# New Hardware Realizations of Non-Recursive Digital Filters

S. Zohar
Communications Systems Research Section

*Analysis of the bit-level operations involved in the convolution realizing a non-recursive digital filter leads to hardware designs of digital filters based on the operation of counting. Two distinct designs are outlined: The first one is capable of very high speed, but is rather expensive. The second one is quite slow, but has the advantages of low cost and high flexibility. The basic designs considered utilize fixed point representation for the data and filter coefficients. Variants which allow floating point representation of the coefficients are also described.*

## I. Introduction

Digital filters are fast becoming an important element in all DSN data handling systems. They are particularly advantageous in systems which are digital throughout. In many applications they also offer significant reductions in data storage requirements.

Consider a filter which transforms its input time function $\xi(t)$[1] to the filtered time function $\eta(t)$. A non-recursive digital filter simulates such a filter by computing samples of the filtered signal,

$$\eta_k = \eta(kT_s) \tag{1}$$

from samples of the input signal,

$$\xi_k = \xi(kT_s), \tag{2}$$

according to the formula

$$\eta_m = \sum_{k=0}^{K-1} \xi_{m-k}\alpha_k \tag{3}$$

in which the coefficients $\alpha_k$ are obtainable from the filter transfer function.

There is a growing need for special-purpose machines that would implement such a filter. Particularly attractive here are machines that would produce filtered output

[1]All quantities dealt with in this paper are real.

samples at the rate input samples are generated in the data acquisition environment.

An obvious strategy in realizing such a machine is to compute $\eta_m$ via the following rephrasing of Eq. (3):

$$\left.\begin{aligned} p_{mk} &= \xi_{m-k}\alpha_k \\ \eta_m &= \sum_{k=0}^{K-1} p_{mk} \end{aligned}\right\} \tag{4}$$

that is, the desired output, $\eta_m$, is evaluated through the intermediate entities $p_{mk}$ which require the use of a multiplier.

We propose here a different strategy in which the intermediate entities are obtained by counting. Our main purpose in this paper is to survey the various possible designs implementing this strategy. Interesting and promising designs are indicated in the high-speed high-cost category as well as the low-speed low-cost class. The paper is organized along the following lines: A master design is developed in *Sections II* and *III*. This design serves as the starting point for developing two different basic designs in *Sections IV* and *V*. *Section IV* presents a high-speed design involving high cost. *Section V* is devoted to an inexpensive low-speed design. Several variants of these two designs are also considered.

## II. The Master Design

Let us assume (temporarily) that the data and coefficients satisfy

$$\xi_k, \alpha_k \geq 0 \tag{5}$$

This assumption simplifies the presentation of the main features of the master design and will be removed in *Section III*.

In representing $\xi_k$, $\alpha_k$ in the machine, we assign $J_x$ bits to each input data word and $J_a$ bits to each coefficient. Denoting by primes the truncated versions of $\xi_k$, $\alpha_k$ consistent with the finite $J_x$, $J_a$, we represent the data and coefficients by the integers $x_k$, $a_k$ defined as follows:

$$\left.\begin{aligned} \xi'_k &= 2^{-X}x_k \\ \alpha'_k &= 2^{-A}a_k \end{aligned}\right\} \tag{6}$$

Suitable values of $A$, $X$ can be determined from the known ranges of $\alpha_k$ $\xi_k$.

The immediate effect of the above representation is that the machine will substitute the approximation $\eta'_m$ for the true $\eta_m$ where

$$\eta'_m = \sum_{k=0}^{K-1} \xi'_{m-k}\alpha'_k \tag{7}$$

$\eta'_m$ may be regarded as a sample of the approximated output $\eta'(t)$.

Note that the adopted representation is equivalent to a fixed-point binary representation. Adaptations employing floating point representation for the coefficients are possible with a reasonable amount of extra hardware. Such adaptations are described in *Sections IV* and *V*.

Substituting Eq. (6) in Eq. (7) we get

$$y_m = \sum_{k=0}^{K-1} x_{m-k}a_k \tag{8}$$

where

$$y_m = 2^{(A+X)}\eta'_m \tag{9}$$

is an integer.

Let us examine now the implication of Eq. (8) in terms of bit-level operations. We start by writing down explicitly the binary representations of $a_k$, $x_k$

$$\left.\begin{aligned} a_k &= \sum_{j=0}^{J_a-1} u'_{kj}2^j & u'_{kj} &= 0,1 \\ x_k &= \sum_{j=0}^{J_x-1} v'_{kj}2^j & v'_{kj} &= 0,1 \end{aligned}\right\} \tag{10}$$

Using the representation (10) in Eq. (8), we get an explicit expression for $y_m$ as a triple sum

$$y_m = \sum_{k=0}^{K-1}\sum_{r=0}^{J-1}\sum_{i=0}^{J_a-1} u'_{ki}v'_{m-k,r-i}2^r \tag{11}$$

where

$$J = J_a + J_x - 1 \tag{12}$$

Application of the "multiply-then-add" strategy of Eq. (4) is equivalent to summing over $i,r$ first (thus getting the $p_{mk}$'s) and only then performing the $k$ summation. In the scheme proposed here, we change the order of summa-

tion, making the $r$ summation the last one. Thus, denoting

$$\sum_{k=0}^{K-1}\sum_{i=0}^{J_a-1} u'_{ki}v'_{m-k,r-i} = h'_r \tag{13}$$

we have

$$y_m = \sum_{r=0}^{J-1} h'_r 2^r \tag{14}$$

Strictly speaking, the entity defined in Eq. (13) should be denoted $h'_{mr}$. However, since in most of the following discussion $m$ is constrained to a specific constant value, the simpler notation, $h'_r$, will do.

Equations (13) and (14) embody the basic strategy of the proposed machine. Let us examine Eq. (13) first. Each element of the double sum is a product of two 1-bit entities. Such a product is either 0 or 1 and, practically speaking, no multiplication is involved in its evaluation. A dual input AND gate is all that is needed. Equation (13) is a summation of $KJ_a$ such terms. Thus, if we design a system in which $KJ_a$ dual input gates are fed by the pairs of bits specified in Eq. (13), a count of the number of TRUE gates, will equal $h'_r$.

Equation (14) is implemented by regarding its right-hand side as a polynomial with coefficients $h'_r$, evaluated for the argument 2. Applying the well-known polynomial algorithm we get

$$y_m = (\cdots(h'_{J-1} \cdot 2 + h'_{J-2})2 + \cdots + h'_1)2 + h'_0 \tag{15}$$

The realization of Eq. (15) can now be carried out in a sequence of shift and add operations in a standard accumulator.

The above discussion leads to the general outline of the machine shown in Fig. 1. We see here four basic elements:

(1) A fixed register ($A$) holding the coefficients $a_k$ prescribing the filter transfer function. Its contents would normally stay fixed throughout a specific filter simulation. They may, however, be varied in the course of the filtering process to realize an adaptive filter.

(2) A shift register ($X$) holding the input words $x_i$. These are fed on the right, a bit at a time. Registers $A$,$X$ are cross-linked by AND gates.

(3) A counter that counts the number of TRUE gates after every shift of register $X$. Each such count will be shown to equal a specific $h'_r$.

(4) An accumulator that combines $J$ counter outputs through a shift and add sequence to generate the output word $y_m$.

### A. A Detailed Example

We turn our attention now to the data configuration in registers $A$ and $X$. This is most easily illustrated in terms of a specific example. We adopt the following parameters for our example:

$$K = 3; \quad J_a = 5; \quad J_x = 3 \tag{16}$$

These values (particularly $K$) are too low to be realistic. Their adoption, however, facilitates the presentation of detailed bit configurations and thus makes it easier to grasp the details of the proposed machine. This example is adopted in most of the figures in this paper.

The situation at some intermediate point in the computation of $y_m$ is shown in Fig. 2b where the following convention is followed: A numeral in a register cell represents the radix power assigned to a 1 bit in that location. This power is referred to as the weight of the bit. We shall find it convenient to designate the bit itself by its weight. Thus, we may say that the first cell on the right of register $A$ contains bit 4 of $a_0$. Some of the bits are zero independently of the data. These are indicated by the letter $z$.

We note first that successive $x_i$'s are separated by $J_a - 1$ (=4) zero bits. Thus, it takes $J$(=7) bit times (shifts) to feed one input word.

Similarly, the $A$ register indicates a separation of $J_x - 1$(=2) zero bits between successive words. Here, however, there is no need to have a hardware implementation of these zero bits so that the $A$ register could be split into $K$ independent registers.[2]

Next, we observe that the $x_i$'s have their most significant bit on the left, while the $a_k$'s have their most significant bit on the right. The spacing and the reversed

[2]One would still implement the $A$ register as a single continuous register when it turns out to be convenient to perform the initial loading of $A$ serially. In this case, $A$ would be a shift register. Another argument for a single $A$ register is the flexibility it allows in modifying word size. (Note that for 1-bit $x_k$'s, there is no spacing in the $A$ register).

bit arrangement guarantee that all pairs of bits feeding a common gate satisfy the index relation required by Eq. (13). The double summation appearing there translates simply into a count of all TRUE gates in Fig. 2b. In the specific situation shown here, the count would yield $h'_4$. Similarly, shifting the $X$ register one bit to the left will now yield $h'_3$, and so on.

Having seen how an $h'_r$ is generated in the counter output, we can consider now the overall scheme culminating in the output $y_m$.

The processing starts with the initial situation shown in Fig. 2a where a count yields $h'_6$ of $y_m$ ($h'_{J-1}$ in the general case). We load this count into the accumulator and proceed to compute $h'_5$. This is done by shifting the $X$ register one bit to the left and counting the TRUE gates. We return now to the accumulator, shift its contents left one bit (equivalent to multiplying by 2) and add $h'_5$. The succession of these shift and add operations is the step by step implementation of Eq. (15). We proceed with this sequence up to the last cycle depicted in Fig. 2c. A count here yields $h'_0$. We return now for the last time to the accumulator, shift its contents one bit to the left and add $h'_0$. It now contains the final result, $y_m$.

Examination of Fig. 2c shows that when the above process is continued, the next 1-bit left shift will bring into the $X$ register the most significant bit of $x_{m+1}$. The resulting configuration is almost identical with the one initiating the computation of $y_m$ (Fig. 2a). The only difference is the replacement of $m$ by $m+1$. Hence, the sequence of operations described here automatically leads to the computation of $y_{m+1}$ following the computation of $y_m$, and so on.

The basic design outlined here is valid only when both data and coefficients are non-negative. In the next section we proceed to remove this constraint.

## III. Modified Design to Allow Negative Numbers

General-purpose computers usually handle negative numbers either by the "sign-magnitude" representation or via the use of "2's-complement." Both methods are quite inappropriate in the present application, as they would drastically increase the complexity of the machine.

A possible solution[3] is to apply bias to both data and coefficients so that the numbers presented to the machine are always positive. This requires corrective measures to compensate for the bias. The logic implementation of these measures has been worked out[4] but will not be presented here. Instead, a simpler more attractive solution based on a negative radix number representation (Refs. 1, 2) will be described in detail. Specifically, we consider representing data and coefficients in base $(-2)$. As a simple illustration, consider the number 3:

$$3 = 1\cdot 2^1 + 1\cdot 2^0 = 11_2$$
$$= 1\cdot(-2)^2 + 1\cdot(-2)^1 + 1\cdot(-2)^0 = 111_{-2}$$

Similarly,

$$-3 = 1\cdot(-2)^3 + 1\cdot(-2)^2 + 0\cdot(-2)^1 + 1\cdot(-2)^0 = 1101_{-2}$$

We see that both positive and negative numbers are accommodated with one and the same representation.

Let us examine now the effect of such a representation on the design of our machine. We start with Eq. (10) which will now be replaced by Eq. (17)[5].

$$\left.\begin{aligned} a_k &= \sum_{j=0}^{J_a-1} u_{kj}(-2)^j \qquad (u_{kj}=0,1) \\ x_k &= \sum_{j=0}^{J_x-1} v_{kj}(-2)^j \qquad (v_{kj}=0,1) \end{aligned}\right\} \tag{17}$$

This in turn leads to the modification of Eqs. (13), (14), and (15) as follows:

$$\sum_{k=0}^{K-1}\sum_{i=0}^{J_a-1} u_{ki}v_{m-k,r-i} = h_r \tag{18}$$

$$y_m = \sum_{r=0}^{J-1} h_r(-2)^r \tag{19}$$

$$y_m = (\cdots(h_{J-1}\cdot(-2) + h_{J-2})(-2) + \cdots + h_1)(-2) + h_0 \tag{20}$$

We conclude that if we remove the constraint (5) and adopt a negative binary representation for $\xi_k, a_k$, the machine design of *Section II* is still applicable, provided we modify the sequence of operations in the accumulator, combining the counts so as to realize Eq. (20) rather than Eq. (15).

[3]Suggested by R. F. Emerson of the Communications Systems Research Section.

[4]Author's unpublished manuscript.

[5]$u_{kj}$ is the $j$th bit of $a_k$ in radix $(-2)$. This is different from $u'_{kj}$ which is the $j$th bit of $a_k$ in radix $(+2)$.

The count $h_r$ is a function of the bits in the negative binary representation of the data and coefficients; $h_r$ itself, however, may be represented in any system. For obvious reasons we adopt a standard (base 2) counter to determine $h_r$. Therefore, Eq. (20) should be implemented in a standard (base 2) accumulator thus yielding the result $y_m$ in binary.

The modification to realize Eq. (20) is particularly simple if the accumulator is of the magnitude-sign-bit type. In this case, each shift operation will simply be accompanied by a simultaneous sign reversal. This is all it takes to ensure that the negative radix inputs, $x_k$, are processed to yield positive radix outputs, $y_m$.

We consider now the implications of the adopted negative radix representation. The $a_k$'s are usually computed on a general-purpose binary computer. Their negative binary representation can be easily obtained by incorporating in the coefficient program a subroutine converting from positive binary to negative binary representation. A suitable algorithm for this subroutine is described in Ref. 3.

The situation with respect to the data is somewhat different. The basic configuration of Fig. 1 shows the data already in digital form. In most practical situations, the data are available in analog form so that the very first element in the input of a practical machine would have to be an A/D converter. The above discussion then indicates that we ought to interpose a radix converter (Ref. 3) between the A/D converter and the $X$ register. This is a reasonable approach when a standard commercially available A/D converter is to be used.

A more logical approach, however, would combine the negative radix converter with the A/D converter in a single functional unit that would directly convert the analog input to its negative binary representation. The design of such a converter operating in serial fashion is discussed in detail in Ref. 4. Such a device is particularly attractive in that one does not have to wait for the complete conversion of an analog sample. Thus, the first bits of a data word may be in register $X$ and partly processed before its last bits have even been determined.

An overall outline incorporating the modifications introduced in this section is shown in Fig. 3. We have added here a D/A converter as the last link in the processing. The transformation (9) is effected here. Note in this context that the digital word to be converted is available in the accumulator with more bits than the input $x_k$'s. Usually, a few of the lower bits will simply be discarded at this point.

Figure 3 serves as the starting point for the more specific designs considered in the following two sections. In preparation for this, we introduce here some necessary terminology.

Let $f_b$ be the rate at which input bits are fed to the machine (bit rate). The corresponding period (bit time) $T_b$ is then

$$T_b = \frac{1}{f_b} \tag{21}$$

From the previous discussion it is obvious that $T_b$ is also the period of the clock pulses controlling the $X$ register shifts. Thus, the entities $h_r$ are produced at the rate $f_b$ and the accumulator shown in Fig. 3 is required to complete a shift-and-add cycle in the interval $T_b$.

The sampling rate, $f_s$, of the analog input signal $\xi$, is related to $f_b$ by

$$f_s = \frac{f_b}{J} \tag{22}$$

or equivalently,

$$T_s = \frac{1}{f_s} = JT_b = (J_x + J_a - 1)T_b \tag{23}$$

Of this, $J_xT_b$ is the time required to feed one data word into the $X$ register, while $(J_a - 1)T_b$ is the time required to feed the associated $(J_a - 1)$ zeros.

Another important parameter is the delay time $T_d$. Thus, while $\eta'_m$ is the approximation to the output of a zero delay filter at time $mT_s$, a practical filter will produce this output at time $(mT_s + T_d)$ where $T_d > 0$.

## IV. The Partitioned-Counting Filter

We consider here an elaboration of the master design that achieves high-speed by performing the counting indicated in Fig. 3 in a special way.

Let $N(=KJ_a)$ be the number of gate outputs to be counted and let $N$ satisfy

$$2^{M-1} \leq N < 2^M \tag{24}$$

The total count, $h$, is therefore bounded by

$$0 \leq h < 2^M \tag{25}$$

and is representable by an $M$-bit binary number.

If we use a standard $M$-bit binary counter with clock time $T_c$, the counting time will be $NT_c$. Hence

$$T_b = (N + 1)T_c \tag{26}$$

and the result is a low-speed machine. Such a design has its merits and is pursued further in *Section VI*.

Our concern here is with a high-speed design. Realizing that the low speed associated with the above approach is a direct result of the serial nature of the counting, we propose to subdivide the $N$ gates into groups of $q$ gates each and count all these groups in parallel. We refer to this method as partitioned counting. In the following, we examine this approach in detail.

First, we note that in the interest of efficiency, $q$ should satisfy

$$q = 2^r - 1 \qquad (r = 2,3,\cdots) \tag{27}$$

so that the count of each group is representable as an $r$-bit binary number. The method considered here would replace the single $M$-bit counter with a large number of smaller elementary $r$-bit counters.

Using the bit terminology of *Section II*, we may say that the initial $N$ gates are a representation of the number $h$ as a sum of $N$ zero-weight bits. Similarly, the binary number representing the final count is a representation of $h$ as a sum of $M(\ll N)$ bits whose weights range from zero to $M - 1$. In partitioned counting, the transition from one representation to the other is effected in a large number of small steps. In each such step, the weights are increased while the total number of bits is decreased.

The large number of elementary counters required, means that the investment in hardware is quite high. However, the resulting speed increase is significant. Thus, assuming the same clock time $T_c$, as before, counting of $q$ gates is accomplished in time $qT_c$. This, however, does not yield the final count yet. The outputs of these counters will now have to be operated on in a similar fashion (described later in detail). Hence they should remain undisturbed for another $qT_c$ interval. This means that

$$T_b = 2qT_c \tag{28}$$

It follows that the speed increase over the single counter method is given by the factor $(N + 1)/2q$. For large $N$ and small $q(q_{\min} = 3)$, this factor may be quite large.

To obtain the speed improvement while minimizing the amount of hardware, we try to make sure that all $r$-bit counters are fed $q = 2^r - 1$ inputs. This is most easily facilitated when

$$N = q^\nu \qquad (\nu \text{ integer}) \tag{29}$$

If $N$ does not satisfy Eq. (29), lower efficiency will usually result, though there are various ways to minimize this effect. To simplify the presentation here, we assume from now on that Eq. (29) is satisfied.

We turn now to the details of the scheme. Consider an $r$-bit counter, counting $q$ bits of weight $k$. The result is a sum of $r$ bits ranging in weight from $k$ to $k + r - 1$. The number of bits has thus been reduced, having been multiplied by

$$\beta = \frac{r}{q} = \frac{r}{2^r - 1} < 1 \qquad (r \geq 2) \tag{30}$$

Applying $q^{\nu-1}$ such counters to the $N(=q^\nu)$ zero-weight bits yields a sum of $\beta N = q^{\nu-1}r$ bits with weights ranging from 0 to $r - 1$. Now we consider all bits of the same weight as inputs for further counting. $q^{\nu-2}r$ counters will be required at this level and the result will be a sum of $q^{\nu-2}r^2$ bits with weights ranging from 0 to $2(r - 1)$.

The pattern emerging from the above description is best illustrated in terms of an array such as that of Fig. 4 which applies for $r = 3$. This is essentially an ordered listing of the number of bits as a function of weight $(k)$ and counting level $(n)$. The numbers for each level have a common multiplier indicated on the left. Thus, the initial situation referred to as level 0 has $q^\nu \cdot 1$ bits of weight zero. In level 1 (following first count), we have $q^{\nu-1}$ bits of each of the weights 0,1,2 and so on. Factoring out the common level multiplier is particularly convenient since the resulting numbers in the array are quite easy to compute in this case.

Let us assign the symbol $B_{nkr}$ to the array element corresponding to level $n$, weight $k$ in a partitioned count-

ing scheme using $r$-bit counters. (Example: $B_{2,1,3} = 2$). The earlier description of the bit transformation effected by a single $r$-bit counter, implies that the elements of level $n$ are derived from those of level $n-1$ as follows: Element $B_{n-1,kr}$ contributes a term $B_{n-1,kr}$ to each of the following: $B_{nkr}, B_{n,k+1,r}, \cdots, B_{n,k+r-1,r}$. Equivalently,

$$B_{nkr} = \sum_{i=0}^{r-1} B_{n-1,k-i,r} \tag{31}$$

The other obvious properties are

$$B_{okr} \rightleftharpoons \delta_{ko} \ (\delta_{ij} \text{ is Kronecker's } \delta) \tag{32}$$

$$B_{nkr} = 0 \qquad \text{for } k < 0 \tag{33}$$

The last equation is a direct result of the fact that we are dealing here with integers so that no bits are assigned negative weights. Equations (31) to (33) form a prescription for the computation of any $B_{nkr}$.

We note in passing that the array of Fig. 4 is actually a generalized Pascal triangle. The special case $r = 2$ yields the standard Pascal triangle. Similarly, $B_{nkr}$ is a generalized binomial coefficient. Indeed, the above eqns. imply that

$$B_{nk2} \equiv \binom{n}{k}$$

The $B_{nkr}$ parameters prescribed by Eqs. (31) to (33) are involved only in the initial levels of counting. The terminating levels display a different pattern. We illustrate this with a particular example involving $N = 2401 = 7^4$ (that is, $r = 3$, $\nu = 4$). The corresponding array is shown in Fig. 5. Obviously, the initial levels of Fig. 5 are identical with those of Fig. 4. Up to level 4, the elements are generated in a unique fashion following Eqs. (31) to (33). At this level, the multiplier is 1 so that the elements in row 4 ($n = 4$) directly specify the number of bits for each weight. Note that while the single zero-weight bit has reached its final value, there are still 19 bits of weight 4. Further counting is therefore required, but the organization will now be different. Obviously, nothing more is to be done with the zero-weight single bit. The 4 bits of weight 1 will require one counter thus entailing a somewhat reduced efficiency. On the other hand, one may postpone action relating to the 4 bits of weight 7, arguing that counting of lower weight bits will increase the number of weight 7 bits at the upper levels. Thus, by postponing their count, we increase the overall efficiency. The same argument applies to the 10 bits of weight 2. Instead of assigning 2 counters to them, we count only 7 of them in one counter, postponing the counting of the remaining 3. Following these reasonable (but not unique) guidelines, we arrive at the numbers shown in levels 5–14.

The numbers on the right margin of the array in Fig. 5 represent the number of elementary counters used between the levels bracketing their position. Thus, $3^3$ elementary counters are required to transform the bits of level 3 to the configuration of level 4. The total number of counters is 602 as indicated. The major part of this is contributed by the counters operating below level $\nu$. It can be shown that the sum of these is in general $(q^\nu - r^\nu)/(q - r)$. For Fig. 5, this value is 580. In comparison, we see from the indicated total that only 22 counters are involved in the terminating levels.

Since the last level has $n = 14$ and transferring from one level to the next takes the time $qT_c$, the final count of the $N$ gates will be ready for the accumulator after a delay $14qT_c = 98T_c$. Successive counts, however, are separated by the shorter interval $T_b$. From Eq. (28) we get $T_b = 2qT_c = 14T_c$.

An even faster design is illustrated in Fig. 6 which applies 2-bit elementary counters to the slightly smaller $N = 2187 = 3^7$ (that is, $r = 2$, $\nu = 7$). Each counter uses fewer components but more counters are needed. The counting delay is given by $20qT_c = 60T_c$, while $T_b = 6T_c$. These figures should be compared to $2187T_c$ $(= NT_c)$ and $2188T_c$ $(=(N+1)T_c)$ which are the corresponding values for the single, 12-bit counter. The reduction from $2187T_c$ to $60T_c$ is due to the parallel counting of bits comprising a specific count. The further reduction of the spacing to $6T_c$ depends on the fact that at each particular instant during the $60T_c$ required to produce a single count, only a small portion of the elementary counters is needed for that specific count. This permits the initiation of a new count long before the earlier count has been completed. In Fig. 6, 10 distinct $h_r$'s are being simultaneously counted in a staggered arrangement. This is further illustrated in Fig. 7. Note that during the interval $qT_c$ starting at $t_1$, the counting of $h_3$ is initiated, that is, its bit configuration is transformed from level 0 to level 1. At the same time, the counting of the earlier input, $h_4$, (the evaluation sequence prescribed in *Section II* is $h_r$, $h_{r-1}$, $h_{r-2}, \cdots$) is further processed by transforming from level 2 to level 3, the still earlier input $h_5$ is transformed from level 4 to level 5 and so on.

The processing of $h_3$ is continued in the next interval $(t_2,t_3)$. Note that the new input, $h_2$, is not entered yet in

order to not disturb the bits in level 1 which are being counted during this interval. The input spacing of $2qT_c$ and its implications are evident in Fig. 7.

The simultaneous counting of several $h_r$'s means that storage has to be provided for those bits of $h_r$ which attain their final value before termination of the count. Consider Fig. 6: The final value of bit 0 of $h_r$ is established at level 7. No further counting involving this bit is required. At this point it resides in a flip-flop of one of the elementary counters. This counter will preserve its information during the next $qT_c$ interval but following that, it will be zeroed and start counting a term of $h_{r-1}$. Evidently, the quiescent interval has to be used to transfer the $h_r$ bit elsewhere. The simplest solution is to transfer the zero bit to a shift register and step this register at intervals $2qT_c$. With the proper number of cells (7 in this case), this register will output the zero bit of $h_r$ at the time all its other bits are ready. Needless to say, this shift register will serve in the same capacity for the zero bits of $h_{r-1}$, $h_{r-2}$ and so on. Inspecting Fig. 6 and repeating the above arguments we find that accommodation of the finalized bit 1 requires a 6-bit shift register, bit 2 needs a 5-bit shift register and so on, down to bit 9 calling for a single flip-flop (1-bit shift register).[6]

### A. Possible Variations

We turn our attention now to three important variations.

The first variant counts $q$ gates in time $T_c$ (rather than $qT_c$). The idea is to replace the elementary counters, serially counting $q$ inputs, with elementary adders which add the $q$ inputs on parallel.

For $q = 3$, the well-known "full adder"[7] (Ref. 5) is just such a device. It has three inputs (a carry and the two bits to be added) and produces two output bits (The sum bit and the new carry bit). These may be regarded as the bits of the output count (the sum is bit zero, the new carry is bit one).

Alternatively, one might consider the $q$ input bits as specifying an address in a ROM organized as $2^q$ words of $r$ bits each. "Counting" in this case is effected by reading the $r$-bit word stored in the corresponding address. This should obviously be restricted to low $r$. Actually it can be shown that the combination $r = 2$, $q = 3$ requires the smallest overall number of memory bits. However, for higher $r$ (which might be dictated by availability of off-the-shelf ROMs), Eq. (27) does not prescribe the optimum $q$.

A second variation is based on the use of buffers to increase speed. If the output of each elementary $r$-bit counter is transferred to an $r$-bit buffer rather than feeding directly the next level counters, the bit period should consist of $qT_c$ to count and $T_c$ to transfer the counts (in parallel) to the buffers. Thus, Eq. (28) should now be replaced by

$$T_b = (q + 1)\,T_c \tag{34}$$

Speed is increased by the factor of $2q/(q + 1)$ at the expense of essentially doubling the number of flip-flops. Therefore, this method should always be compared to the alternative of reducing $q$. Consider for example the case of Fig. 5 ($q = 7$). With 1806 ($= 602 \cdot r$) flip-flops assigned to the elementary counters, we got $T_b = 14T_c$. Adding buffers, we have 3612 flip-flops yielding $T_b = 8T_c$. This should be compared to Fig. 6 ($q = 3$) which assigns 4364 ($= 2182 \cdot r$) flip-flops to get $T_b = 6T_c$ for a slightly lower $N$.

Note the disadvantage of a counting delay of $14 \cdot (q + 1)T_c = 112T_c$ for the buffered, $q = 7$, case as compared to $60T_c$ for the unbuffered, $q = 3$, case.

The third variant considered relates to the use of floating point representation for the filter coefficients $\alpha_k$. Let the ratio of the largest $|\alpha_k|$ to the smallest $|\alpha_k|$ be $\rho$. If $\rho$ is a large number, then even if we allot only a very small number of bits to the smallest $|\alpha_k|$ we still end up with a large $J_a$. This hurts us in two ways:

(a) The sampling rate is reduced (see Eq. 23).

(b) $N = KJ_a$, the number of gates to be counted is increased, sharply increasing the cost.

We consider here a simple strategy which accepts (a) but provides an answer for (b). To illustrate the strategy, consider the following example: We wish to assign 5 bits to each coefficient but if the lowest $|a_k|$ satisfies that, we find that we need $J_a = 20$ to represent the highest $|a_k|$. We organize registers $A$, $X$ on the basis of $J_a = 20$ but attach gates and counters only to the 5 most significant bits of each $a_k$. This is equivalent to the adoption of a floating point representation for the coefficients.

[6]The handling of bits 3, 4, 6, 8 is slightly different, calling for a 1-bit buffer interposed between the finalized counter bit and the shift register.

[7]Plus two latches to store the count.

It should be noted that we are considering here a general-purpose machine that should be able to accommodate a wide range of requirements. In view of this, the best way to implement the above strategy is to use a single, sufficiently long, register for the coefficients but effect the connection of its cells to the gates with relays or switches under the control of the machine operator. In addition to loading the coefficients $a_k$ into the machine, the operator will also set the switches.

It has already been mentioned that the coefficients $a_k$ will usually be computed on a general-purpose computer. Assume then that they are punched on a paper tape which is fed into the digital filter. In situations where the additional expense is justified, this paper tape could be made to command the switches and set them directly without the intervention of the operator.

## V. The Circulating-Lines Filter, A Low-Speed Low-Cost Approach

In the introduction to *Section IV* we considered briefly a design based on a serial counter. We concluded there that such a machine would be very slow. In the present section we proceed to show that this disadvantage is coupled with the advantage of low cost, which makes such a design quite attractive in some applications.

Consider Fig. 3 and assume that counting is to be done serially. Obviously, there is no need in this case to produce all the gates' outputs simultaneously as shown there. Rather, one could take advantage of the serial nature of the counting and produce the gate outputs serially. This suggests using circulating delay lines or MOS shift registers to replace the $X$ and $A$ registers of Fig. 3[8].

We shall see that the hardware simplification accruing to serial counting extends also to the accumulator appearing in Fig. 3. Thus, it will be shown that a somewhat more complex serial counter eliminates the need for the accumulator altogether. We consider now the various aspects of a serial counting design in some detail.

The basic design of Fig. 3 shows the data words being fed into the $X$ register with the most significant bits leading. This means that the $h_r$'s are computed in the time sequence $h_{J-1}, h_{J-2}, \cdots, h_0$. In the present case, we find it more convenient to feed the least significant bits first, generating the $h_r$'s in the time sequence $h_0, h_1, \cdots, h_{J-1}$. Since serial A/D converters produce the most significant bit first, this sequencing means that we have to wait for complete conversion of a word prior to using any of its bits. However, as the evolving design is characterized by long delay, the additional delay due to this effect is negligible.

The A/D converter could be either of the type producing the negative binary representation directly (Ref. 4) or a commercial low-speed A/D converter followed by a radix converter. Taking advantage of the adopted bit sequence, a very simple radix converter is feasible (Ref. 3, Sect. IV).

Use of circulating lines for the $A$, $X$ registers allows one gate to replace the $KJ_a$ gates shown in Fig. 3. It takes one complete revolution of the coefficients line, $A$, to sweep all coefficients past the single gate and generate $h_r$. For the next computation ($h_{r+1}$), the relative alignment of the $X$ and $A$ registers must be changed by one bit position. Also, register $X$ must discard one old bit and admit one new bit. All these objectives are easily attained by applying a widely used "trick," namely, adding an extra cell to the $X$ register and using the flip-flop implementing it, to inject new data bits.

The design is illustrated in Fig. 8a. The extra bit is stored in cell B whose input is controlled by switch S. The indicated state of S refers to its state during the immediately following 1-bit shift. The example shown is the simple one adopted in Eq. (16). Fig. 8a shows the situation following the evaluation of the first term of $h_0$ of $y_m$ (see Eqs. 18 and 19). As the two registers are swept in step past the single gate, the remaining terms of $h_0$ are evaluated. During all these operations, switch S remains as shown in Fig. 8a so that any bit shifted out of the gate feeding cell, lands in cell B.

Fig. 8b shows the situation following the evaluation of the last term of $h_0$ of $y_m$. Note that switch S now connects to the right. This means that in the next shift, the leftmost bit will be lost, and cell B will be loaded with the new data input bit (bit 2 of word $x_m$). This new state of affairs is shown in Fig. 8c. Note that switch S has now returned to its left position and we have here the evaluation of the first term of $h_1$ of $y_m$.

The time difference between Figs. 8a and 8c corresponds to one full revolution of the coefficients line. Hence, these figures represent the state of affairs that would emerge in "strobing" the lines to "freeze" the motion of the coefficients line. The resulting pattern in

[8]Suggested by H. C. Wilck of the Communications Systems Research Section.

the $X$ register (excluding cell B) is a 1-bit shift to the right with a new bit being fed on the left and the oldest bit being ejected on the right. This is almost identical with the basic pattern considered in *Section II*. The only difference is in the shift direction since the order of computing the $h_r$'s is now the reverse of that of *Section II*.

We are now in position to consider the overall design of the machine as shown in Fig. 9. The right part of this figure is essentially a redrawing of Fig. 8a. Switch S of that figure is replaced here with an implementation using three gates controlled by clock sequences $C_1$, $C_2$. Sequence $C_2$ steps the circulating lines, while clock $C_1$ controls the serial radix converter and hence the rate of feeding new bits.

As we have already seen, counting the TRUE outputs of gate G over the proper time interval, will yield $h_r$. This counting as well as the assembly of counts to get the output word $y_m$, is implemented in a modified up-down counter. Unlike the standard up-down counter which accepts inputs into bit 0 only, here we allow inputs to be delivered to the other bits too. This is permissible as long as only one input at a time is activated and the rate of input is sufficiently low so that rippling through of carries resulting from the last bit fed has terminated before a new bit is applied.

The effect of the above modification is that rather than counting (up or down) by ones only, this counter can count by any power of 2. This immediately suggests its application in the construction of the output word $y_m$ according to Eq. (19) which may be rephrased as:

$$y_m = \sum_{r=0}^{J-1} (-1)^r \left\{ \sum_{k=1}^{h_r} 2^r \right\}$$

All we have to do is steer the sequence of bits coming out of gate G to the proper input terminal and properly set the (up/down) mode of counting. Specifically, bits comprising $h_0$ should be steered to input 0, counting up, bits comprising $h_1$ should be steered to input 1, counting down and so on.

As long as the counter does not overflow, this automatically yields the correct $y_m$ with negative numbers appearing in their 2's complement representation.[9] This means that the counter output can be fed directly to a D/A converter to produce the analog output.

Steering the output of gate G to the proper counter input is controlled by circulating line D. It has $J(=7)$ cells. One of these is set to one. The rest are set to zero. Being controlled by clock $C_1$, line D automatically enables bit $r$ input when the bits comprising $h_r$ are coming out of gate G. Circulating line D also controls the up-down mode in an obvious way using two OR gates.

Transfer of the final output to the digital-to-analog (D/A) converter poses an interesting problem. In principle, this could be done in two count cycles, one for copying out and one for zeroing the counter in preparation for the next word. However, this would entail interruption of the flow of input bits during these two cycles. We prefer to implement the data transfer and initialization without interrupting this flow. This can be achieved by the simple expedient of segregating the counter flip-flops into words L and H as shown in Fig. 9 and transferring the data from these two words at different times. Word L is copied out simultaneously with activating the gate connected to the lowest bit of word H. Zeroing of word L follows in the next count cycle. Word H is copied out immediately after its assembly is complete, that is, simultaneously with the entry (to the counter) of the first bit of the next filter output word. Zeroing of word H follows in the next count cycle.

To understand why this scheme works, note that the counter input gates are activated in a sequence leading from the low-order bits to the high-order bits. This means that by the time any gate connected to word H is activated, the assembly of word L is complete and it may be copied out. Furthermore, in assigning the partitioning into words H, L, we make sure that the lowest bit of H has an odd weight. This means that when the first H gate is accessed, the counter is in a down-counting mode. In this mode a $1 \rightarrow 0$ change in flip-flop $i$ does not affect flip-flop $i+1$. Hence, zeroing of word L has no effect on word H. Similarly, entry of the first two bits of the new word (with gate 0 activated) affects at most bit 1 and thus the copying out and zeroing of word H are not disturbed.

Note that this method would not be applicable in a system where the higher weights are computed first (as in Fig. 3). This is the motivation for the input bit sequence adopted here. The basic characteristics of this design, however, are independent of the input sequence adopted.

[9] A $D$-bit up-down counter will properly represent any integer $N$ constrained as follows:

$$-2^{D-1} \leq N \leq 2^{D-1} - 1$$

We turn now to some operational details. Figure 8 indicates that each filtering task will require a specific number of cells in the circulating lines $[KJ - (J_x - 1)]$ for the coefficients line and $[KJ - (J_x - 2)]$ for the data line. An important feature of a practical implementation would therefore be an array of line segments of various lengths which are to be switched into the circulating lines by the machine operator. The flexibility and possibilities of such an arrangement are quite attractive. Yet, all of this is obtained with a relatively modest investment in hardware.

The major disadvantage is of course the low speed. Let $T_g$ be the period of clock $C_2$ stepping the circulating lines. In discussing Fig. 8, we saw that a new input bit is fed with each revolution of the coefficients line. Hence,

$$T_b = \{KJ - (J_x - 1)\}\, T_g \tag{35}$$

and (see Eq. 23)

$$T_s = KJ^2 T_g \left\{1 - \frac{J_x - 1}{KJ}\right\} \tag{36}$$

Obviously the penalty for reducing the amount of hardware is quite severe.

We consider now briefly the question of floating point representation for the filter coefficients. When the range of coefficients results in a large $J_a$, the effect on the speed will be quite pronounced (Eq. 36). However, if the up-down counter is sufficiently large so as not to overflow under these conditions, the only hardware adjustments required are the patching together of sufficiently long circulating lines. Thus, the main argument for floating point representation is speed increase rather than hardware economy.

We give here only a rough outline of a floating point implementation based on approximating $a_k$ by $a'_k(-2)^{e_k}$ in which $e_k$ is a positive integer and $a'_k$ is an integer representable by $J_a$ bits in base $(-2)$. We store $\{a'_k\}$ in the data line and $\{e_k\}$ in an extra register used to control the access to the counter in such a way that a bit belonging to $h_r$ and generated by $a'_k$ is steered to input $(r + e_k)$. This guarantees the correct result. However, the resulting non-monotonic steering raises some difficulties which will not be discussed here.

## VI. Concluding Remarks

We have seen that the counting strategy provides the basis for a fast machine. How does such a design compare with one based on Eq. (4)? In comparing the two, coarse estimates of the amount of hardware will suffice. Using $r$-bit elementary counters to count $N$ gates in groups of $q$, the total number of counters is $N/(q - r)$.[10] Hence, the fast design using "full-adders" $(q = 3, r = 2)$ requires $N$ counters. On the other hand, to attain the same bit rate with serial multipliers applying Eq. (4), we have to use $K$ multipliers operating in parallel and $K$ accumulators[11] to sum their outputs. Each of these accumulators must handle at least $J_a$ bits and thus consist of at least $J_a$ "full-adders" and their associated storage elements. Hence, with $N = KJ_a$, the $K$ accumulators alone will use more hardware than the $N$ elementary counters of the counting design. The $K$ multipliers are therefore extra hardware to be weighed against the single fast accumulator of the counting design.

There is no question, therefore, that the counting design is more economical for the fast machine.[12] However, it should be realized that the partitioned counting approach is also applicable to slow designs, since increasing $q$ will drastically reduce both speed and cost. Thus, adopting $q$ as the basic design parameter, machines of widely different characteristics may be built.

All of the designs described here are conceptual entities. None have been practically implemented. They all look promising, each with its own particular balance of performance versus cost. Particularly intriguing in this context are the possibilities opened by large-scale integration. It remains to be seen, however, whether these expectations will actually materialize in a practical implementation.

---

[10] $(N/q)\{1 + (r/q) + (r/q)^2 + \cdots\} = N/(q - r)$

[11] $(K/2)\left(1 + \frac{1}{2} + \frac{1}{2^2} + \cdots\right) = K$

[12] It is also more precise (1 round-off error insertion versus $K$ such insertions).

## References


1. Pawlak, Z., and Wakulicz, A., "Use of Expansions With a Negative Basis in the Arithometer of a Digital Computer," *Bull. Acad. Polonaise Sci.*, Vol. V, No. 3, pp. 232–236, March 1957.
2. Wadel, L. B., "Negative Base Number Systems," *IRE Trans. Electron. Comput.*, Vol. EC-6, p. 123, June 1957.
3. Zohar, S., "Negative Radix Conversion," *IEEE Trans. on Comput.*, Vol. C-19, No. 3, pp. 222–226, March 1970.
4. Zohar, S., "A/D Conversion for Radix $(-2)$" (submitted to *IEEE Trans. on Comp.*).
5. Hill, F. J., and Peterson, G. R., "Introduction to Switching Theory and Logical Design," pp. 154–158. John Wiley & Sons, Inc., New York, 1968.

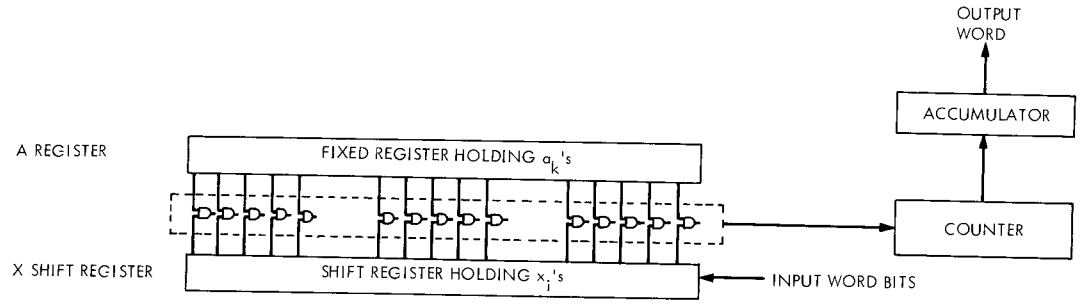


**Fig. 1. Preliminary outline of master design**

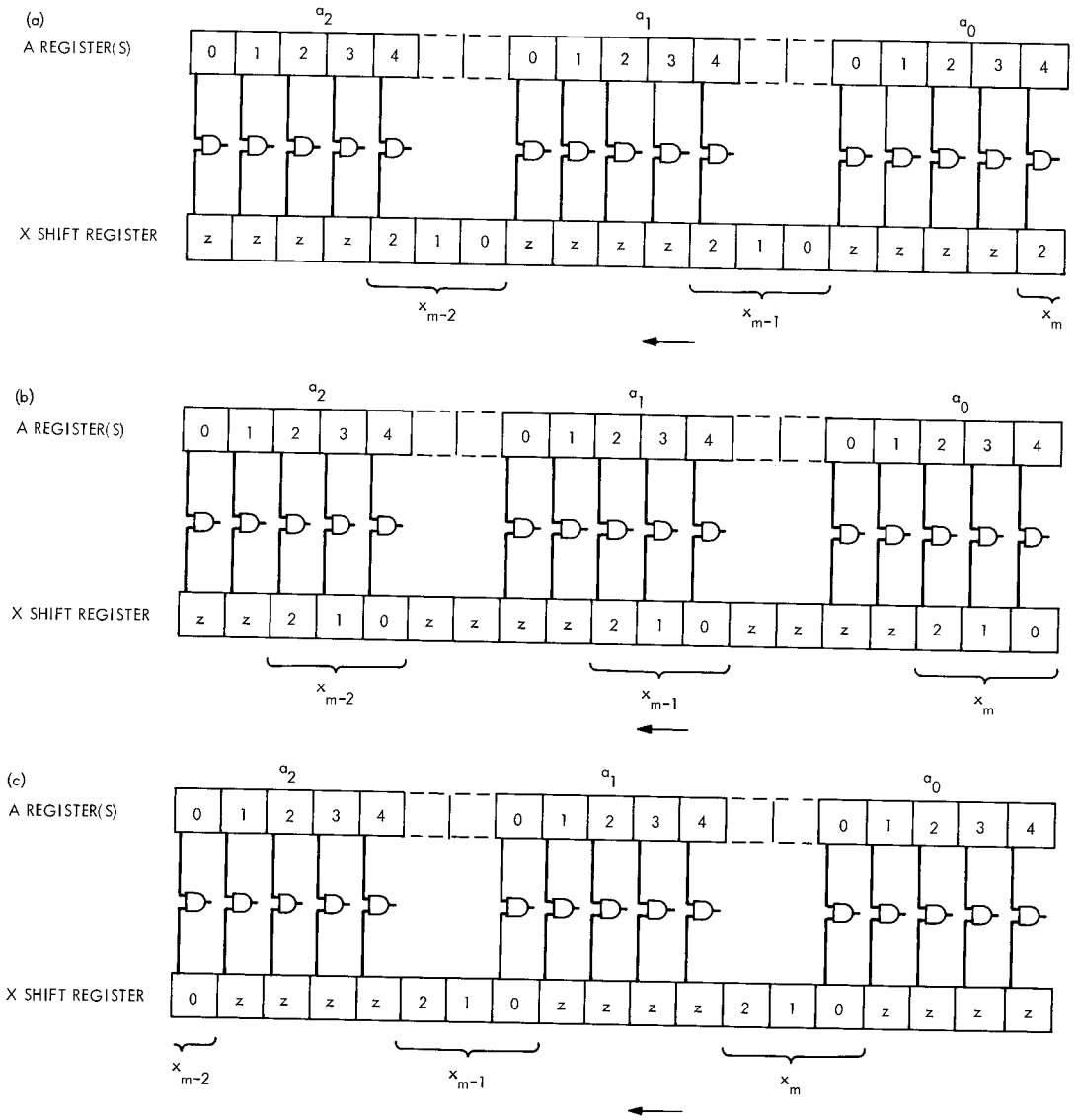


**Fig. 2. Bit sequences in registers *A*, *X***

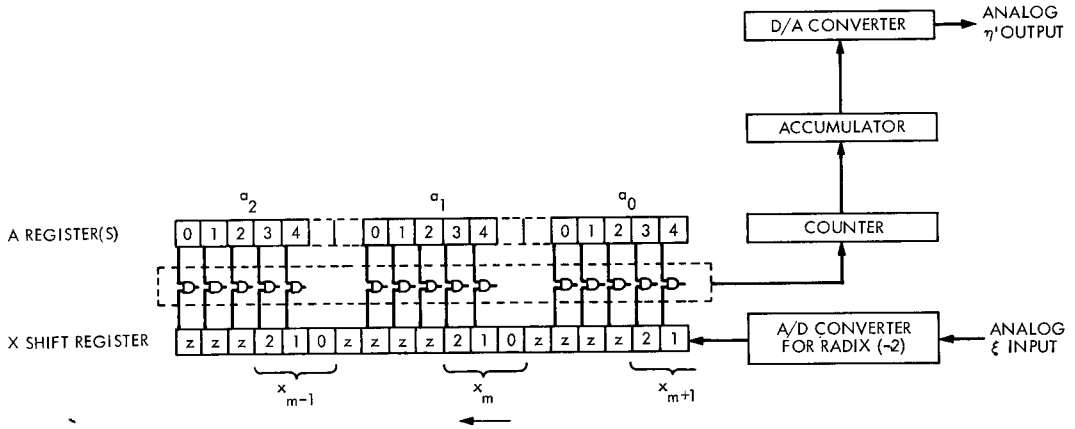


**Fig. 3. The master design**

| MULT | n \ k | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| $q^{\nu}$ | 0 | 1 | | | | | | |
| $q^{\nu-1}$ | 1 | 1 | 1 | 1 | | | | |
| $q^{\nu-2}$ | 2 | 1 | 2 | 3 | 2 | 1 | | |
| $q^{\nu-3}$ | 3 | 1 | 3 | 6 | 7 | 6 | 3 | 1 |

**Fig. 4. Initial stages of partitioned counting for $r = 3$**

| MULT | n \ k | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | CNTRS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $7^4$ | 0 | 1 | | | | | | | | | | | | $7^3$ |
| $7^3$ | 1 | 1 | 1 | 1 | | | | | | | | | | $7^2 \cdot 3$ |
| $7^2$ | 2 | 1 | 2 | 3 | 2 | 1 | | | | | | | | $7 \cdot 3^2$ |
| 7 | 3 | 1 | 3 | 6 | 7 | 6 | 3 | 1 | | | | | | $3^3$ |
| 1 | 4 | 1 | 4 | 10 | 16 | 19 | 16 | 10 | 4 | 1 | | | | 9 |
| 1 | 5 | 1 | 1 | 5 | 6 | 10 | 8 | 8 | 7 | 2 | | | | 5 |
| 1 | 6 | 1 | 1 | 1 | 7 | 5 | 3 | 4 | 3 | 4 | 1 | | | 1 |
| 1 | 7 | 1 | 1 | 1 | 1 | 6 | 4 | 4 | 3 | 4 | 1 | | | 1 |
| 1 | 8 | 1 | 1 | 1 | 1 | 1 | 5 | 5 | 3 | 4 | 1 | | | 1 |
| 1 | 9 | 1 | 1 | 1 | 1 | 1 | 1 | 6 | 4 | 4 | 1 | | | 1 |
| 1 | 10 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 5 | 5 | 1 | | | 1 |
| 1 | 11 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 6 | 2 | | | 1 |
| 1 | 12 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 1 | | 1 |
| 1 | 13 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | | 1 |
| 1 | 14 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |

TOTAL 602

**Fig. 5. Partitioned counting for $r = 3$; $\nu = 4$**

| MULT | n \ k | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | CNTRS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $3^7$ | 0 | 1 | | | | | | | | | | | | |
| $3^6$ | 1 | 1 | 1 | | | | | | | | | | | $3^6$ |
| $3^5$ | 2 | 1 | 2 | 1 | | | | | | | | | | $3^5 \cdot 2$ |
| $3^4$ | 3 | 1 | 3 | 3 | 1 | | | | | | | | | $3^4 \cdot 2^2$ |
| $3^3$ | 4 | 1 | 4 | 6 | 4 | 1 | | | | | | | | $3^3 \cdot 2^3$ |
| $3^2$ | 5 | 1 | 5 | 10 | 10 | 5 | 1 | | | | | | | $3^2 \cdot 2^4$ |
| 3 | 6 | 1 | 6 | 15 | 20 | 15 | 6 | 1 | | | | | | $3 \cdot 2^5$ |
| 1 | 7 | 1 | 7 | 21 | 35 | 35 | 21 | 7 | 1 | | | | | $2^6$ |
| 1 | 8 | 1 | 3 | 9 | 20 | 24 | 18 | 10 | 3 | | | | | 40 |
| 1 | 9 | 1 | 1 | 4 | 11 | 14 | 14 | 10 | 4 | 1 | | | | 28 |
| 1 | 10 | 1 | 1 | 2 | 6 | 9 | 10 | 8 | 5 | 2 | | | | 16 |
| 1 | 11 | 1 | 1 | 1 | 3 | 5 | 7 | 7 | 5 | 3 | | | | 12 |
| 1 | 12 | 1 | 1 | 1 | 1 | 4 | 4 | 5 | 5 | 2 | 1 | | | 8 |
| 1 | 13 | 1 | 1 | 1 | 1 | 2 | 3 | 4 | 4 | 3 | 1 | | | 4 |
| 1 | 14 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 3 | 2 | 2 | | | 5 |
| 1 | 15 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 3 | 2 | | | 3 |
| 1 | 16 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 1 | 3 | | | 2 |
| 1 | 17 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | | 2 |
| 1 | 18 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | | 1 |
| 1 | 19 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | | 1 |
| 1 | 20 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

TOTAL 2182

**Fig. 6. Partitioned counting for $r = 2$; $\nu = 7$**

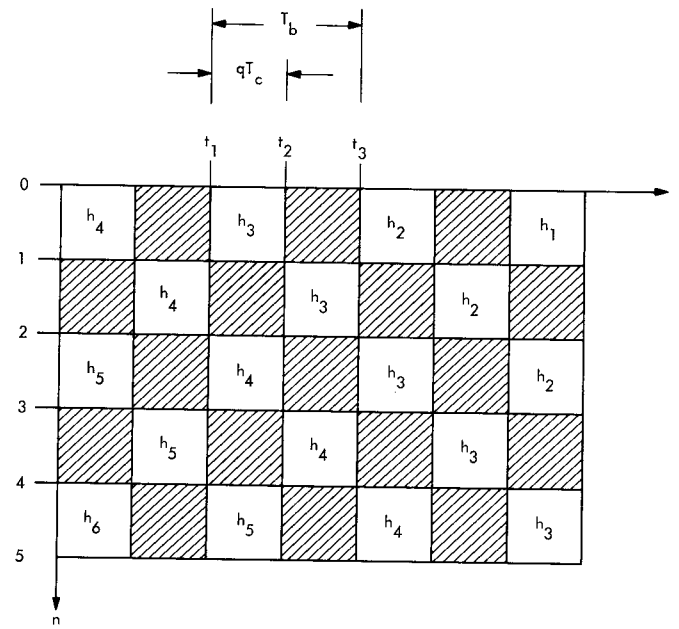


**Fig. 7. Time dependence in partitioned counting**

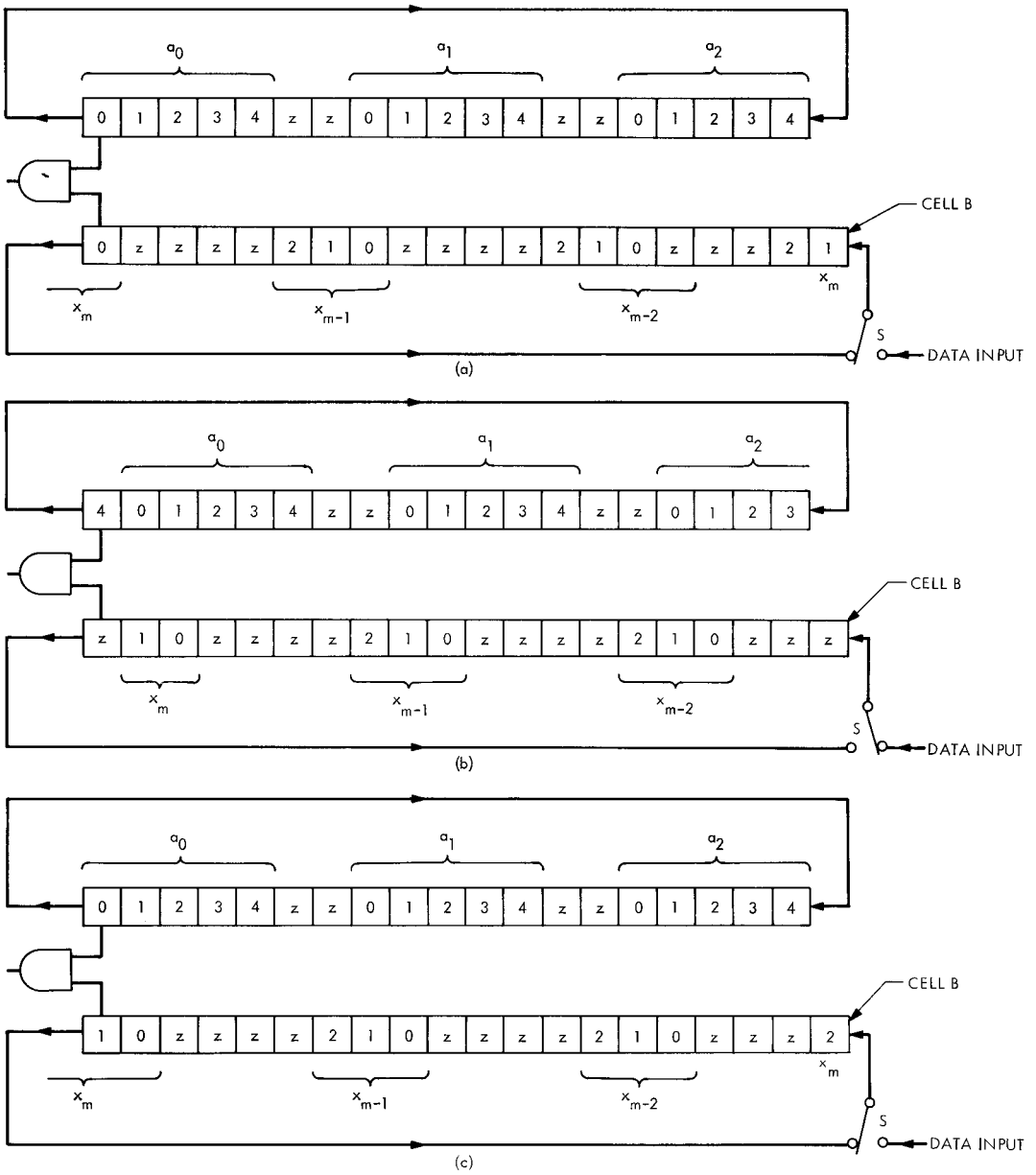


**Fig. 8. Bit sequences in the circulating lines filter**

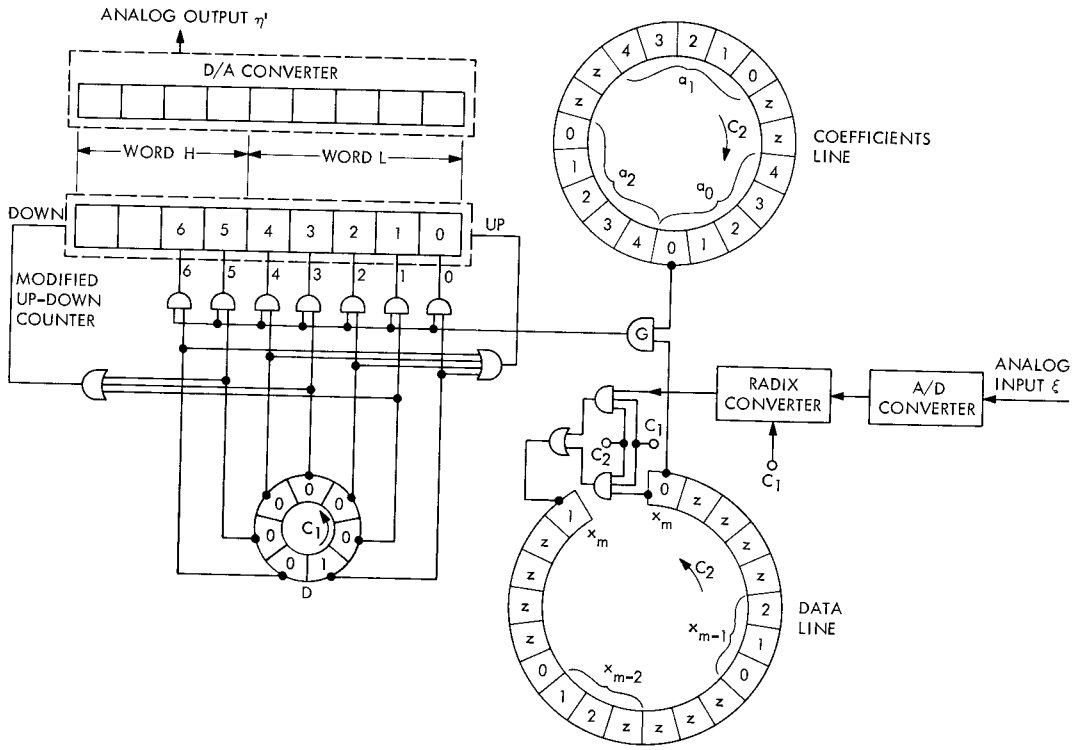


**Fig. 9. The circulating lines filter**