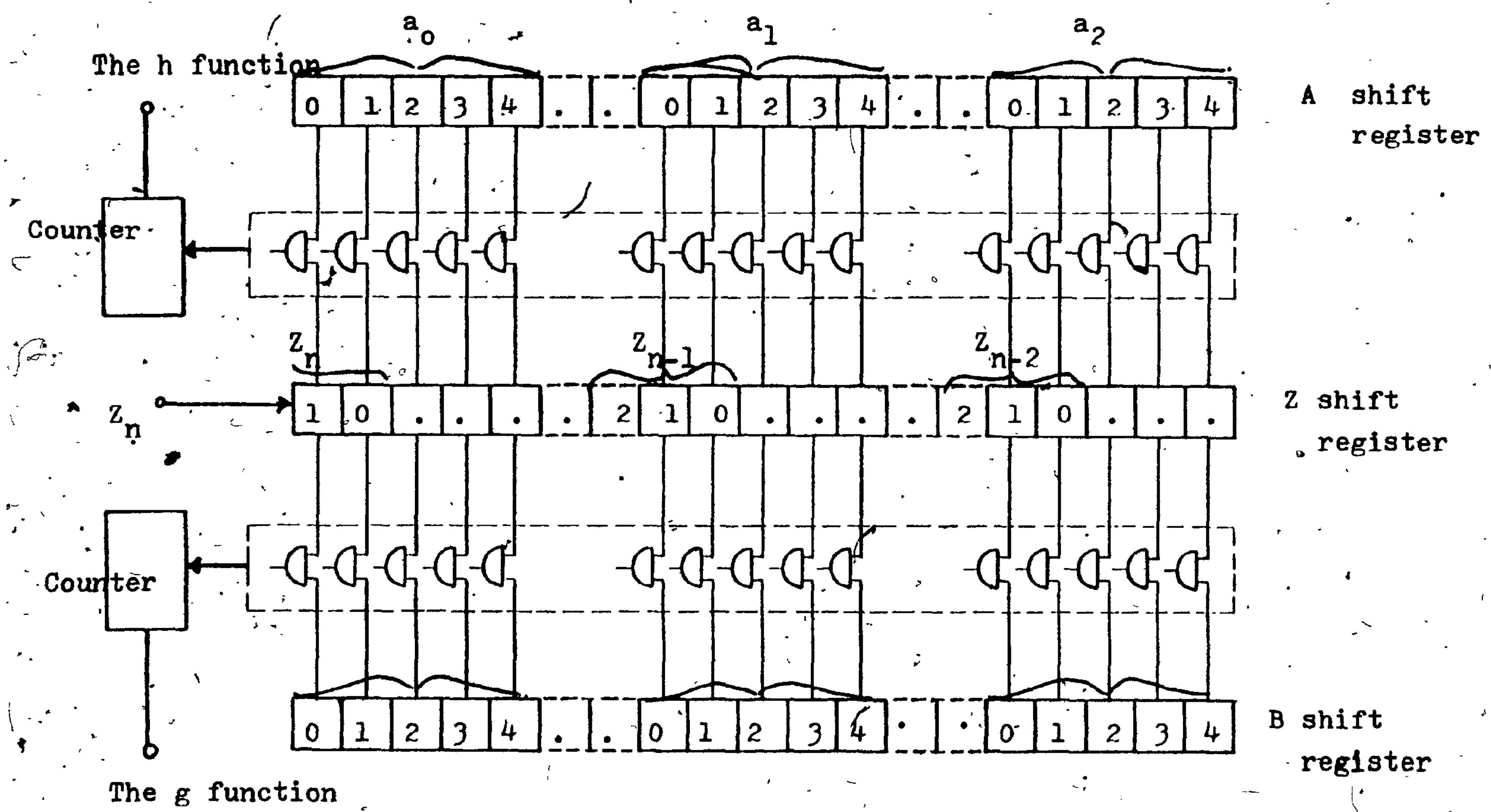


Figure 2.23 Shifting in a decreasing index (DIS) sequence.

$$Y_n = \sum_{i=o}^{N-1} a_i X_{n-i}$$

Let the values of a_i be either unsigned positive binary numbers or binary numbers represented in the 2's complement form. If each input data is represented in a word of B bits, then X_n can be rewritten as:

$$X_n = \sum_{j=1}^{B} x_{n,j} 2^{-j} \qquad (2.60)$$

and

$$x_{n,j} \in \{0,1\}$$

The input is limited to the range [1,-1], i.e.,

$$-1 < X_n < 1$$

This condition can be satisfied by using scaling. Substituting Equation (2.60) in Equation (2.6)

$$Y_n = \sum_{i=o}^{N} (a_i \sum_{j=1}^{B} x_{n-i,j} 2^{-j}$$

After interchanging the order of summation, we get

$$Y_n = \sum_{j=1}^{B} (\sum_{i=o}^{N} a\, x_{n-i,j}) 2^{-j} \qquad (2.61)$$

If we define the function $F_{n,j}$ as

$$F_{n,j} = \sum_{i=o}^{N} a_i \, x_{n-i,j} \tag{2.62}$$

then Equation (2.61) can be expressed as:

$$Y_n = \sum_{j=1}^{B} F_{n,j} \, 2^{-j} \tag{2.63}$$

This is a polynomial of 2^{-j}, which has as coefficients the values of the function $F_{n,j}$. These coefficients can be either precalculated and stored in a table, or they can be calculated at the time they are needed. Hence, a trade-off exists. Speed is achieved by precalculation, but the size of memory required is large. On the other hand, the size of memory required can be minimized but speed is reduced, due to the large amount of computation time. However, since high speed is highly desirable, the function $F_{n,j}$ should be pre-calculated and stored in memory.

Define a table, C of 2^N entries, i.e.,

$$C(K), \; K = 0,1, \ldots, 2^N-1$$

such that

$$C(K) = \sum_{i=o}^{N-1} K_i \, 2^i \qquad K_i \in \{0,1\} \tag{2.64}$$

Comparing Equation (2.62) and Equation (2.64), we can choose K as

$$K \equiv K_{n,j} = \sum_{i=o}^{N} X_{n-i,j} \, 2^{i} \qquad (2.65)$$

for all the B bits j = 0,1, ..., B-1. This is equivalent to taking the j^th bit of the input and its delayed values to form a binary number of N bits. Equation (2.62) and Equation (2.64) become equivalent, i.e.,

$$F_{n,j} = C(K_{n,j}) \qquad (2.66)$$

The n^th address, is given by Equation (2.65). The (n-1) address, $K_{n-1,j}$ is:

$$K_{n-1,j} = \sum_{i=o}^{N-1} X_{n-1,i,j} \, 2^{i} = 2^{-1}[\sum_{i=o}^{N-1} X_{n-1,i,j} \, 2^{i+1}]$$

$$= 2^{-1}[\sum_{i=o}^{N-1} X_{n-i,j} \, 2^{i} - X_{n,j} + X_{n-N,j} \, 2^{n+1}]$$

$$= 2^{-1}[K_{n,j} - X_{n,j} + X_{n-M+1,j} \, 2^{M}]$$

Hence

$$K_{n,j} = 2\,K_{n-1,j} - X_{n-M+1,j} \, 2^{N} + X_{n,j}$$

$$= [2K_{n-1,j}]\bmod 2^{N} + X_{n,j} \qquad (2.67)$$

This is a powerful result, since it enables the next address to be generated directly from the present one. Since the address is a binary number, it is stored in a shift register, such that it can be shifted one bit towards the most significant bit, thus multiplying the current address by 2, as Equation (2.67) shows. The most significant bit would be shifted out of the register, hence the modulo-2^N is computed. Next, in the least significant bit, which is empty now, the $j^{\underline{th}}$ bit of the present value of input is shifted.

The algorithm now is complete. The value of Y_n can be calculated as follows:

1) Find the address of the $j^{\underline{th}}$ coefficient, $K_{n,j}$, by taking the $j^{\underline{th}}$ bits of the input and its delayed values and form an N-bit binary number. Use this number to get $C(K_{n,j})$ from the ROM.

2) Shift the address register towards the most significant bit, i.e., multiplying by 2, and feed in the least significant bit, the next bit of the current input value considered. This will be the address $K_{n-1,j}$. Use the addressed value $C(K_{n-1,j})$ to add it to the final accumulator and then shift it to continue the computation of Equation (2.63).

3) Repeat (2) for all the B bits of the input, in order to compute Y_n. Figure 2.26, depicts this implementation.

2.5.2 Recursive Digital Filters

The difference equation (Equation (2.7)) characterizing a recursive filter, can be expressed as

$$Y_n = \sum_{i=o}^{N-1} a_i X_{n-i} + \sum_{i=1}^{N-1} (-b_i) y_{n-i} \tag{2.68}$$

where

$$Z_x = \sum_{i=o}^{N-1} a_i X_{n-i}$$

and

$$Z_y = \sum_{i=1}^{N-1} Y_{n-i} b_i' \qquad (b_i' = -b_i)$$

Each of these difference equations can be implemented by using the approach of Section 2.5.1.

Two ROMs are used separately to hold tables for both Z_x and Z_y.

One other way of realizing the recursive digital filter is through combining both ROMs into one ROM of 2^{2+N} words.

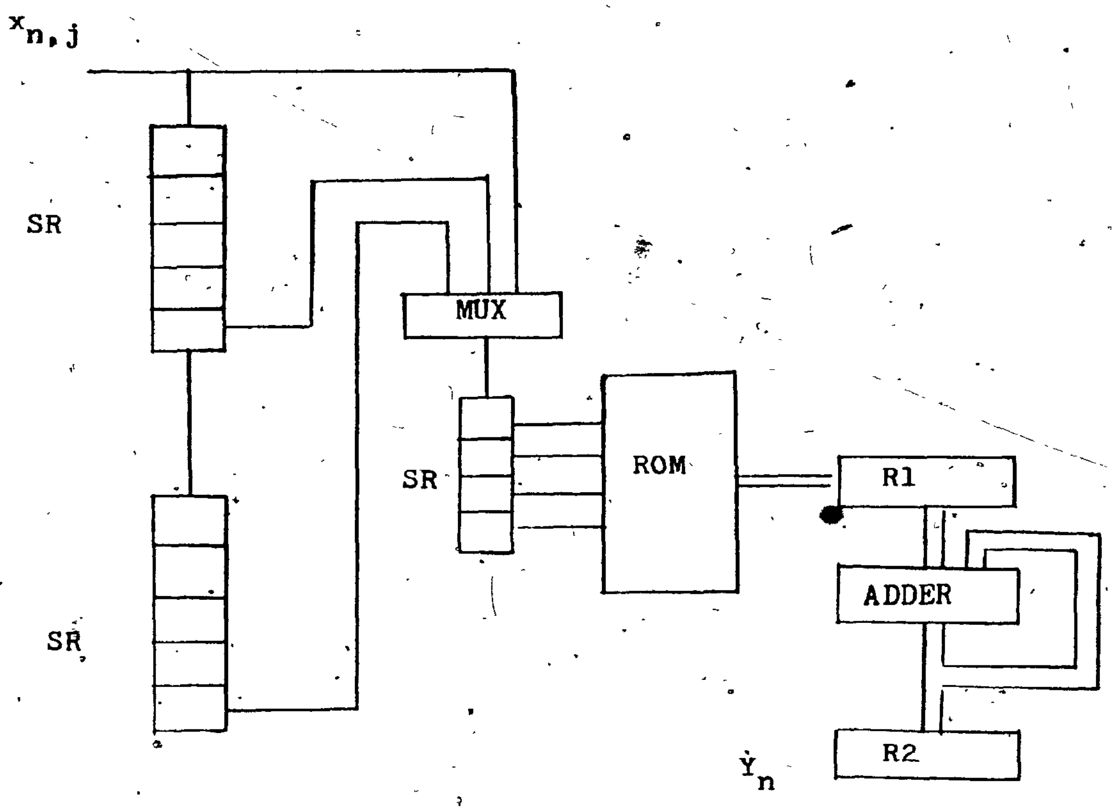


Figure 2.26 Nônrecursive filter implemented according to Little's algorithm.

Define K as a $2N$ bit binary number such that

$$K \equiv K_{n,j} = \sum_{i=o}^{2N-1} V_i \, 2^i \tag{2.69}$$

where

$$V_i = \begin{cases} X_{n-i,j} & \text{for } i < N \\ Y_{n+M-i-1,j} & \text{for } i \geq N \end{cases} \tag{2.70}$$

Table $C(K)$ becomes

$$C(K) = \sum_{i=o}^{N-1} a_i V_i + \sum_{i=1}^{N} b_i V_{i+M-1} \tag{2.71}$$

where

$$K = 0,1,2, \ldots, 2^{2N} - 1$$

A similar analysis for $K_{n-1,j}$, to that used to derive Equation (2.67) yields the following result:

1) To obtain the $K_{n,j}$, take all the $j^{\underline{th}}$ bits from the input and its delayed values, then take the $j^{\underline{th}}$ bits of the $N-1$ past values of the computed output and form a binary number of $2N$ bits as:

$$K_{n,j} \equiv V_{2N-1}, \ldots, V_N V_{N-1}, \ldots, V_1 V_0$$

$$\equiv Y_{n-N+1,j}, \ldots, Y_{n-1,j} X_{n-N+1,j}, \ldots, X_{n,j}$$

2) Address the ROM and use the $C(K_{n,j})$ to add to the accumulator then shift the contents of this accumulator.

3) Shift the address $K_{n,j}$ one bit towards the most significant bit, then replace its 0th bit (the least significant, V_0) by the current input's jth bit. Also replace the Nth bit, V_N by the jth bit of the computed output.

4) Repeat (2) and (3) for every bit of the B bits. This implementation is shown in Fig. 2.27.

2.5.3 Extension To Negative Numbers

The negative input values can be coded in three ways: The signed magnitude, the 2's complement and in the negative radix form. To use the signed inputs, the sign bit should not be used in giving the storage addresses. This can be done by "biasing" the input.

Let β be a number such that

$$0 \leq X_i + \beta \leq 1 - 2^{-N} \tag{2.72}$$

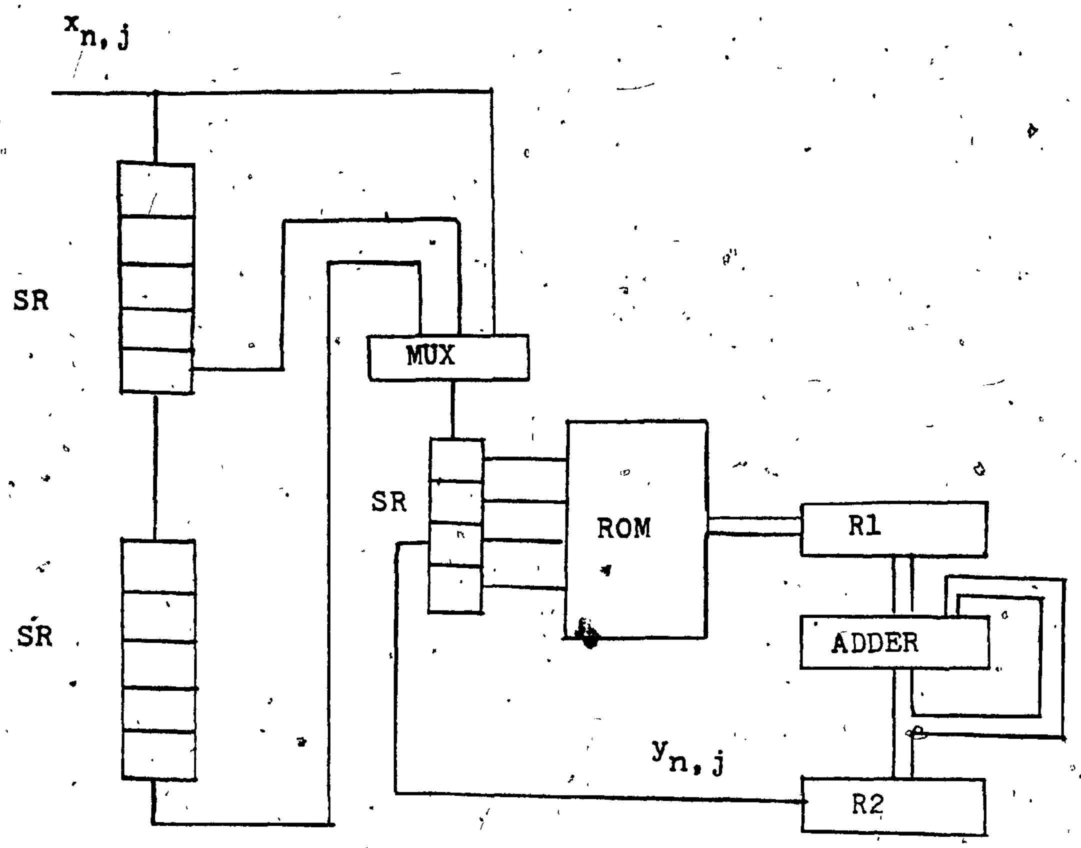


Figure 2.27 Recursive filter implemented according to Little's algorithm.

then Equation (2.6) becomes

$$Y_n = \sum_{i=o}^{N-1} a_i (X_i + \beta) = \sum_{i=o}^{N-1} a_i X_i + \sum_{i=o}^{N-1} a_i \beta \qquad (2.73)$$

Therefore, negative inputs can be handled by adding β to them, compute the output, then subtract from it the term $\beta \sum_{i=o}^{N-1} a_i$, as in Equation (2.73).

One other way, suggested by Yuen [12], is that of using the 2's complement representation for the input. Since it was assumed earlier that

$$0 \leq X_{n-i} < 1$$

we have

$$X_{n-1} \equiv \sum_{j=1}^{B} x_{n-i,j} 2^{-j} \qquad (2.74)$$

With $x_{n-i,j}$ being the $j^{\underline{th}}$ bit of X_{n-i}. If a sign bit is used, and X_{n-i} is represented in the 2's complement form, we can rewrite Equation (2.74) as

$$X_{n-i} = - S_{n-i} + \sum_{j=1}^{B} x_{n-i,j} 2^{-j} \qquad (2.75)$$

where

S_{n-i} is the sign bit of X_{n-i}.

Take this sign bit as the $0^{\underline{th}}$ input bit, $x_{n-i,0}$, then Equation (2.75) becomes

$$X_{n-i} = \sum_{j=o}^{N} x_{n-i,j} \, P(j) \qquad (2.76)$$

where

$$P(j) = \begin{Bmatrix} -1 & \text{if } j = 0 \\ 2^{-j} & \text{otherwise} \end{Bmatrix}$$

With this representation, Equation (2.63) becomes

$$Y_n = \sum_{j=o}^{N} F_{n,j} \, P(j) \qquad (2.77)$$

The input to recursive filters would follow the same analysis as shown for the non-recursive filters. The third way of handling negative numbers, is that of using the negative radix conversion. The input should be converted from base + 2 to base - 2, hence enabling the coding of negative numbers.

2.6 THE RAM DIGITAL FILTERS

The fast algorithm of Little ~~for~~ implementing digital filters described in Section 2.5, turns out to be quite efficient, and practical to use. However, the hardware needed is demanding. Zohar [8] tried to apply the basic principle of the counting digital filter, discussed in Section 3. He introduced a RAM that will store the table of all possible products. In fact, the RAM filter has aspects of the counting filter, Little's algorithm and the approach of

Peled and Liu, as shown in Section 2.2.

Rewriting Equation (2.6) as

$$Y_n = \sum_{i=o}^{N} a_i X_{n-i}$$

then, as shown in Section 2.4, we can express Y_n as

$$Y_n = \sum_{j=1}^{B} F_{n,j} (-2)^j \qquad (2.78)$$

where

$$F_{n,j} = \sum_{i=o}^{N} a_i x_{n-i,j} \qquad (2.79)$$

Here, B is the word-length of the input in bits, and the input is coded as a word of N bits in base (-2). This is done to allow the representation of negative numbers.

Hence

$$X_n = \sum_{j=o}^{B-1} x_{n,j} (-2)^j \qquad (2.80)$$

where

$$x_{n,j} \varepsilon \{0,1\}$$

Again, by a similar derivation to that of Section 2.5.1, we can represent the table, $F_{n,j}$ as:

$$C(K) = C(K_{n,j}) \equiv F_{n,j}$$

$$= \sum_{i=o}^{N} a_i x_{n-i,j} \qquad (2.81)$$

where

$$K_{n,j} \equiv K = \sum_{i=o}^{N} x_{n-i,j} \; 2^i \qquad (2.82)$$

Therefore, if the table C(K) is stored in a RAM, the address needed to read a partial output K, has to be generated again and again, for every delay input. As Equation (2.80) indicates, this address is nothing but a binary word composed by taking the j^th bit from each input delay and that of the present output. Furthermore, as Little's algorithm shows, the next address can be found by shifting the present address and storing in the least significant bit the j^th bit of the present input.

To implement a non-recursive RAM digital filter, the analog input has to be converted into a binary input in base (-2). This is then fed, bit by bit, the most significant bit leading, into a large shift register of length MN bits. The first, B^th, 2B^th, NB^th bits are tapped and used to form a word of N+1 bits. This word is the address of the RAM. To generate the next address, shift the contents of the shift register one bit and feed in the next bit of the present input. Every time an address is generated, the addressed value in the RAM is the $F_{n,j}$ needed to compute Y_n, as in Equation (2.78). This addressed value, is accumulated in a shift register-accumulator. Each time a new value is added, the content of the accumulator is shifted towards the next significant bit, thus multiplying the partial sum by 2. This operation has to be done N+1 times. Notice that the accumulator is a standard binary adder (base + 2). This is because the table C(K), stored in the RAM is irrelevant to the input coding. The non-recursive RAM digital filter is depicted in Fig. 2.28.

The shifting process in the input register provides the new addresses very simply. Tapping the particular bits of this register is all that is needed. Fig. 2.29 explains the tapping and shifting needed to generate different addresses.

The recursive digital filter can be implemented by noting that it can be split into two non-recursive digital filters. Thus, the input is used to address a certain RAM, and the output also would be summed. Recalling Equation (2.68), the recursive digital filter is characterized by

$$Y_n = \sum_{i=o}^{N} a_i X_{n-i} + \sum_{i=1}^{N} b_i' Y_{n-i}$$

$$= Z_x + Z_y \qquad (2.83)$$

where

$$Z_x = \sum_{i=o}^{N} a_i X_{n-i}, \quad Z_y = \sum_{i=1}^{N} b_i' Y_{n-i}$$

Therefore, the implementation of this filter can be obtained by storing all the entries of each table needed to compute Z_x or Z_y in a RAM, then address it by generating the addresses by tapping bits from input or output values. Notice

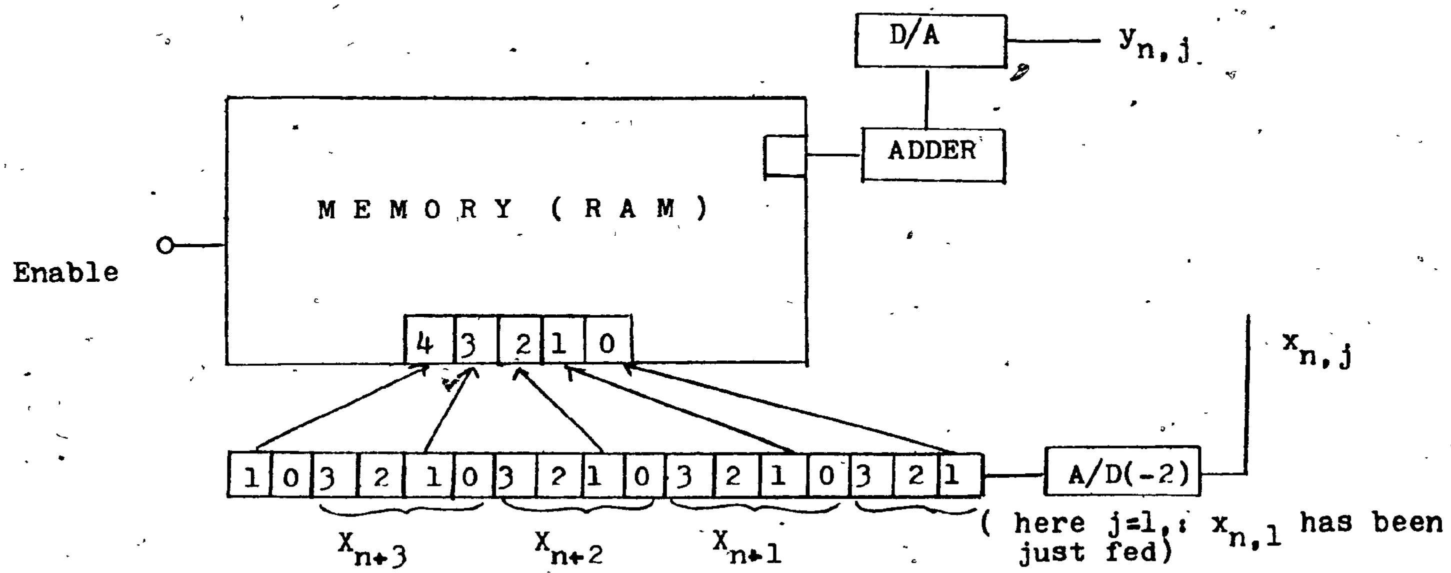


Figure 2.28 The RAM nonrecursive filter .

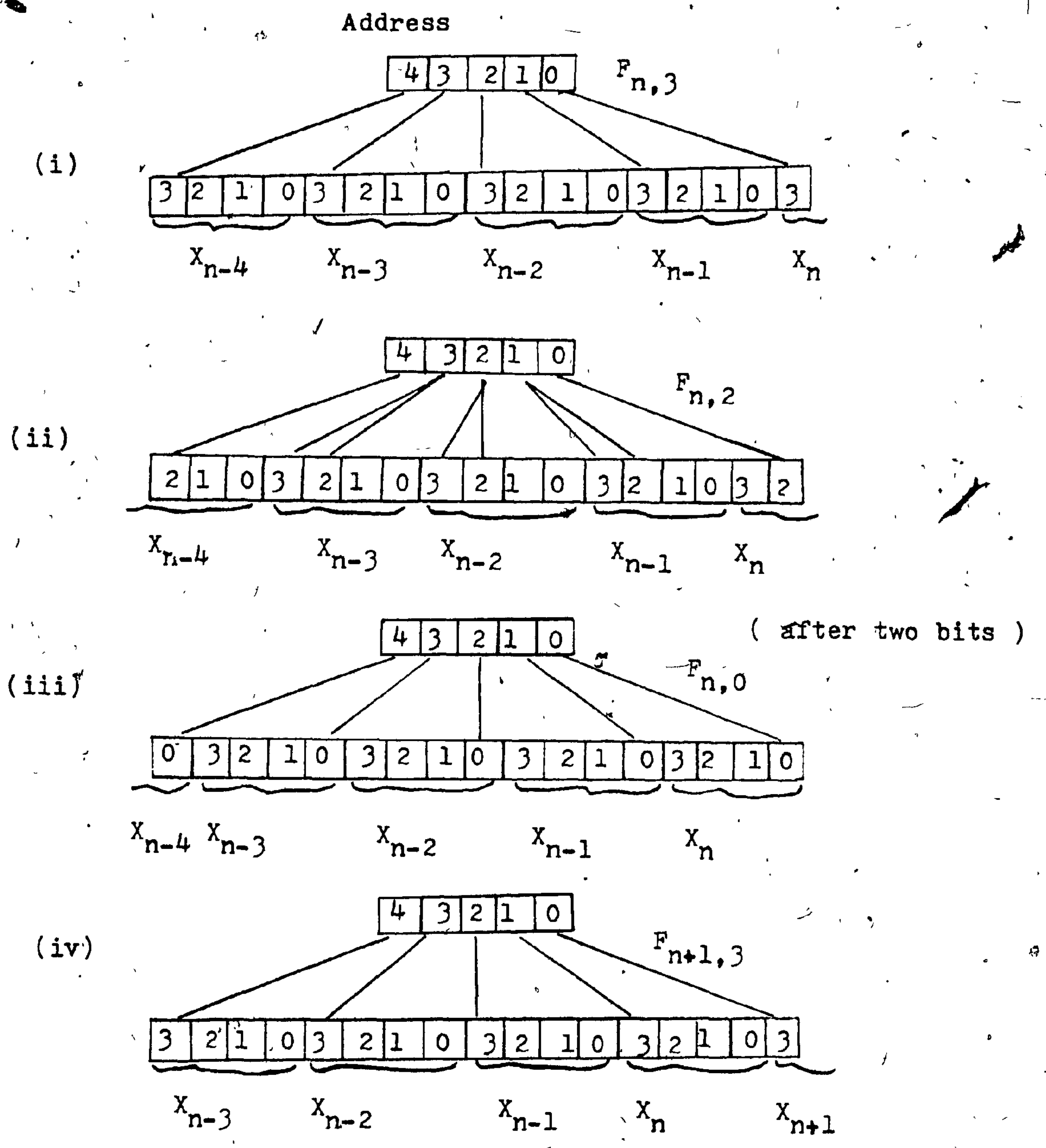

Address
$F_{n,3}$
(i)
X_{n-4} X_{n-3} X_{n-2} X_{n-1} X_n
$F_{n,2}$
(ii)
X_{n-4} X_{n-3} X_{n-2} X_{n-1} X_n
(after two bits)
$F_{n,0}$
(iii)
X_{n-4} X_{n-3} X_{n-2} X_{n-1} X_n
$F_{n+1,3}$
(iv)
X_{n-3} X_{n-2} X_{n-1} X_n X_{n+1}

Figure 2.29 Shifting and generating a RAM address

that the output, Y_n is taken from the accumulator and is converted to negative radix representation before it addresses the Z_Y RAM. Fig. 2.30, shows the RAM recursive digital filter by using two RAMs.

One other approach is to store both tables in one RAM, using only one address. This address is given by

$$K_{n,j} = Y_{n-N,j}, \ldots, Y_{n-1,j}\, X_{n-N,j}, \ldots, X_{n,j}$$

Therefore, the output is converted to the negative radix representation and fed into the output general shift register to be tapped for the new address.

This implementation is shown in Fig. 2.31.

2.7 RESIDUE NUMBER SYSTEM IMPLEMENTATION

Among the different number systems there exists an unusual one which has some interesting features, for example, it leads to high-speed multiplication and addition. Although division and sign detection are done at a slower rate in this system, it can still provide a very good implementation of digital filters. This number system is the residue number system (RNS).

Residue arithmetic was introduced by Szabo and Tanaka [13] and was later applied to filter implementation by Jenkins and Leon [14]. It depends basically on choosing a finite

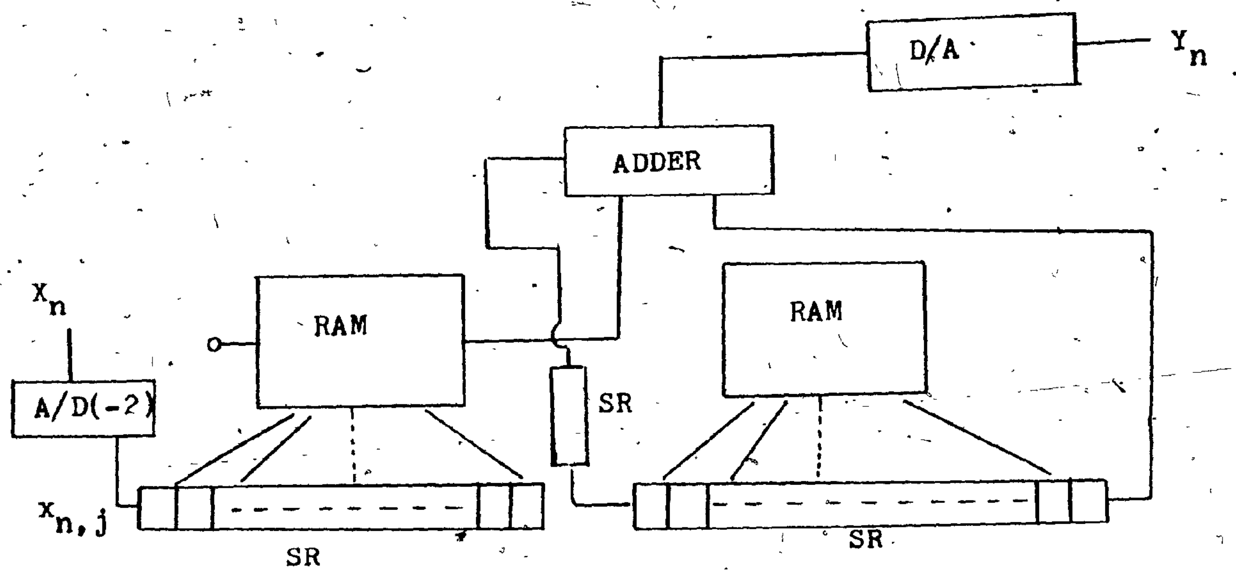


Figure 2.30 A recursive RAM filter using two RAMs.

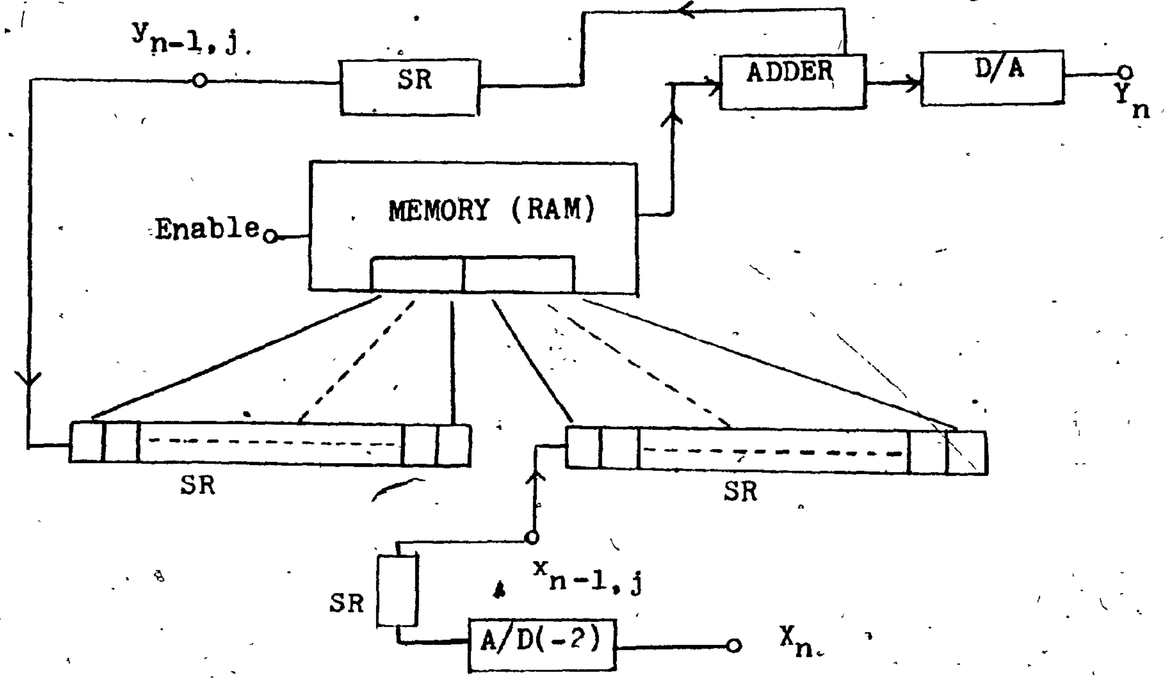


Figure 2.31 A RAM recursive filter.

Galois field which contains integers. A process of coding the input and coefficients is needed before the residue arithmetic is performed. The output also has to be decoded by employing the Chinese remainder theorem.

2.7.1 The Algebra of Finite Fields

Let 0,1,2, ..., N be the set of all the N+1 integers from 0 to N, and let $p = N+1$ be a prime number. Define also two operations, the p-modulus addition and the p-modulus multiplication, as follows:

$$S = A * B$$

where

$*$ is either multiplication or addition, and

$$\begin{aligned} S_{\text{mod-}p} &= |S|_p \\ &= \text{mod-}p(A * B) \\ &= |A * B|_p \\ &= \text{residue of } (A * B) \text{ at } p. \end{aligned}$$

These two operations and the set of integers, 0,1, ..., N form a finite field which has p finite states. This field, F(p), is the basis of the residue arithmetic.

Let m be an integer, such that

$$m \in F(p), \quad \text{i.e.,} \quad m \in \{0,1, \ldots, p-1\}$$

Then, the multiplicative inverse of this number is defined as the least multiple of m which has a p-modulo of 1. For example, let p=11, m=6, then the multiplicative inverse of m, denoted by m^{-1} is

$$m^{-1} = \frac{(p.i+1)}{m}$$

$$= \frac{(11.i+1)}{6} \quad , i\ 0,1,2, \ldots \quad (2.84)$$

The smallest i that satisfies the condition that m^{-1} should be an integer is 1, which gives $m^{-1} = 2$, as the multiplicative inverse of 6 in (11).

In general, the multiplicative inverse can be written as

$$\left| m^{-1} \right|_p = \left| \frac{(i \cdot p+1)}{m_p} \right|_p \qquad (2.85)$$

The RNS uses a set of numbers as moduli and on which other numbers are acted upon. Let $P = p_1, p_2, p_3, \ldots, p_L$ be a set of relatively prime numbers, integers, then any integer $I \in [-w,w]$ can be uniquely represented in a residue number system that is defined by the moduli set P. This representation is a set of residues which are the residues of I at the moduli in P. The interval defined by w is dependent on the set P, such that w is defined by

$$w = \frac{1}{2}\left[\sum_{i=1}^{L} p_i - 1\right] \qquad (2.86)$$

The residues used to represent I are $R_1, R_2, \ldots, R_L$, in other words,

$$I = (R_1, R_2, \ldots, R_L)$$

where

$$R_i = \begin{cases} I \text{ mod-}p_i & I \in [0,w] \\ p_i - I \text{ mod-}p_i & I \in [-w,0] \end{cases} \qquad (2.87)$$

$$i=1,2, \ldots, L$$

The arithmetic of the RNS is valid within the operations defined for the fields of each p_i. Hence, addition in the RNS applies the addition operation pairwise to the corresponding residues in the residue representations of the two integers involved in the operation. Multiplication is done in the same way, i.e., by multiplying the corresponding residues of the two integers. In general, for an operation defined as *, and for any two residue representations of any two integers, r,s, where

$$r = R_1, R_2, \ldots, R_L$$

and

$$s = S_1, S_2, \ldots, S_L$$

then for an operation defined, and operating on r and s:

$$r*s = (R_1, R_2, \ldots, R_L) * (S_1, S_2, \ldots, S_L)$$

$$= (W_1, W_2, \ldots, W_L) . \qquad (2.88)$$

where

$$W_i = (R_i * S_i)\text{mod-}p_i$$

The choice of the moduli set determines the dynamic range of the system. Actually, the computed results are identical to those obtained through the usual integer arithmetic, if the result is within the range $[-w,w]$. Noting that there is no rounding involved, the result can be obtained free from rounding error, provided enough dynamic range width is used. This can be done by choosing the moduli set large enough to give a large w, as in Equation (2.86). This gives a wide dynamic range. This, of course, requires more hardware.

The multiplication through modulus p can be stored as a function in a ROM for all possible values within $F(p_i)$, thus giving fast multiplications. The coding and decoding of a residue set will be the operation that should be performed before the operation of the filter is considered. The calculations of the output can be done in a parallel way, thus each residue of the set will be processed separately, in parallel circuits.

2.7.2 Encoders and Decoders

The input to the digital filter has to be converted from the binary to the residue number representation. This implies that the input has to be coded into a set of residues from a binary number. Szabo and Tanaka [13] suggested a method, in 1967, which turned out to be simple and easy to implement.

Consider an input X and let it be represented in the two's-complement form of a word-length of B bits. Let also the sign bit to be the B th bit:

$$X = \sum_{i=o}^{B} \beta_i \, 2^i \qquad (2.89)$$

where

$$\beta_i \varepsilon \{0,1\}$$

This value has to be coded as

$$X = X_1 X_2, \ldots, X_N$$

where

$X_j \varepsilon [0, p_j^{-1}]$ and p_j's are the N moduli of the RNS chosen.

Since

$$X_j \triangleq |x|_{p_j}$$

then from Equation (2.89)

$$x_j = |x|_{p_j} = \left| \sum_{i=o}^{B} \beta_i \, 2^i \right|_{p_j} \qquad (2.90)$$

Taking the residue of the term that represents the powers of 2, and separating the B th bit as the sign bit, we can rewrite Equation (6.7) as follows:

$$X_j = \left| \beta_B (p_j - |2^B|_{p_j}) + \sum_{i=o}^{B-1} \beta_i |2^i|_{p_j} \right|_{p_j} \quad (2.91)$$

Let $F_j(i)$ be a function that is defined as

$$F_j(i) = \begin{cases} |2^i|_{p_j} & i=o,1,2, \ldots, B-1 \\ p_j - |2^B|_{p_j} & i=B \end{cases} \quad (2.92)$$

With this definition, Equation (2.91) can be rewritten as

$$X_j = \left| \sum_{i=o}^{B} \beta_i \, F_j(i) \right|_{p_j} \quad (2.93)$$

Eventually, all the values of the filter function $F_j(.)$ can be stored in a ROM. Such a ROM can be addressed by the input bits,hence giving partial results. If those partial results are added in modulo-p_j adders, we get the jth residue of the input, as given by Equation (2.93). Furthermore, the speed can be improved by storing partial results of the function in different ROMs, thus by using several input bits to address several ROMs simultaneously, the time is saved.

The other process that has to be done before the filter operation is what to do to get the output back from the residue number representation. Certain decoders help in doing so, and those are implementations of certain algorithms or methods to convert a residue representation to a natural integer. The method discussed here is the most well-known one and it uses the Chinese Remainder Theorem.

Let $x \in [0,M)$

where

$$M = \prod_{j=1}^{N} p_j$$

and let $X_1 X_2, \ldots, X_N$ be the residue representation of x. The Chinese Remainder Theorem can be used to compute the value of x:

$$x = \left| \Sigma m_j \, |m_j^{-1} X_j|_{p_j} \right|_M \qquad (2.94)$$

where

$m_j = \dfrac{M}{p_j}$ and m_j^{-1} is the multiplicative inverse of $|m_j|_{p_j}$ within $F(p_j)$, defined by Equation (2.85).

Notice that M is a large integer which demands relatively large amounts of hardware to perform multiplication or addition in modulo-M. However, the terms $|m_j^{-1}|_{p_j}$ are fixed parameters and do not change with input. Therefore they can be premultiplied and the results can be stored as a modified filter function in a ROM.

Let $$Q_j = |m_j^{-1} \cdot X_j|_{p_j} = \sum_{k=o}^{s} b_{jk} \, 2^k$$

where

s is the largest of the word lengths of the moduli p_j, and $b_{jk} \in \{0,1\}$.

Equation (2.94) reduces to

$$X = \left| \sum_{j=1}^{N} m_j \sum_{k=0}^{s} b_{jk} 2^k \right|_M$$

$$= \left| \sum_{k=0}^{s} 2^k \{ \sum_{j=1}^{N} m_j b_{jk} \} \right|_M$$

$$= \left| \sum_{k=0}^{s} 2^k F(k) \right|_M \qquad (2.95)$$

where

$$F(k) = \sum_{j=1}^{N} m_j b_{jk} \qquad (2.96)$$

Such a function uses binary bits of each residue of the N moduli and produces a certain coefficient of the power-of-2 polynomial, which when evaluated, gives the output in a natural integer form.

2.7.3 Implementation of Digital Filters

The application of RNS coding to digital filters can be very desirable in many cases, especially in real time-processing of digital signals at high data rates. The minimization of round-off error is easily achieved in such implementations, as well.

There are certain considerations that determine the efficiency of a residue number digital filter. Any implementation should be designed such that these considerations be met properly, hence producing an efficient and desirable filter. Of these considerations:

1) The dynamic range of the filter determines the moduli set to be chosen. This range can be set to $[-w,w]$, where

$$w = \frac{1}{2} \left(\prod_{i=1}^{N} p_i \right) - 1$$

and the p_i's are the moduli set. The magnitude of the p_i's determine the word-lengths of the individual residue representations of the input and the output. The order of the set, N, determines the number of channels to be used for residue representations.

Notice that the p_i's should be prime integers, or at least, prime to each other, so as to have a unique residue decoding to a natural integer, i.e., the set of moduli should be a finite field and not a finite ring, so as to be able to use the Chinese Remainder Theorem, so as to decode a residue representation uniquely to a natural integer.

2) The choice of the moduli set determines the magnitude of the quantization error of the residue number representation of the filter coefficients. Therefore, the larger the quantization word-length is, the higher the accuracy of the frequency response of the filter becomes. Usually, eight to ten bits would yield a good quantization accuracy.

3) The filter function can be stored in PROMs rather than ROMs, thus providing for more flexibility in the design. Besides, the residue representations can be split into different ROMs or PROMs, of small size and addressed simultaneously to get partial results, rather than using large ROMs to be addressed by a larger subset of the residue representation. The partial results can be combined properly to give the final result, thus decreasing noticeably the size of the ROMs needed.

4) The speed of operation is determined by the number of parallel paths the filter has. The use of RAMs can speed up the process and thus the filter operates on a higher data rate.

The design, itself, is straight forward. Fig. 2.32 shows a residue number implementation of a digital filter using a moduli set of five relatively prime integers: 7,9,11,13,16. The input X(n) is fed to five different PROMs which would

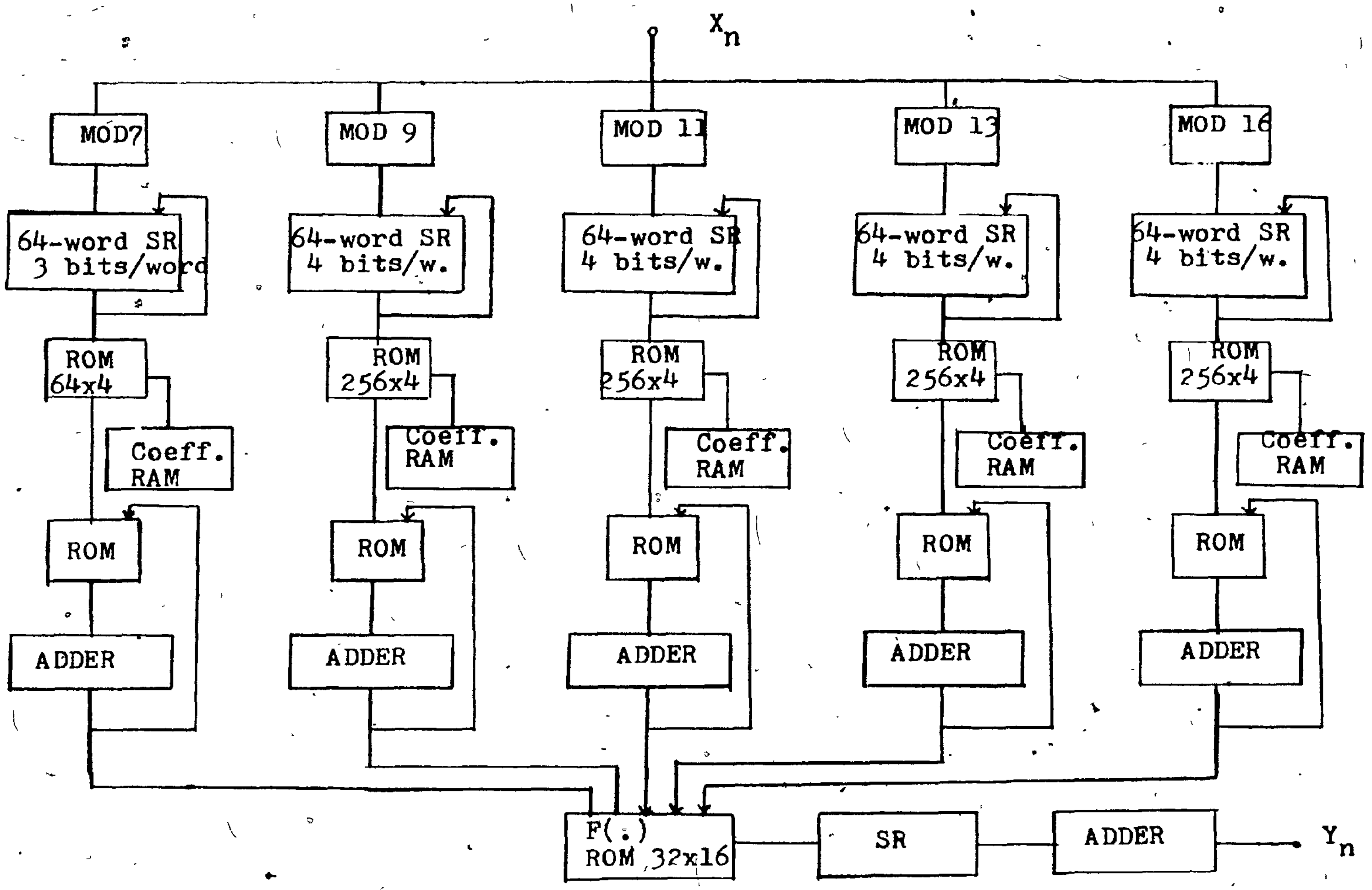


Figure 2.32 A Residue Number Implementation of digital filters. 5 moduli are used : 7,9,11,13,16 .

produce the residue representations of the input as $X_1X_2X_3X_4X_5$. Shift registers are used to keep the incoming bit representation and produce words which will be used to address a RAM that will contain the filter coefficients. The overall result, up until now, is the partial product of the input bits of the particular residue in concern. This now can be used to address the filter function which is stored in a ROM, thus giving the total product of the input modulo p_i. Accumulating those results in an accumulator for all corresponding bits of each residue, then shifting and adding the results in decoding the output from the residue number representation to binary representation.

However, further hardware simplification can be achieved by shifting the "distributing" shift register ahead of the parallel modulo-p_i ROMs, thus saving more hardware components. Fig. 2.33 shows this.

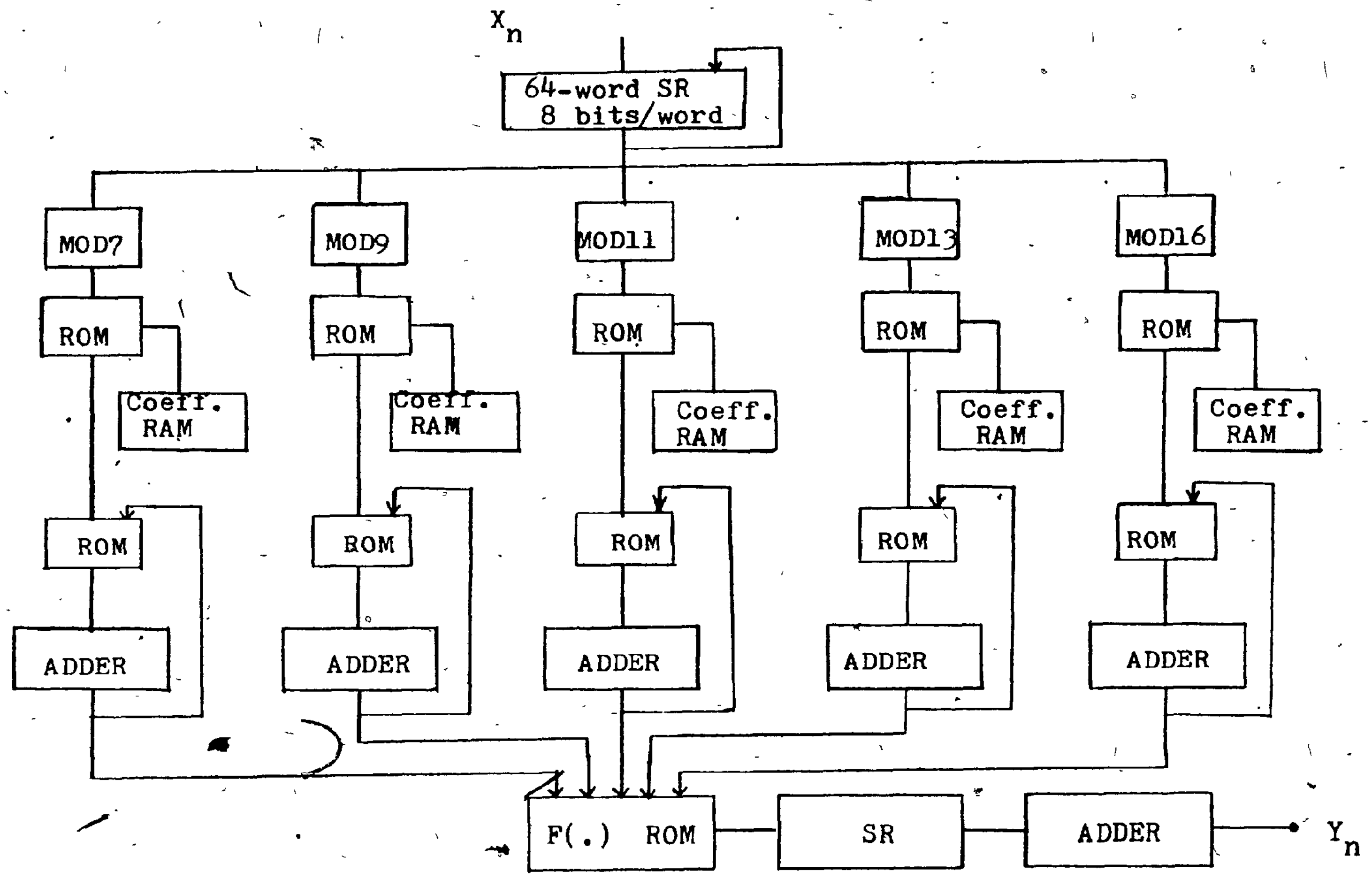


Figure 2.33 Another RNS implementation of filters.

CHAPTER III

FURTHER ASPECTS OF IMPLEMENTATIONS

3.1 INTRODUCTION

The implementations discussed in Chapter II can be extended and varied in different ways. Such variations constitute the basic layout for more efficient implementations. In this Chapter, some variations on the implementations discussed in Chapter II are analysed. Some related aspects are also discussed. Those aspects include scaling, convolution and hardware considerations.

3.2 TECHNIQUES IN RESIDUE NUMBER SYSTEM IMPLEMENTATION

Residue arithmetic proved to be a very promising tool in the implementation of digital filters. The theory, in fact, is generalized to include residue algebra of rings and finite Galois fields. To implement digital filters using residue number systems (RNS), data and coefficients should be transformed into integers by using scaling. Scaling might be a problem, but still there are effective scaling algorithms that work agreeably [15]. Furthermore, new methods in implementing addition and subtraction in RNS would result in a more efficient implementation.

3.2.1 Scaling

RNS does not allow fractions, therefore, dividing a residue number by a constant and quantizing the result to an integer is a difficult problem. However, Jenkins [16] has suggested an algorithm which is easier to implement. The Chinese remainder theorem helps to convert back an RN representation into a normal integer.

Let L moduli be chosen as $m_1, m_2, \ldots, m_L$, and let $y_i(n) = [y(n)]_{m_i}$, $i=1,2, \ldots, L$. $Y(n)$ can be computed as

$$Y(n) = [\sum_{i=1}^{L} \bar{m}_i |\bar{m}_i^{-1} y_i(n)|_{m_i}]_M$$

where

$$\bar{m}_i = \frac{M}{m_i}, \quad |\bar{m}_i \bar{m}_i^{-1}|_{m_i} = 1$$

and

$$M = \prod_{i=1}^{L} m_i \qquad (3.1)$$

The computation of $Y(n)$ can be simplified by multiplying $\bar{m}_i$ in many ways. If the filter has constant coefficients that are permanently stored, the $\bar{m}_i^{-1}$'s can be premultiplied by them and stored. It can also be premultiplied into the multiplication tables stored, or by the translation codes, if encoding and decoding are done by stored translation codes.

3.2.2 Hardware Structures

The implementation of residue recursive digital filters is highly modular and structured. Three concepts of this hardware structuring are the following:

a) Residue coded combinational filter:

Recalling the ROM implementation method of Croisier, Peled and Liu [4], a second-order section can be used as a subfilter. Three moduli can be chosen as $2^k, 2^k-1, 2^{k-1}-1$, where k is an integer. Notice that the choice of these three moduli $L = 3$, will have a convenient scaling factor of $2^k(2^{k-1}-1)$. Those three subfilters are then implemented and connected in parallel. This is shown in Fig. 3.1.

b) Small moduli combinatorial filters:

If the moduli set consists of small integers, then the output $y_i(n)$ can be expressed as

$$Y_i(n) = [Y(n)]_{m_i} = F_1[(F_2(|X_{n-1}|_{m_i}, |X_{n-2}|_{m_i}) + F_3(y_i(n-1), y_i(n-2)), |X_n|_{m_i}]]$$

where

F_1, F_2 and F_3 are filter functions which depend on the coefficients.

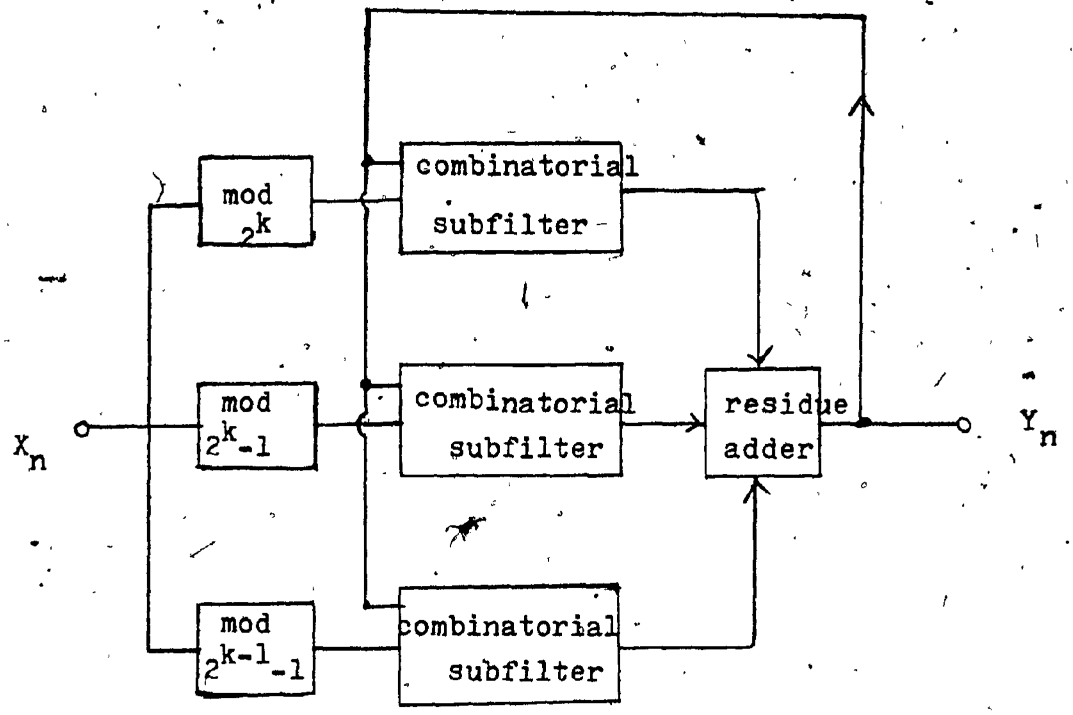


Figure 3.1 . Architecture for a combinatorial recursive filter in residue number system.

Figure 3.2 depicts this implementation. This is a high-speed and high-precision implementation, since the sub-filters operate simultaneously, and no bits are lost through the use of general scaling circuitry.

c) Scaling circuitry

For filters with varying coefficients, i.e., adaptive filters, the filter function has to be computed every time they change. An alternative is to use conventional multiplier ROMs that are more flexible for such variations.

3.2.3 Other RNS considerations

Filters that are implemented by RNS and combinatorial methods are referred to as hybrid, such as shown in Figures 3.1 and 3.2. Hybrid implementation, in fact, is highly desirable, due to the favourable qualities of both combinatorial and RNS implementations.

If the order of the filter is $N=LM$, where L,M are integers, then the filter can be implemented, using M sections with an $L^{\underline{th}}$ - order subfilter in each. With $M \leq 10$, combinatorial implementation is suitable, and hence it can be used. L subfilters can be connected in parallel to give the partial output of that section.

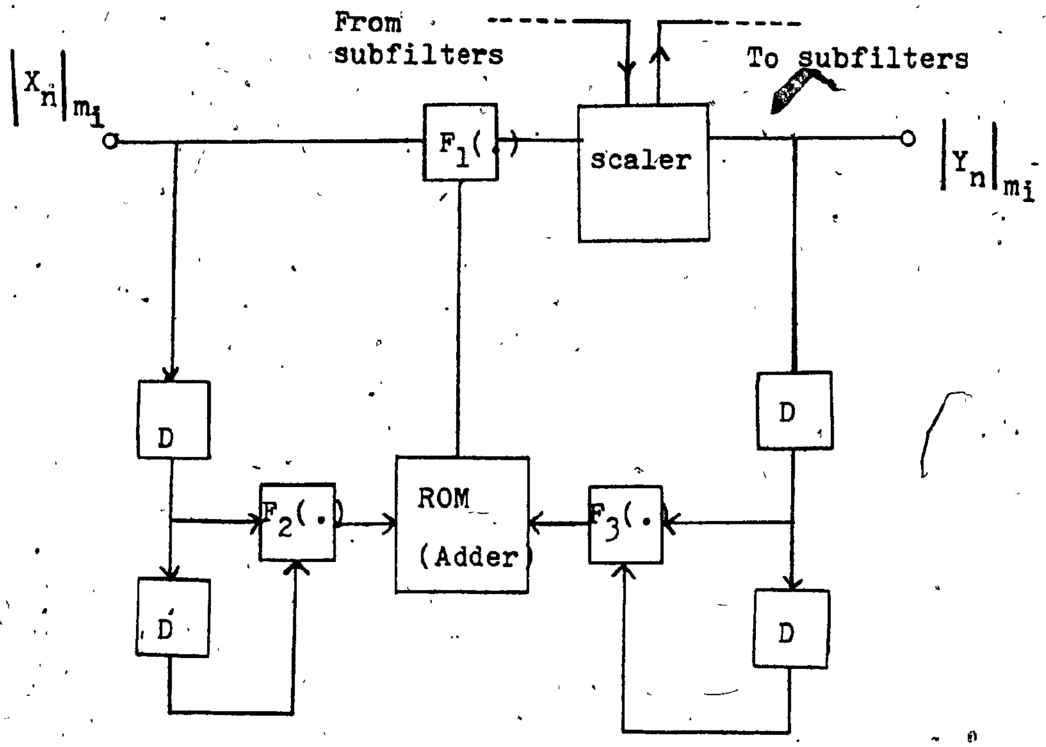


Figure 3.2 Architecture for a high-speed combinatorial filter in a general residue system with 4-bit moduli.

This is fed to the next section and so on. The subfilter itself, is, in fact, a simple combinatorial ROM filter. This is shown in Figure 3.3.

This implementation provides a filter which has data rate reasonably independent of both coefficient and data word length, because increased precision can be obtained by adding residue subfilters in parallel to the original structure.

Soderstrand [17] suggested a RNS implementation for a low-cost high-speed digital filter. In this implementation, blocks are used to construct the filter structure and each block handles a certain function. In Figure 3.4, a digital filter with three moduli is implemented. The moduli are 4,7,15. The block labelled BRNS (Binary to RNS) converts an input data sample from binary to RNS representation, while the RNSB (RNS to Binary) does the opposite. Each MOD block is a modulo adder with delay feedback which does everything, delay, add and look-up for precomputed multiplications. MUL are designed to perform a table look-up for fractional multiplication. Addition and subtraction of RNS codes are the basic operations often required in filter implementation. A good implementation of such adders is due to Bannerji [18].

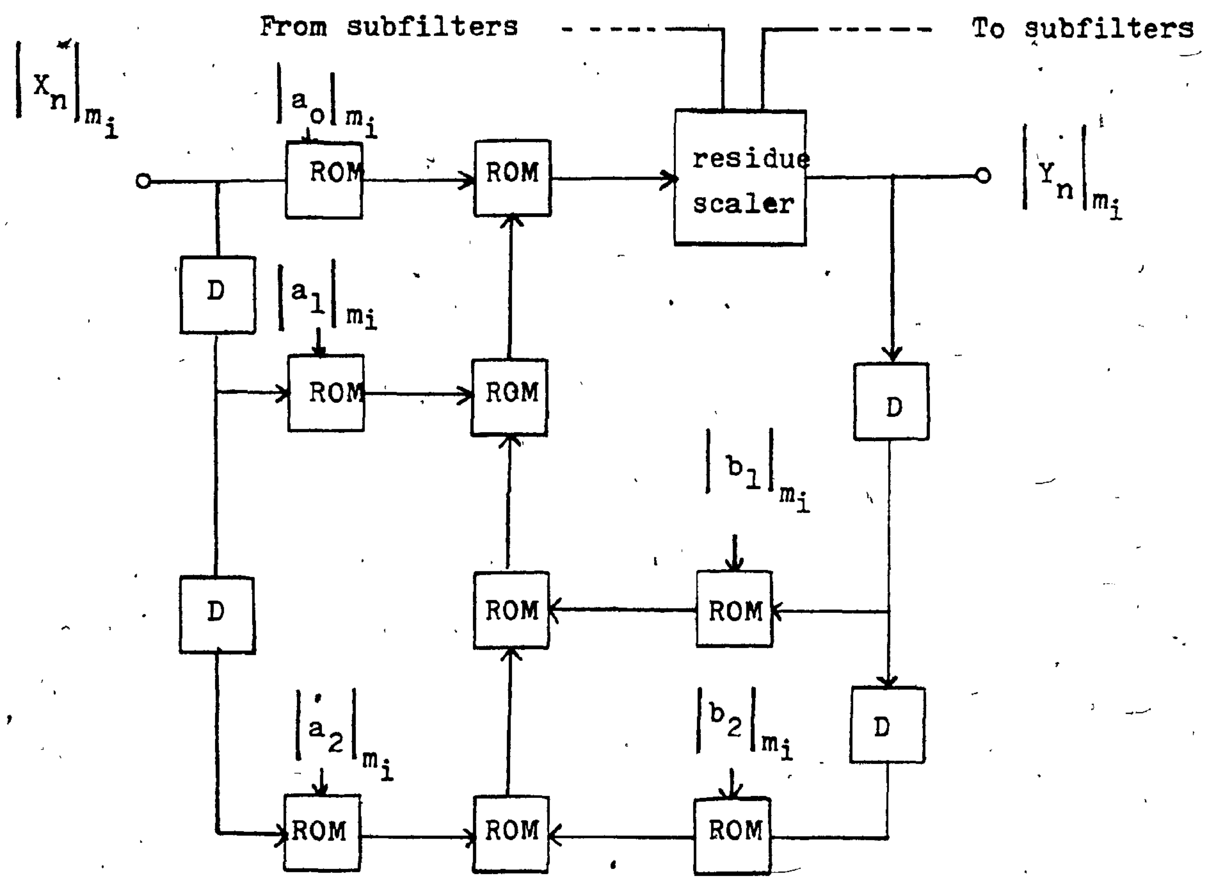


Figure 3.3 Architecture with general ROM multipliers for an adaptive second-order section.

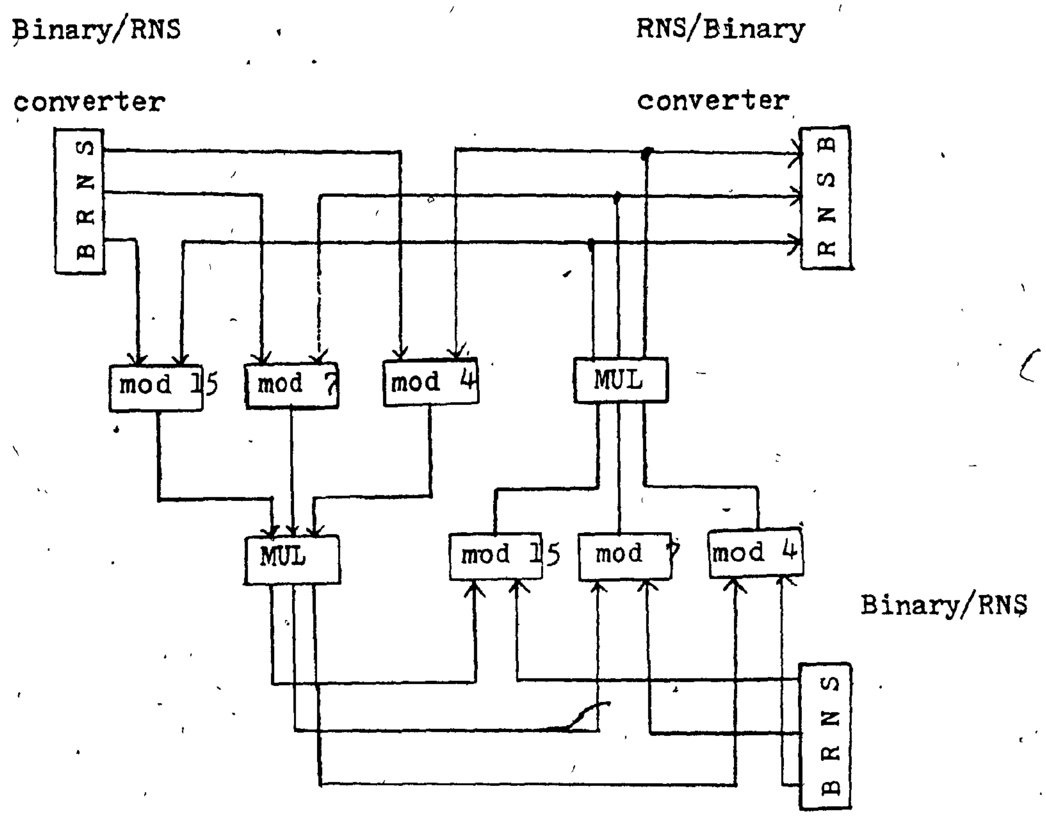
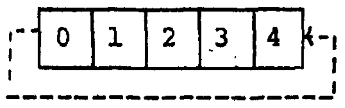


Figure 3.4 A residue number system second-order section based on Soderstrand's approach.

One important property of any residue number system is that in any operation, addition, subtraction or multiplication, any digit of the result is solely dependent or the corresponding digits of the operands. This means that no carry is needed from one residue digit to another. It is this property that is used, here. Notice that the set of residues are cyclic under addition modulo M. For example, if M=5, the addition table for $|X+Y|_5$ can be written as

x \ y	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3



If these residues are stored in a shift register, then to add two numbers, X,Y, the contents of the register are rotated to the left x times, then the y th cell in the register contains the sum of x and y. So, if x=3, y=4, then $|x+y|_5 = |3+4|_5 = 2$.

By shifting the contents of the shift register three times to the left, the contents would be

3	4	0	1	2

Eventually, the 4th cell contains the sum which is 2. The implementation of a mod-3 adder is shown in Fig. 3.5.

The subtraction can likewise be done by shifting the contents of a register which contains the subtraction residues. The residues for M=5 are 0,4,3,2,1. The shift, in this case, is performed x times, and at the end, the yth cell contains the difference X-Y. The resulting implementation for M=3 is shown in Figure 3.6.

3.3 <u>TECHNIQUES IN DIRECT IMPLEMENTATION</u>

There are certain aspects of the direct implementation methods that make them suitable for MSI or LSI. These aspects include modularity, high-speed, low-cost, accuracy, and so forth. Certain variations or changes in the basic implementation may result in better implementations. Two important variations are discussed here. The techniques usually used to obtain better implementations include maximizing the pre-computations, optimizing memory locations, multiplexing, and structuring the implementation.

3.3.1 <u>Cycling Filter</u>

The implementation of a filter using a counter was discussed in Section 2.3. Notice, that in Figure 2.20, the gates used produce outputs simultaneously. This is not

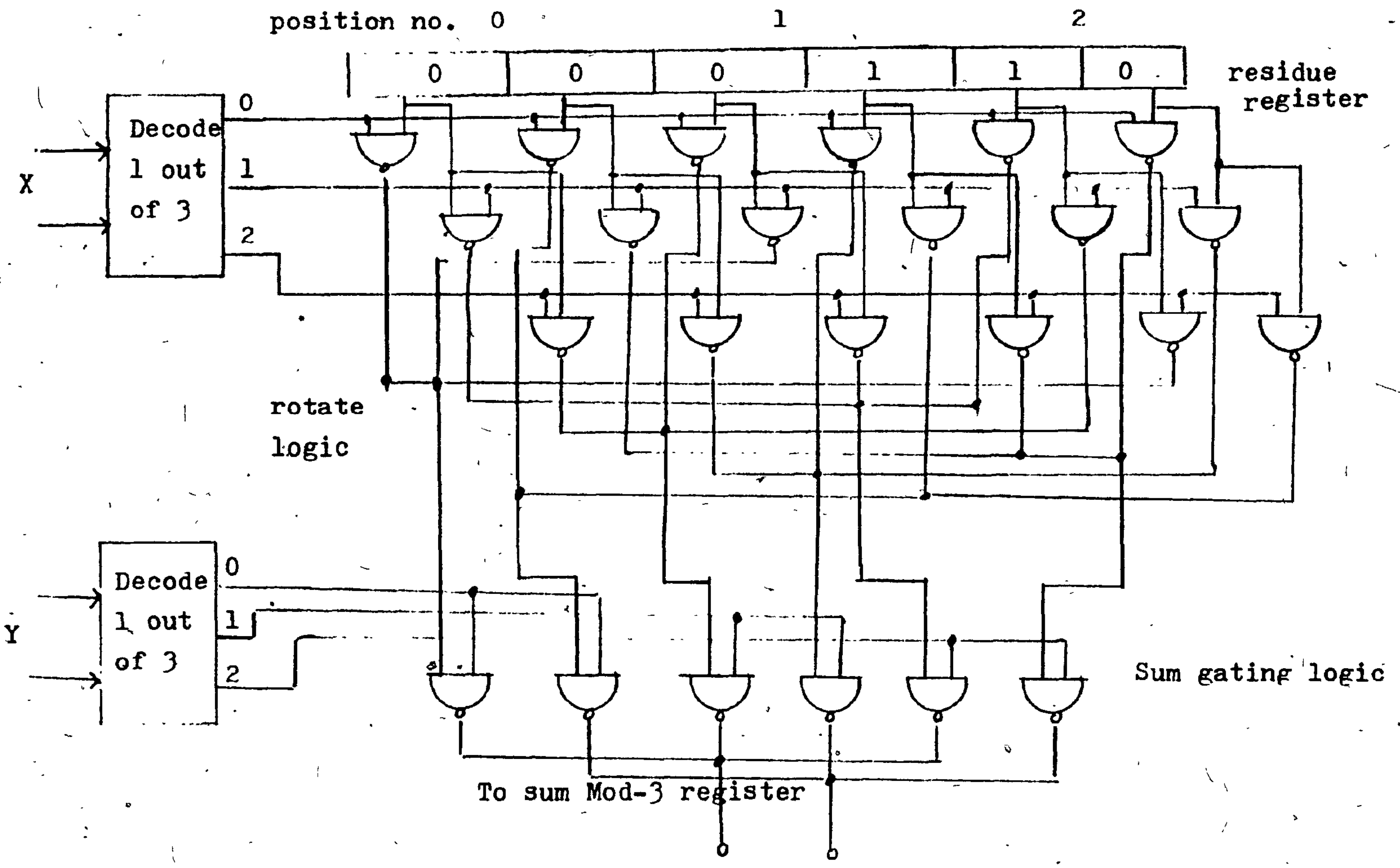


Figure 3.5 Modulo 3 adder realization using rotate logic.

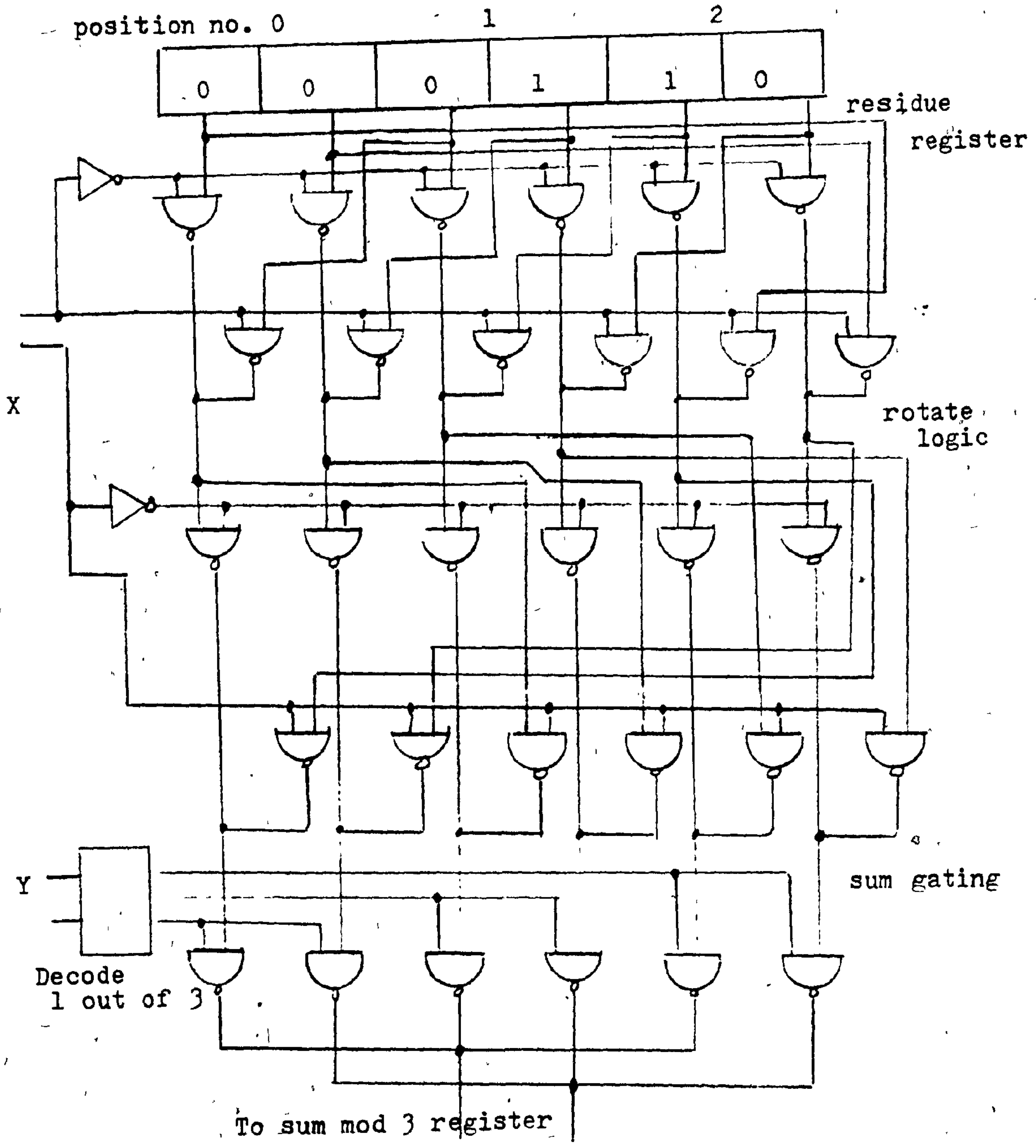


Figure 3.6 Alternative implementation of mod adder (mod-3) using rotation by power of 2.

needed in practice, since only one output of the gates is counted each time. Hence, one gate can be used to perform all the gating. This can be implemented by using a shift register such as a circulating line. Consider a circular shift register, which contains the coefficients, and another which contains the input data and its delayed values. This register has an entry point, where data gets into it at every sample period. During this period, both registers would shift in opposite directions at different pulses. Thus, each pair of bits is gated and is then fed into an up-down counter. This implementation of a non-recursive digital filter is depicted in Figure 3.7. It is extended to a recursive filter in Figure 3.8.

3.3.2 The Heute Filters

Another implementation which depends upon the shifting coefficients or data in registers, is that due to Heute [19]. This implementation depends on shifting completely the contents of registers in opposite directions such that one input word and the corresponding coefficients are multiplied together. This multiplier can be a serial arithmetic unit which is used in the conventional multiplier implementation, or a ROM which contains all the possible products of a B-bit coefficient and a data word.

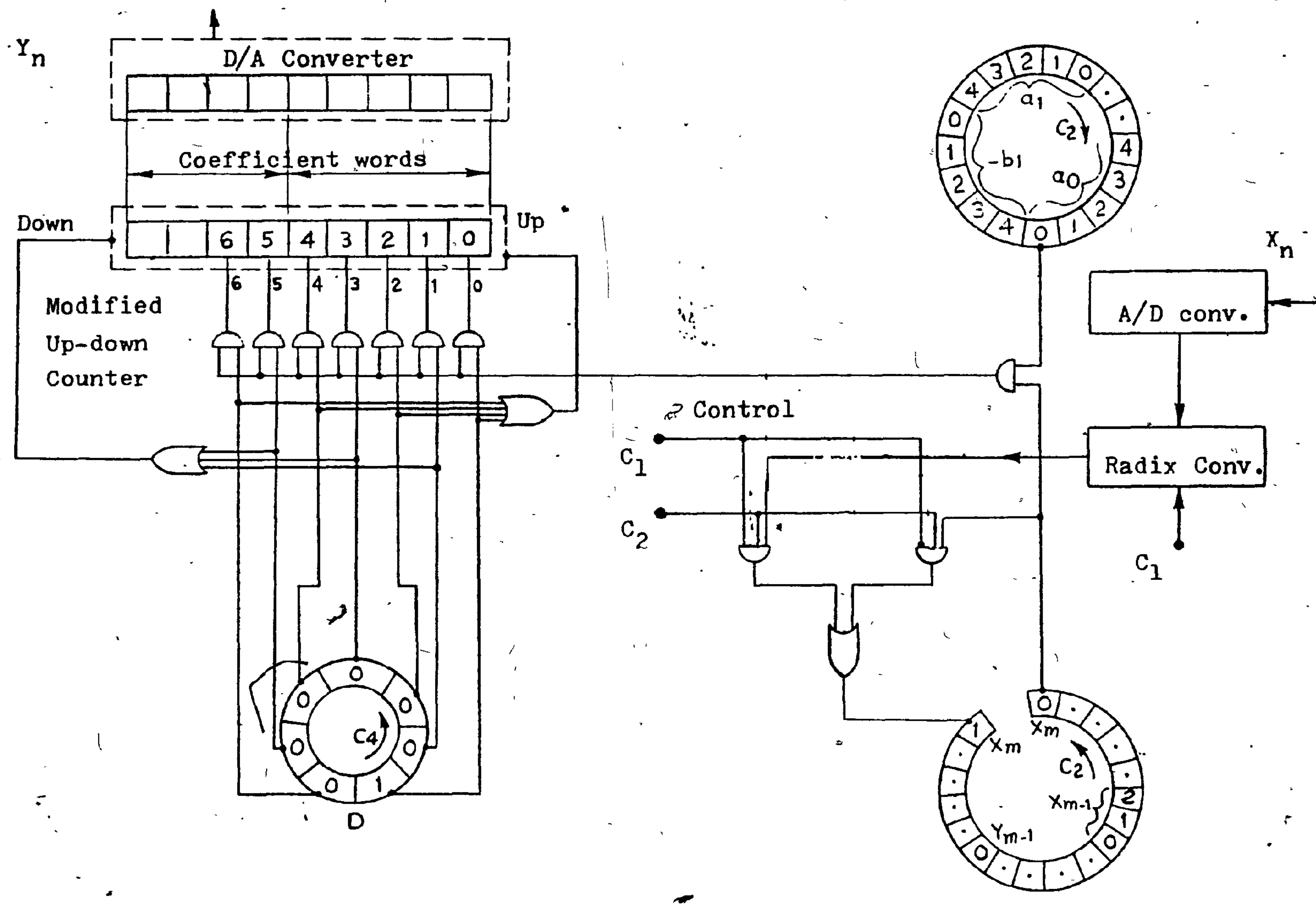


Figure 3.7 The cycling nonrecursive filter.A variation on the implementation of the nonrecursive filter using an up-down countec.

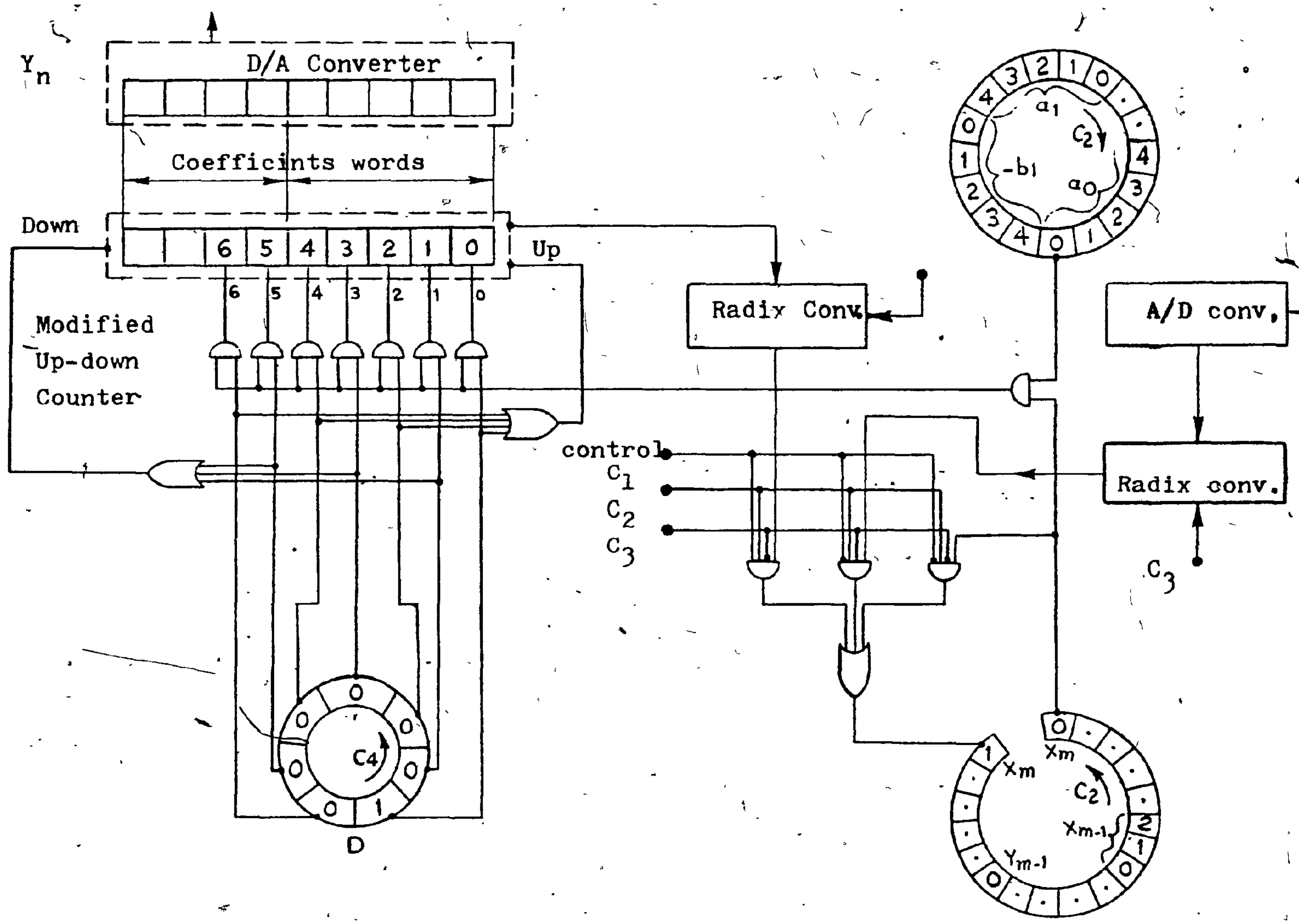


Figure 3.8 The cycling recursive filter. A variation on the implementation of the recursive digital filter using an up-down counter.

Two registers would contain the input data and the filter coefficients. These registers have to shift in parallel, and each bit is tapped on both registers to address the ROM and produce the product. Figure 3.9 shows an implementation, using a serial multiplier, whereas, Figure 3.10 shows an implementation using a look-up table in a ROM. This implementation becomes demanding in storage or in multiplier hardware for a large N. To eliminate this problems, so as to speed up the filter operation, parallelism is applied.

One form of parallelism is splitting the filter into two or more subfilters. Take $n_1 = [N/2]$, i.e., the integer part of half N, then implement the filter in one subfilter for the lower half, and in another subfilter for the upper half. The filter can be expressed by

$$y_n = \sum_{i=o}^{N} a_i X_{n-i} = \sum_{i=o}^{N_1} a_i X_{n-i} + \sum_{i=n_1+1}^{N} a_i X_{n-i}$$

$$= Y_{1,n} + Y_{2,n}$$

where

$$Y_{1,n} = \sum_{i=o}^{n_1} a_i X_{n-i}$$

and

$$Y_{2,n} = \sum_{i=n_1+1}^{N} a_i X_{n-i}$$

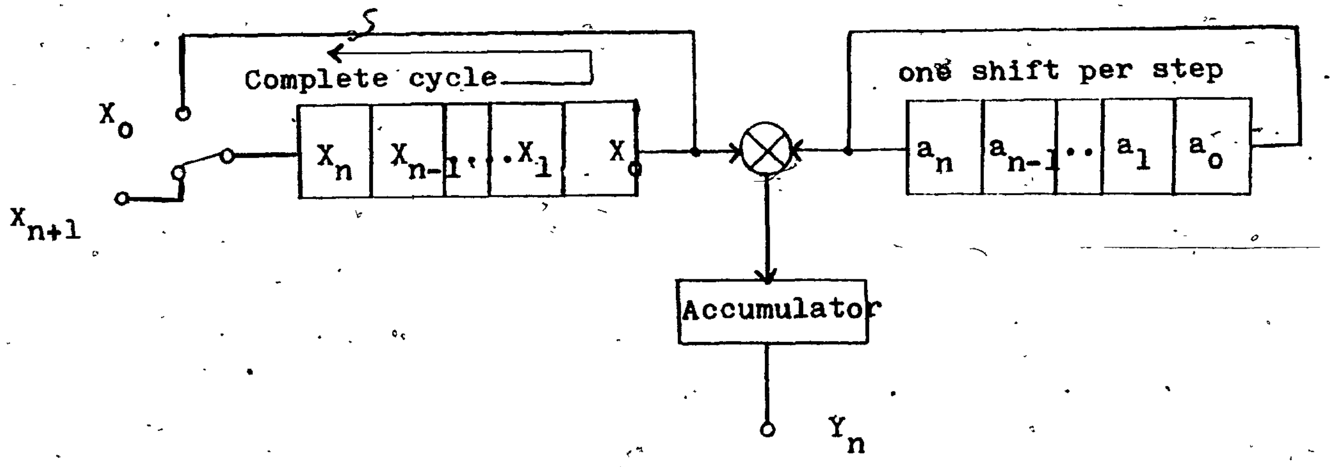


Figure 3.9 The Heute implementation using a multiplier.

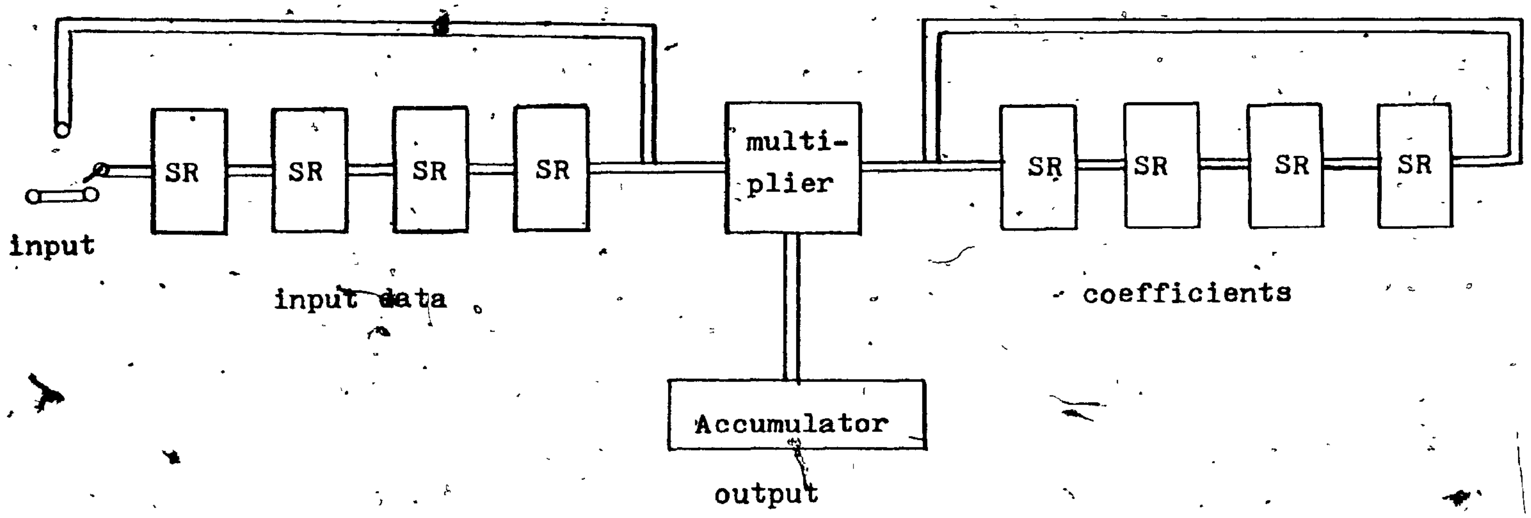


Figure 3.10 The basic structure for the Heute filters.

This implementation is depicted in Figure 3.11. This is very useful when the hardware used is too slow for the required sampling rate.

If the multiplier is slow, the filter can be speeded up by using two multipliers. As shown in Figure 3.12, X_n and a_n are tapped fed to the first multiplier, while at the same time, X_{n-1} and a_{n-1} are fed to the second multiplier. Both products are added to an accumulator. A shift of two words would follow, feeding X_{n-2} and a_{n-2} to the first multiplier and X_{n-3} and a_{n-3} to the second, and so forth.

One other variation is the one shown in Figure 3.13. This is useful when both the registers and the multiplier are slow. Each word is split into two parts and fed in parallel to two multipliers. Such an implementation is efficient for linear phase filters.

3.4 IMPLEMENTATION OF DIGITAL SIGNAL PROCESSORS

There are certain methods of calculating the output of a digital filter that require the evaluation of a transform of the input and the coefficients. The output is computed easily in the domain of the transform. This

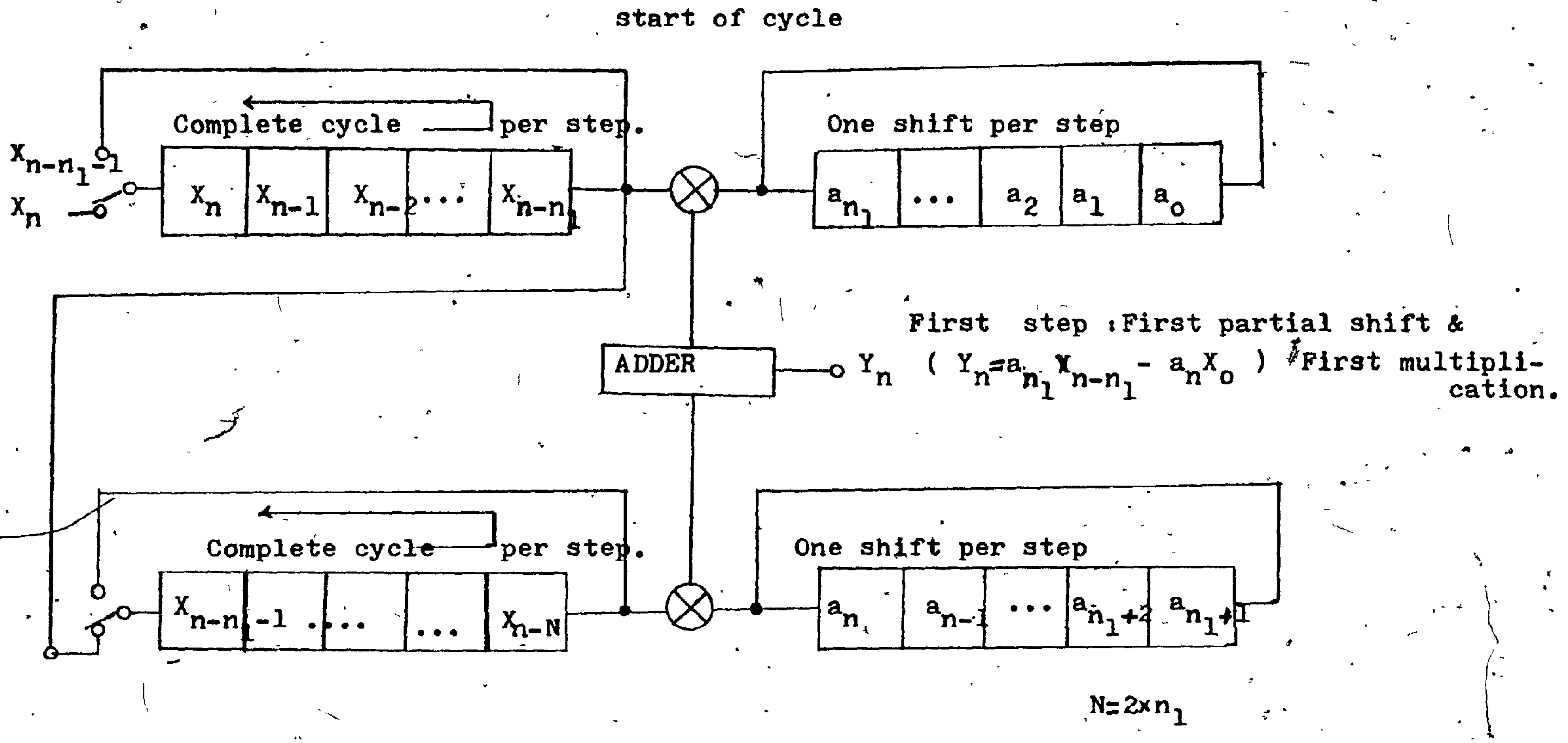


Figure 3.11 A parallel double section Heute filter.

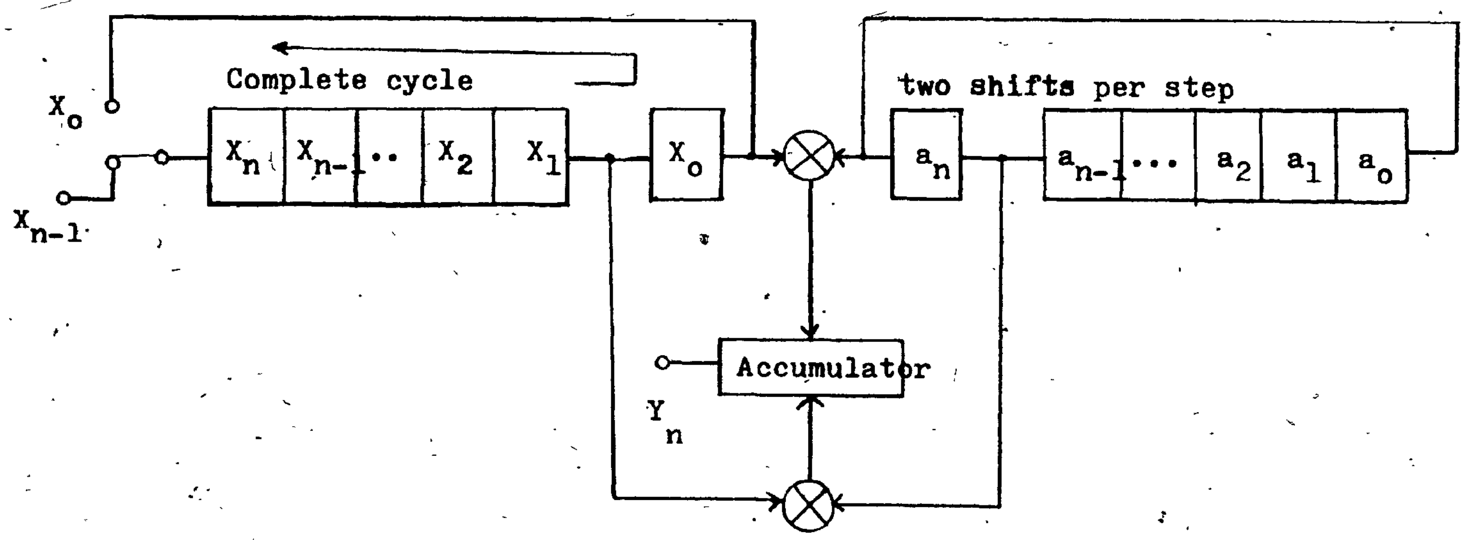


Figure 3.12 The Heute implementation using two multipliers.

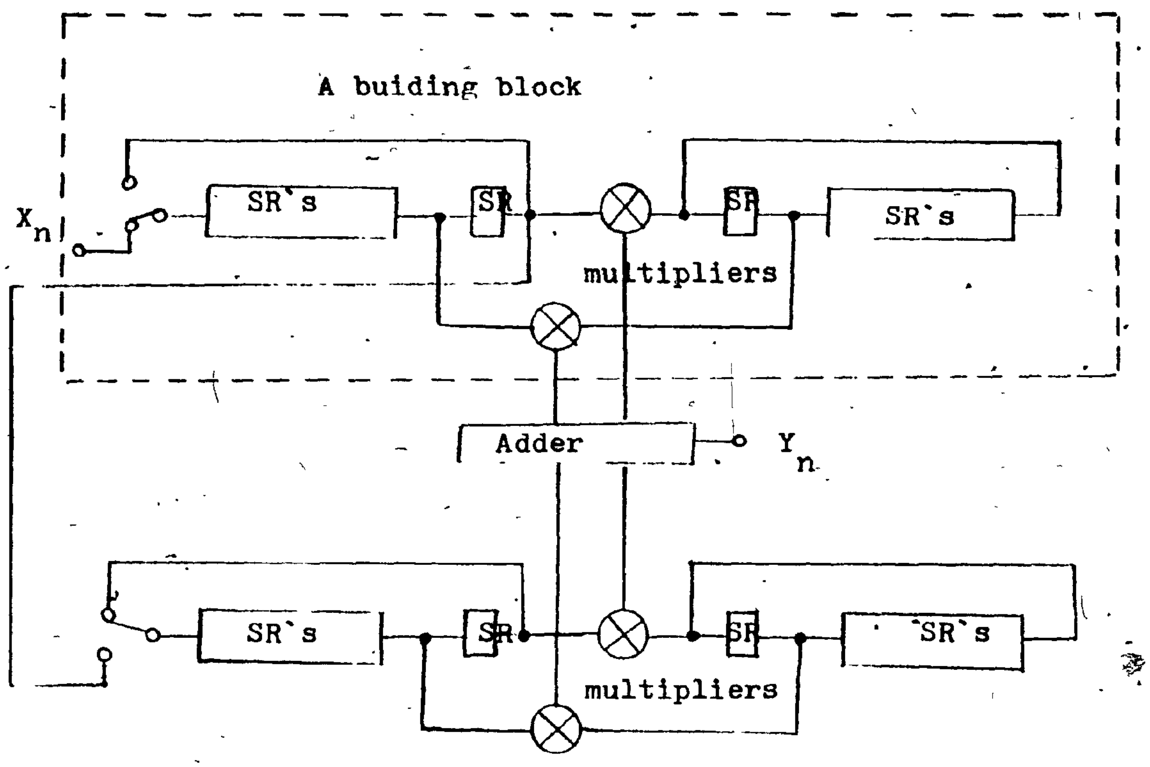


Figure 3.13 Combined parallelism : two multipliers and two general shift registers.

method is very good for finite-impulse digital filters. Such methods are useful for the evaluation of convolution, as well.

There are hardware implementations of certain processors that take the transform of the input and evaluate the output in the transformed domain. Such implementations are more complicated than the digital filter implementations, due to the fact that larger numbers of data points have to be dealt with at the same time. Even so, the transform methods can be very suitable and efficient for digital filter implementations. The abstract presented here, presents the general overview of FFT processor implementations which can be generalized easily to other transform processors, then summarizes the characteristics of the hardware architecture of these implementations.

3.4.1. Convolution

Convolution is a basic operation that involves multiplications of arrays. It is, in fact, the process of evaluating the output of a non-recursive digital filter. The complete input sequence has to be available, in order to compute the output sequence.

Convolution demands a lot of multiplications. There are many methods and algorithms that do the convolution in a lesser number of multiplications. Many of those algorithms depend upon the idea of transforming the data into a new domain. Such a transformation will lead to the reduction of the number of multiplications required. The output can be obtained by inverse-transforming the products back to the original domain. The other type of algorithms depend on numerical techniques not associated directly with a transform.

The Discrete Fourier Transform (DFT) [1] is one of the most important of all. It is the discrete version of the Fourier Transform. It is defined as

$$\bar{X}_k = \sum_{n=o}^{N-1} x_n W^{-kn} \qquad (3.1)$$

where

$$W = e^{2\pi j\frac{1}{N}} \qquad (3.2)$$

If the non-recursive filter is characterized by

$$Y_n = \sum_{i=o}^{N-1} a_i X_{n-i} \qquad (3.3)$$

$$n=0,1, \ldots, N-1$$

Then by taking the DFT of both sides, we get

$$\bar{Y}_n = \sum_{j=o}^{N-1} \left(\sum_{i=o}^{N-1} a_i X_{n-i}\right) W^{-nj}$$

$$= \left(\sum_{i=o}^{N-1} a_i W^{-nj}\right)\left(\sum_{j=o}^{N-1} X_{n-j} W^{-n(j-i)}\right) = \bar{A}_n \cdot \bar{X}_n \qquad (3.4)$$

Hence, by taking element-by-element product of the DFT of both the input sequence and the coefficients, we get the DFT of the output sequence. Taking the inverse DFT of $\bar{Y}_n$, we get the output Y_n. There are algorithms to compute the DFT, referred to as the Fast Fourier Transforms (FFT). Any implementation of a non-recursive digital filter by convolution has to consider the implementation of a digital signal processor that does the DFT and the inverse DFT.

There are other transforms which are useful. A particular one, called the Fermat Number Transform (FNT) provides a very good basis in implementing the convolution. This is discussed in Section 3.4.1.1. Other transforms that can be useful include Mersenne Number transforms [20], Walsh Transforms [21] and the Discrete Cosine Transform [22]. These transforms require complicated hardware, and are useful in particular cases, depending on the nature of signal that is to be filtered. One other method is discussed in Section 3.4.1.2, namely, the Cook-Toom algorithm for computing the convolution. This represents the other way of computing the convolution where no direct or specific transform is used.

3.4.1.1 Number theoretic transform

Consider any N by N nonsingular matrix T whose elements are $t_{k,m}$ for $k,m=0,1, \ldots, N-1$, and N is a power of 2. This matrix can be used to transform the input X_n and the coefficients A_n, as in Equation (3.10). Let this transformation be defined as

$$\underline{X} = T\,\underline{x}\ , \quad \underline{A} = T\,\underline{a}$$

$$\underline{Y} = T\,\underline{y} \tag{3.5}$$

where

$\underline{x}$, $\underline{a}$ and $\underline{y}$ are again N-element column vectors, whose elements are x_i, a_i and y_i, respectively. $\underline{X}$, $\underline{A}$ and $\underline{Y}$ are their N-element transformation vectors.

The matrix T can be chosen such that

$$\underline{Y} = \underline{X} * \underline{A} \tag{3.6}$$

where

* denotes element-by-element multiplication

Such transformations are called number theoretic transforms (NTT). The proper choice of the matrix T determines the complexity of the transform. A proper choice is

$$t_{k,m} = \alpha^{km} \qquad k,m=0,1,\ \ldots,\ N-1$$

and the inverse T^{-1} has elements $t_{k,m}$ as

$$t'_{k,m} = N^{-1}\alpha^{-km}$$

and the structure established by this choice is orthogonal. The transformation involves computations which are of the form

$$X_k = \sum_{i=o}^{N-1} x_i\alpha^{ik}$$

Notice that this is of the same form as the DFT if $\alpha = e^{-j\frac{2\pi}{N}}$. However, if this computation is done in modulo arithmetic, the resulting transformation is known as Fermat Number Transform (FNT) [23]. Modulo arithmetic is very helpful when large or negative numbers are involved. The FNT is defined as

$$X_k = \sum_{i=o}^{N-1} x_i\ \alpha^{ik}(\text{mod } F_t) \qquad (3.7)$$

where

Modulus F_t is called the t<u>th</u> Fermat number.

The inverse transform is defined as

$$X_n = N^{-1}\sum_{k=o}^{N-1} x_k\ a^{-nk}(\text{mod } F_t) \qquad (3.8)$$

A Fermat number is of the form

$$F_t = 2^{2^t} + 1$$

and α is chosen such that

$$\alpha^N = 1 \pmod{F_t}$$

This hardware implementation depends upon two facts:

1) α, itself is a power of two, i.e., $\alpha = 2^s$

   where

   s is an integer.

This means that in taking the FNT according to Equation (3.7), or the inverse FNT, as in Equation (3.8), the operations needed are shifted right (or left) a number of times, the contents of the input register and then accumulate the result for all input data that are loaded into the input register.

2) The second fact is that the choice of a Fermat number of the form

$$F_t = 2^{2^t} + 1$$

makes the F_t-modulo arithmetic easy to implement.

The operations modulo a Fermat number can be implemented as follows:

a) Addition:

If two 2^t-bit integers are added, a carry bit may be obtained, which is

$$\text{carry} \quad 2^{2^t} = -1 \pmod{F_t}$$

Therefore, if the carry bit is subtracted from the result, the modulo addition is completed. e.g., take

$t = 2 \qquad F_2 = 17$

$10 + 9 = 19 = 2 \pmod{17}$

```
 1010
 1001 +
10011
    1
 0010 = 2
```

b) Subtraction:

To subtract find the F_t complement of the subtrahend, and add. The F_t complement is obtained by complementing every bit and adding 2. e.g., take

$9 - 4 = 5 \pmod{17}$

$\quad = 9 + (-4) = 9 + 13 = 5 \pmod{17}$

```
  4 : 0100
- 4 : 1011 + 10 = 1101
                  1001 +
                 10110
                     1
                  0010 = 5
```

c) <u>Multiplication:</u>

The multiplication gives the result in 2t bits or two adjacent words of 2^t bits. The multiplication is done by simply subtracting the higher word from the lower one. e.g., take

$$13 \times 9 = 117 = 15 \pmod{17}$$

$$117 = \underset{H}{0111}\ \ \underset{L}{0101}$$

$$-H = 1000 + 10 = 1010$$

$$\begin{array}{rr} L & 0101 \\ -H & \underline{1010} \\ & 1111 = 15 \end{array}$$

3.4.1.2 <u>The Cook-Toom Algorithm</u>

The number of multiplications required for the convolution of Equation (3.3) can be reduced further than that needed in the DFT implementation. The Cook-Toom algorithm [24] is a very efficient one and performs the convolution in 2N-1 multiplications.

Define a generating polynomial for X_n as

$$X(z) = \sum_{i=o}^{N-1} x_i z^i \qquad (3.9)$$

This is, in fact, the Z-transform of x_i except that the powers of z are positive. Likewise, define H(Z) for h_n and let

$$Y(Z) = H(Z) \times (Z) \qquad (3.10)$$

where

Y(Z) is a 2N-2 degree polynomial.

Select 2N-1 distinct numbers, b_i, i=0,1, ..., 2N-2, and substitute those for Z in Equation (3.10), to obtain m_i such that

$$m_i = Y(b_i) = A(b_i)X(b_i) \qquad (3.11)$$

There is one polynomial of degree 2N-2,W(Z), that satisfies these equations. Using the Lagrange interpolation [24], W(Z) can be expressed as

$$W(Z) = \sum_{i=o}^{2N-2} m_i \prod_{j \neq i}^{2N-2} \frac{(Z-b_j)}{(b_i-b_j)} \qquad (3.12)$$

The evaluation of W(Z) requires 2n-1 multiplications. Since $A(b_i)$'s and $X(b_i)$'s are linear combinations of the a_i's and X_i's, respectively, Equation (3.11) can be rewritten in a matrix-vector form

$$\underline{m} = (\underline{B}\ \underline{A}) * (\underline{B}\ \underline{X}) \qquad (3.13)$$

where

$\underline{A}$ and $\underline{X}$ are N-element column vectors, whose elements are a_i and X_i, respectively.

$\underline{M}$ is a 2N-2 element column vector with elements $m_i = Y(b_i)$.

The operation * is an element-by-element multiplication. The matrix B has its elements as b'_{ij} where

$$b'_{ij} = b_i^j \quad \begin{array}{l} j=0,1, \ldots, N-1 \\ i=0,1, \ldots, 2N-2 \end{array}$$

Now, Equation (3.12) implies that Y(Z) is a polynomial that has coefficients which are linear combinations of m_i, or if

$$Y(Z) = \sum_{i=o}^{2N-2} y_i \, Z^i$$

then

$$Y_i = \sum_{j=o}^{2N-2} C_{ij} \, m_j \qquad (3.13)$$

where

C_{ij}'s are constants.

This can be written as

$$\underline{Y} = \underline{C} \, \underline{m}$$

where

$\underline{Y}$ and $\underline{M}$ are 2N-1 element column vectors with elements W_i and M_i, respectively, and $\underline{C}$ is 2N-1 by 2N-1 matrix.

The hardware implementation of this algorithm requires a lot of precomputation, thus making the implementation, itself, simple. Each b_i is different, and non-repeating, for example, ... −3,−2,−1,0,1,2, ..., then matrix B can be constructed. Since a_i are constants (filter coefficients), the product $\underline{B}\ \underline{A}$ can be precomputed, too. Let

$$X_i = \sum_{j=o}^{B-1} X_{ij}\ 2^i \tag{3.14}$$

where

$X_{ij} \varepsilon \{0,1\}$ and

B is the word-length of the input data.

And let $\underline{X}.j$ be an N-element column vector whose elements are the $j^{\underline{th}}$ bits of all X_i. Equation (3.12) becomes

$$\begin{aligned}\underline{m} &= (\underline{B}\ \underline{a}) * \underline{B}\ (\sum_{j=o}^{B-1} \underline{X}.j\ 2^j) \\ &= \sum_{j=o}^{B-1} \{(\underline{B}\ \underline{a}) * \underline{B}\ \underline{X}.j\}2^j \\ &= \sum_{j=o}^{B-1} \underline{F}.j\ 2^j\end{aligned}$$

where

$\underline{F}.j$ is an N-element column vector defined by

$$\underline{F}.j = (\underline{B}\ \underline{a}) * (\underline{B}\ \underline{X}.j)$$

Furthermore

$$m_i = \sum_{j=o}^{B-1} F_{ij}\ 2^j \qquad i=0,1,\ ...,\ 2N-1 \tag{3.15}$$

Substituting Equation (3.15) in Equation (3.13)

$$Y_i = \sum_{j=o}^{2N-2} C_{ij} \sum_{k=o}^{B-1} F_{jk} 2^k$$

$$= \sum_{k=o}^{B-1} G_{ik} 2^k \quad (3.16)$$

where

$$G_{ik} = \sum_{j=o}^{2N-2} C_{ij} F_{jk} \quad (3.17)$$

According to Equation (3.16), the output of the filter can be computed by using the function G_{ik}, which is derived from Equation (3.15), using Equation (3.17). Notice that the evaluation of Equation (3.15) requires the $j^{\underline{th}}$ bits of all the input data samples.

3.4.2 FFT Implementations

FFT implementations differ widely due to the fact that the process of evaluating FFT, as well as the approach, can be different. Many of the ideas of digital filter implementation can be adopted as a basis towards the implementation of the FFT. In fact, Peled and Lin [25] suggested hardware FFT implementations, using ROMs, and Zohar [26] suggested another using counters. However, FFT implementations are of higher complexity than those of the digital filters.

There are many methods, but four of these are very basic, and well-suited for processor design. They are as follows:

1) Sequential processor

A memory is used to store resulting Fourier coefficients. Since only one basic operation is involved (i.e., one complex multiplication followed by an addition or a subtraction), and since also the accessing pattern is highly regular, the amount of hardware can be relatively small. The sequential processor is shown in Figure 3.14. This processor has one arithmetic unit in which $(N/2)\log_2 N$ operations are performed sequentially, with execution time of $T\ N/2\ \log_2 N$. T is the time needed to perform one operation.

2) Cascade processor

Parallelism can be introduced into the flow diagram of Figure 3.14. This is done by using separate arithmetic units for each iteration. The throughput is increased by a factor of $\log_2 N$ by doing so. The first arithmetic unit performs the operations labelled 1 through 4, the second performs 5 through 8, and so forth. Fig. 3.15 shows this cascade structure. There are m arithmetic units, m iterations are performed in parallel, $N/2$ operations are

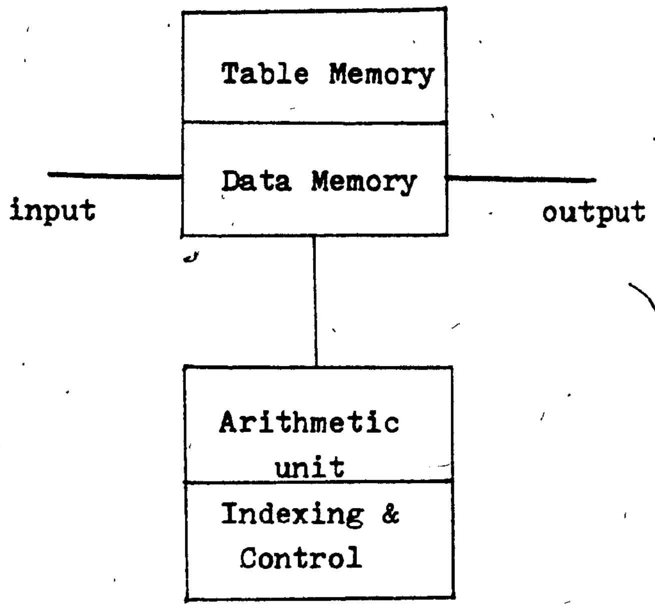


Figure 3.14 A sequential processor.

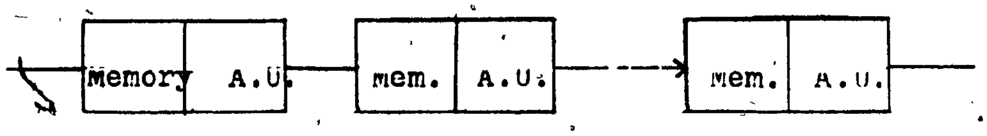


Figure 3.15 A cascade processor . .

performed sequentially. Buffering is incorporated within the processor in the form of time delays. This organization lends itself well to multi-channel operation where sets of samples are interleaved.

3) <u>Parallel Herative Processor</u>

Parallelism can be introduced within each iteration. By using four arithmetic units, the operations labelled 1 through 4 can be performed in parallel before performing operations 5 through 8 in parallel, etc. The processor performs iterations sequentially, but performs all of the operations for each iteration in parallel. Figure 3.16 depicts this organization. N/2 arithmetic units are needed with execution time of T $\log_2 N$ μs. There are m iterations performed sequentially.

4) <u>Array analyzer</u>

By pipelining a number of different sets of data through this processor simultaneously, the effective execution time is simply the time required for performing one basic operation. There are (N/2) $\log_2 N$ arithmetic units and that much operations performed in parallel. The execution time is T μ sec, which is very fast, but very expensive, in terms of hardware. Figure 3.17 shows this structure.

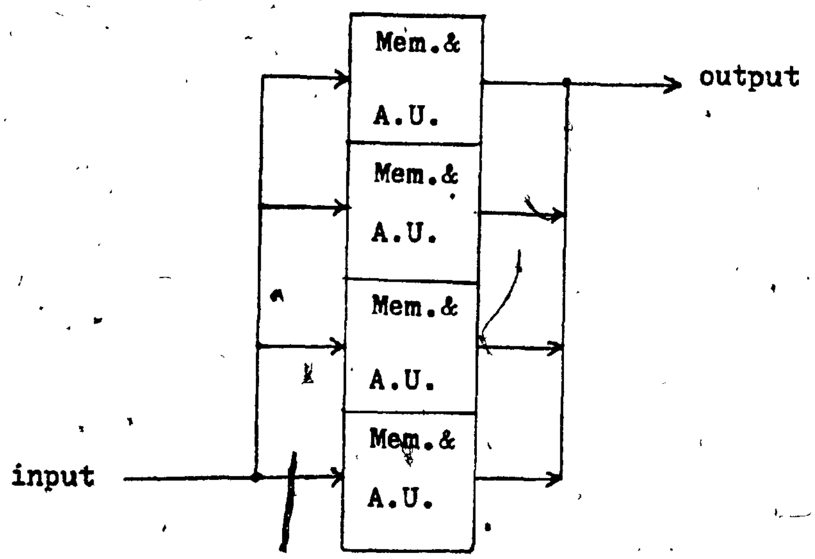


Figure 3.16 Parallel iterative processor.

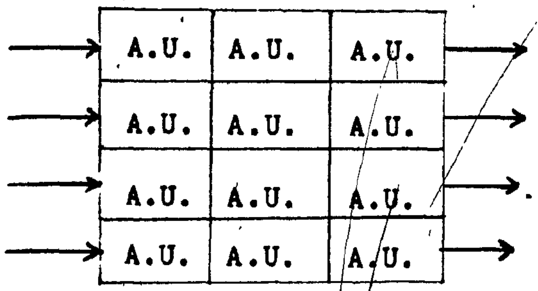


Figure 3.17 Array analyzer.

3.4.3 Hardware Architecture

The design of hardware systems for digital signal processors, can be done in a similar way to that of other designs of other hardware systems, i.e., it is based on a good hardware architecture. The principles involved are various. The most important ones are discussed here.

a) Parallelism

The sub-units used for the design can be constructed and connected in a parallel way. This implies that some units are working in parallel to speed-up the processing.

b) Modularization

If the units are designed in such a way so that they can be assembled together to represent the processor as an architectural outfit, composed of well-designed and integrated module, the implementation becomes more efficient and suitable for MSI, or even LSI.

c) Specialized memories

Instructions leading to the processor performance have to be stored in memories. If special memories are used, the instruction fetch can become much faster. Therefore, higher speed is achieved.

d) Specialized arithmetic units

High-speed multipliers that can be multiplexed easily would be very important building blocks for the arithmetic units of the processor. A ~~function~~al module which computes (x.y) + Z is highly useful since it is ideal for sum-of-products calculations. Such a unit would be perfect for implementing convolution, transformation and inverse-transformation with high-speed floating-point units.

e) Combinatorial and sequential logic

The design of the arithmetic units can depend on either sequential or combinatorial logic. Combinatorial logic is more efficient and includes array multipliers and multiplexors.

CHAPTER IV
CONCLUSIONS

There are several approaches to the hardware implementations of digital filters. Each approach has to trade one aspect of the implementation for another. Speed, for example, has to be traded for high cost. The different methods, discussed earlier, provide very convenient implementations for certain applications.

The conventional implementation, by using a multiplier, employs a small set of relatively simple digital circuits in a highly regular and modular configuration well suited to LSI construction. By using parallel processing and serial two's complement arithmetic, the required arithmetic circuits are greatly simplified. The processing rate is limited only by the speed of the basic digital circuits (adders and multipliers). The resulting filter is readily multiplexed to process multiple data inputs, or to effect multiple, but different filters or both. This multiplexing would use the same arithmetic circuits, thus providing for efficient hardware utilization.

Combinatorial implementation is more efficient than conventional implementations. The multiplier is replaced by memory in this approach. This method offers significant savings in cost and power consumption over conventional implementation. It permits operation at higher speed also. These savings and efficiency are due to the fact that the flexibility inherent in using a multiplier and thus permitting an infinite number of transfer functions to be realized is not utilized in practice when implementing filters for fixed systems. This method eliminates this over-capacity. Therefore, the general purpose multiplier is avoided and instead, high density and high speed memory are needed. Such implementations are generally better, more efficient, and faster than the conventional implementation.

Implementation using counters provides a very convenient method for eliminating round-off errors. It provides also a high speed and relatively low cost. However, the cost is affected by the fact that radix converters are necessary. Computation errors can be minimized in this implementation, especially when the digital filter is designed in second-order blocks. This implementation, however, requires a large number of bits for a specified error bound. Even so, it still provides a definite cost advantage. If higher speed rather than low cost is the main goal, one should consider using this implementation to construct the second-order building blocks and realize the overall filter as a cascade of these.

Implementation using RAMs can be advantageous in some applications. A filter implemented by using a RAM would achieve its speed by the total elimination of any computing to get the accumulator, the partial results to be summed. Therefore, its delay is lower (the time elapsed from the availability of an input sample to the generation of the corresponding output sample.) Hence, the RAM filter has a definite advantage in those applications where a minimal delay is important. Since the storage requirements of the RAM filter increase exponentially with its order, this implementation becomes less appealing for a high-order filter.

The Residue Number System implementation is attractive for non-recursive digital filters, which require only multiplication and addition, because these operations are very fast in an RNS. Both residue addition and multiplication can be implemented by look-up tables stored in high-speed bipolar PROM. The Chinese remainder theorem provides a simple and efficient decoding algorithm which is easily implemented. Since the RNS implementation in its fundamental form produces filter outputs with full precision (no roundoff error), it is particularly attractive for real time filtering of radar and image data. Although emphasis has been placed on the RNS as a high-speed implementation mechanism, the RNS structure is also attractive for certain low-speed operations. Although cost-speed trade-offs are evident, the RNS structure appears to compete favourably with the conventional filters.

Larger digital signal processors can be implemented and used in filtering data. Processors that use the Fast Fourier Transform algorithms to compute the Discrete Fourier Transform of the input sample can do the filtering by minimal multiplications. Other processing would use other transforms, like the Fermat Number Transform and others. They also may be implementations of algorithms implemented to do convolution. Convolution, as a matter of fact, is a non-recursive filtering process. Hardware implementations are demanding in terms of hardware components. However, they are very useful and can be used to implement a high order non-recursive digital filter. The major application of such implementations is digital signals and image processing.

REFERENCES

[1] A. Antoniou, Digital Filters: Analysis and Design, New York: McGraw-Hill, Inc. 1979.

[2] L. Jackson, J. Kaiser and H. McDonald, "An Approach to the Implementation of Digital Filters", IEE Trans. Audio Electroacoustics, AU-16(3), pp.413-421 (1968).

[3] R. Gabel, "A Parallel Arithmetic Hardware Structure for Recursive Digital Filtering", IEEE Trans. Acoustics. Speech, Signal Processing, ASSP-22(4), pp.255-258, (1974).

[4] A. Croisier et al., "Digital Filter for PCM Encoded Signals", U.S. Patent 3-777-130, Dec. 4, (1973).

[5] A. Peled and B. Liu, "A New Hardware Realization of Digital Filters", IEEE Trans. Acoustics, Speech, Signal Processing, ASSP-22(6), pp.456-462, (1974).

[6] A. Peled and B. Liu, "A New Approach to the Realization of Non-Recursive Digital Filters", IEEE Trans. Audio Electroacoustics, AU-21(6), pp.477-484, (1973).

[7] O. Monkewich and W. Steenart, "Compounding For Digital Filters", Proc. 1975 IEEE Int. Symp.Circuits Syst., pp.68-71.

[8] S. Zohar, "New Hardware Realizations of Nonrecursive Digital Filters", IEEE Trans. Comp., C-22(4), pp. 326-338, (1973).

[9] S. Zohar, "The Counting Recursive Digital Filters", IEEE Trans. Comp. C-22(4), pp.338-346, (1973).

[10] S. Zohar, "Negative Radix Conversion", IEEE Trans. Comp. C-19(3), pp.222-226, (1970).

[11] W. Little, "An Algorithm for High-Speed Digital Filters", IEEE Trans. Comp. C-23(5), pp.466-469, (1974).

[12] C. Yuen, "On Little's Digital Filtering Algorithm", IEEE Trans. Comp., C-25(4), p.309, (1976).

[13] N. Szabo and R. Tanaka, Residue Arithmetic and Its Applications to Computer Technology, New York: McGraw-Hill, 1967.

[14] W. Jenkins and B. Leon, "The Use of Residue Number Systems in the Design of Finite Impulse Response Digital Filters", IEEE Trans-Circuits Syst., CAS-24(4), pp.191-200, (1977).

[15] W. Jenkins, "A New Algorithm For Scaling in Residue Number Systems With Applications to Recursive Digital Filtering", Proc. IEEE Int.Symp.Circuits and Systems, New York. pp.56-59, (1977).

[16] W. Jenkins, "Recent Advances in Residue Number Techniques for Recursive Digital Filtering", IEEE Trans. Acoustics, Speech, Signal Processing, ASSP-27(1), pp.19-30, (1979).

[17] M. Soderstrand, "A High-Speed Low-Cost Recursive Digital Filter Using Residue Number Arithmetic", Proc. 1977, IEEE, pp.1065-1067.

[18] D. Banerji, "A Novel Implementation Method for Addition and Subtraction in Residue Number Systems", IEEE Trans. Comp. C-23(1), pp.106-108, (1974).

[19] U. Heute, "Hardware Considerations for Digital FIR Filters Especially With Regard to Linear Phase", AEÜ, Band 29, (1975), Heft 3.

[20] H. Nussbaumer, "Digital Filtering Using Complex Mersenne Transforms", IBM Journal of Research and Development, Vol. 20, No. 5, September 1976, pp. 498-504.

[21] D. Pitassi, "Fast Convolution Using the Walsh Transform", IEEE Trans. Acoustics, Speech, Signal Processing, ASSP-19, (1971).

[22] N. Ahmed, T. Natarajan and K. Rao, "Discrete Cosine Transform", IEEE Trans. Comp., C-23(1), pp.90-92, (1974).

[23] R. Agarwal and C. Burrus, "Fast Convolution Using Fermet Number Transforms With Applications to Digital Filtering," IEEE Trans-Acoustics, Speech, Signal Processing, ASSP-22(2), pp.87-99, (1974).

[24] R. Agarwal and J. Cooley, "New Algorithms for Digital Convolution", IEEE Trans. Acoustics, Speech, Signal Processing, ASSP-25(5), pp.392-410, (1977).

[25] A. Peled and B. Liu, Digital Signal Processing, Wiley, New York, 1976.

[26] S. Zohar, "Fast Hardware Fourier Transformation Through Counting", IEEE Trans.Comp., C-22, pp.433-441, (1973). Addendum, C-23, p.989, September 1974.

ADDITIONAL REFERENCES

A. Elliot, S. Haykin and D. Hawkes, "Hardware Implementation of a Recursive Digital Filter for M.T.I. Radar", Proc.IEE, Vol.122, No.2, pp.137-141, (1975).

E. Lüder, "Integrierbare Filter In VLSI-Technik", AEÜ, Band 32, Heft 10, pp.386-388, (1978).

T.Thong, "A New Sum of Products Implementation for Digital Signal Processing", IEEE Trans. Circuits Syst., CAS-25(1), pp.27-31, (1978).

W. Jenkins, "A Highly Efficient Residue-Combinatorial Architecture for Digital Filters", Proc.IEEE, Vol.66, No. 6, pp.700-702, (1978).

H. Nussbaumer, "Fast Multipliers for Number Theoretic Transforms", IEEE Trans.Comp.C-27(8), pp.764-768, (1978).

S. Mitra and G. Sorkens, "On the Implementation of a Two-Dimensional FIR Filter Using a Single Multiplier," IEEE Trans. Comp. C-27(8), pp.762-764, (1978).

A. Peled and B. Lin, "Some New Realizations of Dedicated Hardware for Digital Signal Processors", IEEE EASCON, Proc. ~~1974~~, pp.464-468.

G. Dehner, "Design of Efficient Recursive Digital Filters", AEÜ Band 33, Heft 2, pp.87-90, (1979).

B. Lin and A. Peled, "A New Hardware Realization of High-Speed Fast Fourier Transformers", IEEE Trans-Acoustics, Speech, Signal Processing, ASSP-23, (1975).

S. Zohar, "A Realization of the RAM Digital Filter", IEEE Trans. Comp., C-25, pp.1048-1052, (1976).

R. Heyder, "A Hardware Realization of Residue Number Digital Filters", M.S.Thesis, University of California, Davis, CA, 1977.

W. Schüssler, Digitale Systeme zur Signalverarbeitung, Springer-Verlag, Berlin, 1973.

K. O'Keefe, "Remarks on Base Extension for Modular Arithmetic", IEEE Trans.Comp., C-22, pp.833-835, (1973).

S. Freny, "Special Purpose Hardware for Digital Filtering", Proc. IEEE, Vol.63, Apr.1975.

D. Banerji and J. Brozozowski, "Sign Detection in Residue Number Systems", IEEE Trans. Comp. C-18, pp.313-320, (1969).

W. Jenkins and B. Leon, "The Use of Residue Coding in the Design of Hardware for Nonrecursive Digital Filters", Conference Record from the Eighth Asilomer Conference on Circuits, Systems and Computers, Dec.1974, pp.458-462.

C. Yuen, "A Hole on Base-2 Arithmetic Logic," IEEE Trans. Comp., C-23, pp.325-329, (1973).

S. Zohar, "A/D Conversion for Radix (-2)", IEEE Trans. Comp. C-22, pp.698-701, (1973).

S. Bass, D. Gibson and B. Leon, "A Laboratory for Digital Filter Instruction", IEEE Trans. Circuits Syst., CAS-23, pp.212-221, (1976).

C. Burrus, "Digital Filter Structures Described by Distributed Arithmetic", IEEE Trans. Circuits Syst., CAS-24(12), pp.674-680, (1977).

E. Anderson, "A Digital Filter Implemented in Parallel Form", 1971 Symp. on Digital Filtering, Imperial College, London, Aug. 1971.