Using Single Buffers and Data Reorganization to Implement a Multi-Megasample Fast Fourier Transform

R. D. Brown
Communications Systems Research Section

Data ordering in large fast-Fourier transforms (FFT's) is both conceptually and implementationally difficult. This article describes a method of visualizing data orderings as vectors of address bits, which enables the engineer to use more efficient data orderings and reduce double-buffer memory designs to single-buffer designs. In particular, this article details the difficulties and algorithmic solutions involved in FFT lengths up to 4 megasamples (Msamples) and sample rates up to 80 MHz. Although the particular solutions mentioned may be directly applicable only to the particular system for which they were intended, the methodology by which these solutions were found could be useful to anyone confronted with similar problems.

I. Introduction

The Search for Extraterrestrial Intelligence (SETI) Program has recently completed the wire-wrap prototype wideband spectrum analyzer (WBSA) [1] and is about to start design of the sky-survey signal processor (SSSP) [2]. Both of these machines are high-speed fast-Fourier transform (FFT) processors followed by special-purpose signal-detection hardware. In the course of building the WBSA, a particular methodology grew for the design of FFT machines (memory boards, in particular). These methods have cut the memory requirements for the proposed SSSP system by about 50 percent.

This article begins with a brief description of the SSSP to familiarize the reader with the system from which examples will be drawn. The backbone of this article is the data-reordering scheme, which is followed by a section outlining the method of replacing double buffers with single buffers. Finally, the design of the SSSP's input buffer (INBUF) board is presented as an illustration of the design techniques.

II. Brief Description of the SSSP System

Figure 1 is a block diagram of one of the eight identical processors that will constitute the SSSP. Each processor of the system accepts complex samples at a rate of 80 MHz and performs an FFT on each group of 4 megasamples (Msamples). Although the processor's internal clock runs at only 40 MHz, the processor can accept data at 80 MHz because it has two input lines, the high-input line and the low-input line. The high data point is the one that was sampled first and held until the low point was sampled. Every 25 nsec (40-MHz rate), two sample points enter the processor, and two frequency values from a previous spectrum leave the processor at the back end of the pipe.

A pipeline configuration was chosen to facilitate real-time processing. To accommodate a 4-megapoint (Mpoint) FFT, the data are configured as a 4096-by-1024 matrix, and the FFT is broken into two orthogonal FFT's, one for the data in each column and one for the data in each row. Each FFT is further broken into radix-4 FFT stages. The two groups of these stages are called super-stages.

Proper FFT implementation demands that the first FFT super-stage be performed on points that have the largest possible sampling interval between them, which indicates the columns of the matrix. Accordingly, the INBUF board transposes the matrix before the first of the two FFT super-stages. The result of the transposition is a 1024-by-4096 matrix, in which the first new row contains the first point of all the old rows, the second new row contains the second points, and so on.

Using the convention that the matrix is always oriented so that FFT's are performed on rows, the first super-stage performs an FFT on each 4-kilopoint (kpoint) row. Then the matrix is transposed by the corner-turn memory (CTM) board to prepare it for the second super-stage. The 1-kpoint FFT's are performed on each of the new rows, resulting in a 4-Mpoint spectrum of frequency bins.

The real adjust (RADJ) board reorganizes (but does not transpose) these frequency values for its own purposes. The unscrambler (UMR) board undoes the shuffling done by the RADJ board and transposes the matrix for a final time, putting the frequencies in sequential order.

III. Relating Data Order to Addressing

Consider the sampled data stream as an array, in the case of the SSSP, a 2^{22}-point array. As such, each data point can be specified by a 22-bit index number. Since the SSSP is a dual-rail system, 2^{21} of the samples come in on the high rail, and the other 2^{21} enter on the low rail. The 22-bit index number can be divided as follows: One bit, known as the hi/lo bit, indicates which rail the data came in on, 0 for high, 1 for low. (As a convention, this bit is always the least-significant bit, lsb.) The remaining 21 bits form a 21-bit binary counter, which increments with each clock cycle. As a design convention, the bits of this data counter are labeled C_0 (lsb) through C_{20}, the most-significant bit (msb).

At each clock cycle in Fig. 2, two data points enter the SSSP, one on the high data rail and one on the low data rail. Each of these data points has its own 22-bit index number, and they must be different. Because they enter the board at the same time, the counter bits (the most-significant 21 bits) are the same. The only difference is the hi/lo bit, which will be a 0 for the data point on the high rail and a 1 for the data point on the low rail. Figure 2 depicts this graphically for three clock cycles in the middle of a spectrum.

At each of the three clock cycles, the value of the time counter is given in both decimal and binary form. Below the time counter are sections of the high- and low-data channels, one data point per channel per clock cycle. The data-order index of each data point is given in decimal and binary forms inside the data box. Because data in this example are in sequential order, the bits in the time counter are in the same order as they appear in the data-order index. If a different order had been chosen, the bits would have been scrambled, but sequential order is easiest to visualize. In sequential order, the index for data on the high channel will always be exactly double the time counter, and the index for data on the low channel will always be one more than the simultaneous data on the high channel.

Once this 22-bit index number is established for a given data point and the addressing algorithm is known, everything that happens to that data point is also known. The data point is stored in a memory array using the 22-bit number as an address, so it is known where the data are stored. When the data are read out of memory, their channels (high or low) and their positions in the data stream are based on those same 22 bits. In the 4-bit example in Fig. 3, data (in the boxes) enter in sequential order. Above each box is its 4-bit data-order index. The 3-bit time counter always matches the 3 left-most index bits, and the right-most bit indicates the high or low channel. The resultant pattern, A_0 through A_3, differentiates each data point by identifying its position in the sequential data stream (0–15).

As an example, these points are output at the bottom of Fig. 3 in high–low (HL) order. An N-point data stream in HL order looks like:

$$\begin{aligned} \text{HI CHANNEL:} \quad & 0,\ 1,\ 2, \\ & 3,\ 4,\ 5,\ \cdots \\ \text{LO CHANNEL:} \quad & N/2,\ N/2+1,\ N/2+2, \\ & N/2+3,\ N/2+4,\ N/2+5,\ \cdots \end{aligned}$$

To organize data points in HL order, the msb of the data-order index moves to the least-significant position. The other bits are all in sequential order, but they are shifted to the left by one bit. Again, the time counter matches with the left-most 3 index bits, but now those bits are A_2, A_1, and A_0. (They were A_3, A_2, and A_1 in sequential order.) The right-hand bit that indicates the high or low channel is now A_3. However, when these bits are written in their correct numerical order, $A_3A_2A_1A_0$,

they are the correct binary representation of each data-order index (the large numbers inside the boxes).

In general, any data-reordering scheme (from sequential order to HL order in the above example) can be viewed as a transformation of the data-index values. In this way, the SSSP data index is viewed as a 22-bit vector and the transformation as a 22-by-22 matrix. The resultant new vector is composed of 22 new data-index values. Each new index value is a function of one or more of the original index values. For simple reordering (as in the above example), each new index value is a function of exactly one old index value, and the transformation is simply a reordering of the bits of the data index. Other, more complex, transformations are also useful.

IV. Addressing Single Buffers and Double Buffers

The SSSP is based on the decision to use single buffers instead of double buffers wherever possible, sacrificing simplicity for the sake of hardware savings, board space, and lower power consumption. In the case of the INBUF board, which will be discussed later, this amounts to 8 Mbytes of memory, saving $8400 and a maximum of 18 W for each of the 8 copies that will be made. Savings for the CTM board are treble this, a net total of over $200,000 and 384 W for the 8 copies of the CTM.

Figure 4 shows the way in which the use of single buffers differs from double buffers. In each example, the WRITE line tells which spectrum is being written at any particular time, and the READ line tells which one is being read. (Notice that the first spectrum to be read is unlabeled because it contains no real data and is discarded.) Below the WRITE and READ lines, the addressing lines tell the addressing used for each buffer.

Notice that when using a double buffer, only two addressing orders are required, a read order and a write order. Whenever a spectrum is being read out of one buffer (using the read addressing), another spectrum is being written to the other buffer (using the write addressing). Then the latter buffer is read, while a third spectrum is being written into the former buffer.

For simplicity's sake, it is often convenient to write the incoming data in sequential order and read them out in transformed order; call the transformed order T. If the order of the data can be written as an N-point vector, then the transformation between two orders can be written as an N-by-N matrix. Thus, each data order in Fig. 4 is labeled as sequential or as a power of the transformation matrix, T, a 5-by-5 matrix. The columns of each matrix correspond to the different address lines of the memory buffer, A to E from top to bottom. The order of these columns mirrors the order of the address bits in the 5-bit data-order index.

It is important to realize that if data are to be read out in sequential order after the transformation, they would have to be written in inverse-T order. This is a very important basic concept: *Data can be written in any order, as long as the transformation between the written order and the read order is T.* For example, if instead of being written sequentially, the data order were transformed so that data were written in W order, they would have to be read out in R order such that $R = T * W$.

This idea is the basis for the single-buffer addressing schemes. Using read-write cycles, in which a given address is read then overwritten, the first spectrum is written sequentially. However, when it is read out later (again using read-write cycles), the second incoming spectrum must be written in the same order that the first is going out. (If new data were written in a different order, some memory locations would be rewritten before they were read.) This means that when Spectrum 0 is read out in transformed order, T, Spectrum 1 is being written in T order.

This is the problem. If the second spectrum is read out in T order, as in the double-buffer example, it would appear in the same order it went in, sequentially. So it must be read out in $T * T$ order. This means that Spectrum 2 will be written in $T * T$ order and read in $T * T * T$ order, Spectrum 3 will be written in $T * T * T$ order, and so on. Since there are only a finite number of ways to reorganize the data stream, the pattern will eventually repeat. Reworded in more rigorous terms, a finite P exists such that $T^P = T^0 = I$, where I is the 5-by-5 identity matrix representing sequential order. However, if P is very large, the cycle will take a long time to repeat, requiring extra counters, and leading to complicated address equations and confused engineers. The trick is to simplify the address patterns as much as possible by minimizing P.

V. Interleaving

This section deals with the practical constraints of the present technology. The first difficulty one is likely to run into is the speed of available memories. The particular memory modules used in this processor have a 100-nsec read/write cycle time. (Faster memories are available but not in sizes that are practical for the amount of data required.) Because the clock period is 25 nsec, new data can

be written only every fourth clock cycle. In the meantime, data must be stored in registers until the start of the next 100-nsec cycle (see Fig. 5). Because 2 new points are ready to be written each clock cycle (25 nsec), 8 points accumulate every 100 nsec. All 8 data points must be written at the same time, requiring 8 memory modules operating in parallel (see Fig. 6).

Likewise, 8 data points are read out during each 100-nsec read/write cycle. Two data points are selected by the multiplexer during each of the next four clock cycles. Meanwhile, another 8 points are being read, and the cycle repeats. The following examples will refer to writing data, but it is to be understood that reading data is an analogous operation that is happening simultaneously.

If the sequential example from Fig. 3 were implemented using Fig. 6, the first point on the high channel would be stored in the H0 memory slice, the second point on the high channel in the H1 memory slice, and so on. Something must designate which memory slice a point will be written to. In this simple example, the answer is obvious. The bottom 2 counter bits can be fed directly to the demultiplexer to specify where to steer the data. (This is a slight oversimplification. In actual implementation, the input registers must include clock enables that depend on the same 2 bits.)

More complex situations demand a more formal approach. The 8 memory slices can be distinguished with a 3-bit number, $M = (M_2, M_1, M_0)$. Since each group of 8 data points is written simultaneously, each point within the group must be written to a different memory slice. Thus, the 8 different data indices must each designate a different memory. It is known that these 8 points all arrived within 4 clocks of each other. That means that all the bits of the indices will be the same except for the 2 least-significant (fastest changing) counter bits and the hi/lo bit. These 3 bits must all be what are called memory-selection bits. The memory-slice number, M, is a function of those 3 bits, and the 8 possible values of the 3 bits must correspond to every possible value of M. To clarify notation, M is the 3-bit number that specifies the memory slice, M_{number} is one of the bits of M, and M_{letter} is a memory selection bit within the data index number. Each M_{number} is a function of one or more M_{letter} bits.

Let the hi/lo bit be named M_a, the least-significant counter bit M_b, and the second counter bit M_c. In the example of Fig. 3, the simplest solution is

$$M_2 = M_a$$

$$M_1 = M_c$$

$$M_0 = M_b$$

Under this scheme, the data from the high channel will fill memory slices 0 to 3 in sequential order, and the data from the low channel will fill memory slices 4 to 7 in order. However, it is perfectly reasonable to use a different ordering. For example, the functions could have been defined

$$M_2 = M_a$$

$$M_1 = M_b$$

$$M_0 = M_b \text{ ^ } M_c$$

where ^ is taken to be the C-language symbol for "exclusive-or."

At time $t = 0$, the high-channel data will be written to memory slice 0 and the low channel to slice 4. At $t = 1$, $M_b = 1$ and $M_c = 0$, so the high-channel data will be written to slice 3 and the low channel to slice 7. At $t = 2$, high-channel data will be written to slice 1 and low data to slice 5. Finally, at $t = 3$, high and low data are written to slices 2 and 6, respectively. The cycle repeats at times $t = 4, 8, 12$, and so on.

VI. INBUF Board

The first (and simplest) memory board in the system is the INBUF board. Data enter sequentially:

HI CHANNEL : 0, 2, 4, 6, 8, ···

LO CHANNEL : 1, 3, 5, 7, 9, ···

This ordering can be represented as 22 address bits, labeled A to V, with V being the least-significant (hi/lo) bit (see Fig. 7). The data can also be looked at as being in a large 4096-by-1024 matrix, without altering the ordering at all. Step 1 is a purely conceptual step. The most-significant 12 bits designate the row number. The least-significant 10 bits specify the column (or position within the row). The row bits are separated from the column bits by a hyphen. As a standard convention, the least-significant bits are always row bits unless otherwise specified.

To fully reap the benefits of doing an FFT, the first stage operates on the samples that are most widely separated in time [3]. Subsequent stages work on data points nearer each other in time, and the last stage operates on sequential samples. In terms of the matrix FFT, the first super-stage should operate on the vector that contains samples $(S_0, S_{1k}, S_{2k}, S_{4k}, S_{6k}, \cdots S_{4Meg-1k})$. This is the first column of the matrix. (The first super-stage operates, of course, on each individual column, not just the first.)

The FFT boards that follow the INBUF operate on groups of 4 kpoints of data at a time. To minimize the amount of memory required within the FFT boards, those 4 kpoints should enter sequentially, so the INBUF board transposes the matrix in Step 2. The old row bits become column bits and vice versa.

The next reorganizational step (Step 3) is dictated by a knowledge of the inner workings of the FFT boards. They have been designed to do a separate FFT on each channel, with no data being shared between the high and low channels. Thus, a given row appears on one channel or the other, not both. This requires that the hi/lo bit be a row bit. The least-significant row bit moves to the hi/lo position.

Step 4 also depends upon the particular architecture of the FFT boards. They contain cascaded stages of radix-4 FFT's, which do 4-point FFT's on groups of the incoming data. The first stage does its work on the following quadruples:

$$(S_0,\ S_{1Meg},\ S_{2Meg},\ S_{3Meg}),$$

$$(S_1, S_{1Meg+1}, S_{2Meg+1}, S_{3Meg+1}),$$

$$(S_2, S_{1Meg+2}, S_{2Meg+2}, S_{3Meg2}), \text{etc.}$$

The order in which these quadruples appear is not important, but the order of the points within each quadruple is. So the two most-significant column bits move to the least-significant positions, creating what is known as high-low radix-4 order.

This completes the required reordering of the INBUF board. If data are written sequentially, A through V (as at the top of Fig. 7), and read using the addressing scheme at the bottom of the figure, M through V, all the necessary operations will be performed in a single step. Unfortunately, if this addressing pattern is implemented using a single buffer, the pattern only repeats every 21 spectra (as seen in Fig. 8). The worst feature of this particular design is that each address bit appears in a different column each spectrum. Thus, each and every address bit is a function of 21 different data-counter bits and a 5-bit spectrum counter, yielding huge unwieldy logic equations. With a little ingenuity, this can be simplified greatly.

Figure 9 demonstrates the design process of an efficient scheme. Step 1 is the same as before, and Steps 2 and 3 are similar. As noted before, the order in which the quadruples come is irrelevant. In fact, the order in which the rows come is also irrelevant. The only bits that are fixed are the 2 least-significant column bits, A and B. The other column bits are all interchangeable, as are the row bits, so they are all left blank. (Column bits are not interchangeable with row bits because a complete row must go into each channel of the FFT before the next row enters. This ensures that a separate FFT is being done on every single row. Thus, row and column bits are specified as such although they are not specifically identified.)

The goal of this method is to keep the addressing equations as simple as possible; whenever possible, address bits will remain in the same locations. This is possible only in the cases of the 3 least-significant row bits (which become the 3 most-significant column bits) and the least-significant column bit (which becomes the least-significant row bit). These bits are J, K, L, and V, respectively. In Step 5, these bits are assigned to remain in their respective positions.

A and B are the only bits that are constrained to move to particular positions. To minimize the cycle number, N, these two bits must return to their original positions as soon as possible, preferably after the next cycle. Thus, it is desirable that whichever bits lie in the second and third positions from the right in one spectral addressing will move to the two most-significant positions one spectrum later. By extension, T and U, which were in the second and third positions before the transformation, are required to be in the most-significant positions after the transformation. See Step 6.

By Step 7, all the constraints of the INBUF board have been satisfied, so a column order that is beneficial to the FFT boards is used. (This particular order allows the four FFT boards that comprise the two stages of the FFT to be built as identical copies of each other and minimizes the amount of memory needed to reorganize the data between each internal stage.) Once this order has been determined, the rest of the design follows from matching pairs of address bits in the same manner that A and T were matched and B and U were matched. This ordering scheme repeats after two spectra, which can easily be verified.

Now the design of the transformation is complete, but the implementational design remains. The implementational phase occurs when the 22 data-index bits, originally dubbed A through V, are designated as memory-selection bits or addressing bits. Because the design requires an eight-way memory interleave, three of the bits will be reserved for memory selection. The memories are each 512 kpoints deep, requiring 19 addressing bits (although some of these will also be used for memory selection).

Figures 10 and 11 describe the process of renaming the data-index bits. A bit chosen as an addressing bit is renamed A_{number}, and a memory-selection bit is renamed M_{letter}. These are just new names for the bits that were previously designated A through V. The old names were only placeholders; the new names also give information about how the transformation will be implemented.

There is a lot of freedom here, but the most straightforward choice is to assign the 3 memory-selection bits in the right-hand positions and the 19 address bits in order to the left (see Fig. 10). This works fine for spectrum 0. However, when the address bits have been transformed for spectrum 1, only one memory-selection bit remains in the three right-hand spots. This means that when a group of 8 data points comes in, the board will attempt to write them all to only two memory slices.

This problem is alleviated by stipulating that A and B must also be memory-selection bits as well as address bits (see Fig. 11). M_0 is now a function of both A and T, "exclusive-or"-ing bit A with bit T. This ensures that out of each group of 8 data points, half will be directed to even memory slices and half to odd memory slices. Likewise, B is "exclusive-or"-ed with U to determine M_1. Now each group of four clocks (100 nsec) accesses all eight memory slices. Note: Only groups that start on a 100-nsec boundary ($C_1 = C_0 =$ hi/lo $= 0$) need to be considered.

Now the design of the INBUF is complete. The Boolean equations for each address bit can be read directly off of Fig. 11. For instance, A_0 is equivalent to C_2 during spectrum 0, and it is equivalent to C_{17} during spectrum 1:

$$A_0 = (C_2 * S') + (C_{17} * S)$$

where S is the 1-bit spectrum counter, C_{number} is a bit selected from the 21-bit data counter, $'$ is logical "not," $+$ is logical "or," and $*$ is logical "and." Similar equations can be found for A_1 to A_{16}.

Notice that two address bits occur in the least-significant three places during spectrum 1. This causes some conceptual difficulties and requires different address lines to different slices of memory. But remember that A_{18} is defined as being equivalent to M_e, which in turn is equivalent to M_0 ^ M_c, where ^ is "exclusive-or." Thus:

$$A_{18} = C_{20}$$

for memory slices 0, 2, 4, and 6, and

$$A_{18} = (C_{20} * S') + (C_{20'} * S)$$

for memory slices 1, 3, 5, and 7. Equations for A_{17} can be similarly derived.

VII. Different Transformations

All previously mentioned transformations reorganized the address bits but leave the bits unchanged. Sometimes, the designer may wish to read out the data using address bits that are functions of, but not identical to, the original address bits. The simplest example is reverse order. If data are written sequentially, using address bits A, B, and C, they can be read out in reverse order by just inverting the address bits, as shown below:

Counter (A,B,C)	000	001	010	011	$\cdots$
Data in	0	1	2	3	$\cdots$
Reverse counter (A',B',C')	111	110	101	100	$\cdots$
Data out	7	6	5	4	$\cdots$

Another useful transformation is the downcounter transformation, which is easily implemented by substituting a downcounter for the regular counter. The bits of the downcounter are indicated as $\tilde{A}$, $\tilde{B}$, $\tilde{C}$.

Counter (A,B,C)	000	001	010	011	$\cdots$
Data in	0	1	2	3	$\cdots$
Downcounter $(\tilde{A},\tilde{B},\tilde{C})$,	000	111	110	101	$\cdots$
Data out	0	7	6	5	$\cdots$

These two operations have one thing in common; they are reversible. That means that after the address bits are transformed, no information is lost; the original address bits can still be recovered. Reversibility is important because it prevents memory slices from attempting to write

multiple data points to the same spot. There are many reversible operations, such as "exclusive-or"-ing two bits (as in Fig. 11).

VIII. Limitations

The use of single buffers is limited by two considerations. First, the interleaving required to implement read/write cycles at the proper speed may require as much memory as a double buffer. In such a case, no savings would be realized by using a single buffer.

The other consideration is the complexity of the addressing, which may rise to a point of impracticality.

IX. Conclusion

Visualizing a data stream as a vector of addressing bits allows the designer to treat data reorganization as a matrix transformation on that vector. This allows the designer to easily manipulate many transformations at once and find data orderings that are beneficial to other boards in the system. It also allows the designer to find address patterns that make single-buffer memory banks possible (as opposed to double-buffer banks), further reducing the necessary hardware.

Once an address bit transformation has been chosen, the implementation can be easily designed using methods outlined in this article.

References

[1] M. Quirk, M. Garyantes, H. Wilck, and M. Grimm, "A Wide-Band High-Resolution Spectrum Analyzer," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, no. 12, pp. 1854–1861, December 1988.

[2] M. Garyantes, M. Grimm, and G. Zimmerman, "A Wideband High Resolution Digital Spectrum Analyzer for the Search for Extraterrestrial Intelligence," in preparation for the 1992 IEEE International Conference on Acoustics, Speech, and Signal Processing.

[3] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Englewood Cliffs, New Jersey: Prentice-Hall, pp. 356–371, 1975.

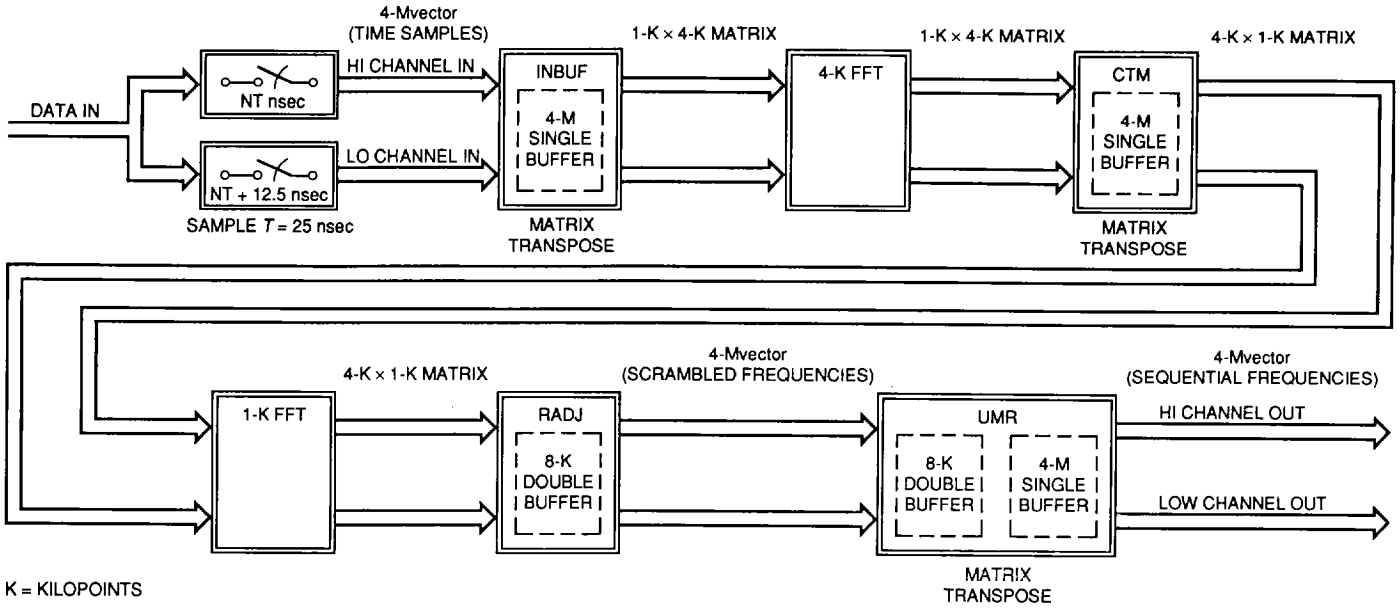

Fig. 1. SSSP.

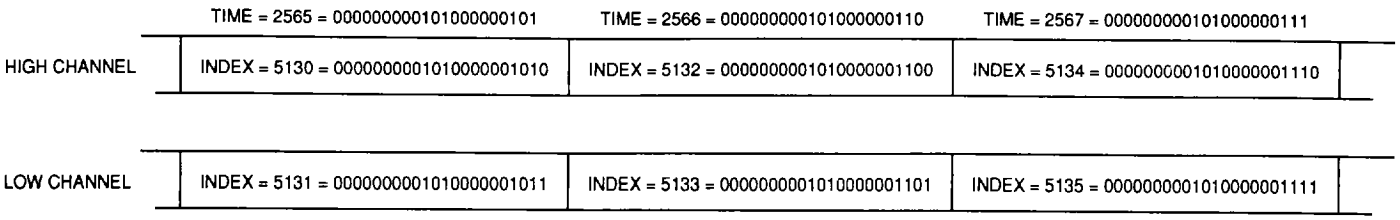

Fig. 2. Relating the time counter to the index number.

INPUT – SEQUENTIAL ORDER ($A_3\ A_2\ A_1\ A_0$)

	T = 000	T = 001	T = 010	T = 011	T = 100	T = 101	T = 110	T = 111
HIGH CHANNEL (HI/LO = 0) $A_3\ A_2\ A_1\ A_0$ =	0000	0010	0100	0110	1000	1010	1100	1110
	0	2	4	6	8	10	12	14
LOW CHANNEL (HI/LO = 1) $A_3\ A_2\ A_1\ A_0$ =	0001	0011	0101	0111	1001	1011	1101	1111
	1	3	5	7	9	11	13	15

OUTPUT – HL ORDER ($A_2\ A_1\ A_0\ A_3$)

	T = 000	T = 001	T = 010	T = 011	T = 100	T = 101	T = 110	T = 111
HIGH CHANNEL (HI/LO = 0) $A_2\ A_1\ A_0\ A_3$ =	0000	0010	0100	0110	1000	1010	1100	1110
	0	1	2	3	4	5	6	7
LOW CHANNEL (HI/LO = 1) $A_2\ A_1\ A_0\ A_3$ =	0001	0011	0101	0111	1001	1011	1101	1111
	8	9	10	11	12	13	14	15

Fig. 3. Relating input order to a binary counter.

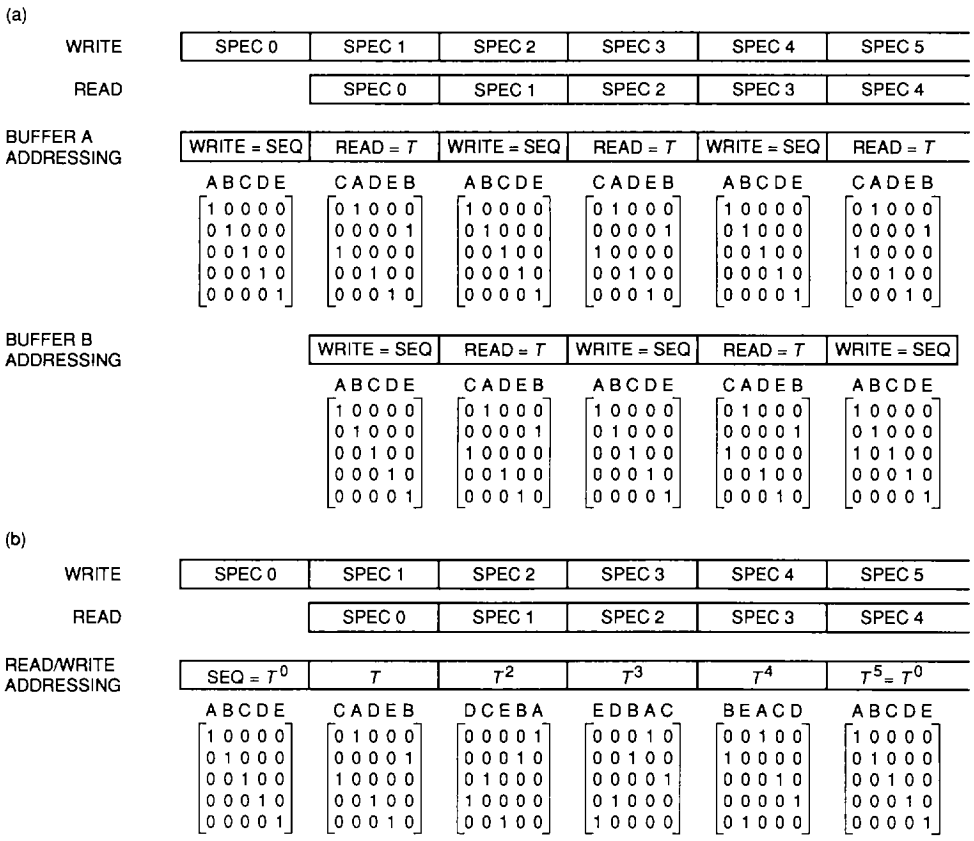

Fig. 4. How to address memories: (a) using a double buffer and (b) using a single buffer.

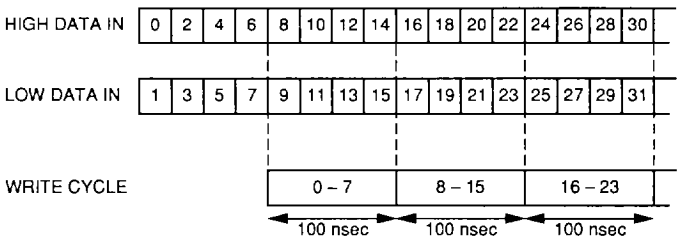

Fig. 5. Timing of eight-way interleave.

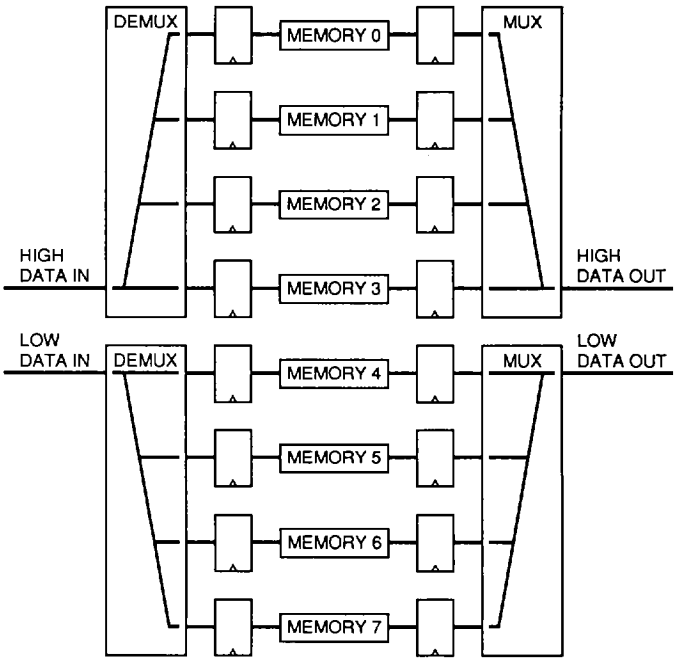

Fig. 6. Memory buffer divided into eight slices.

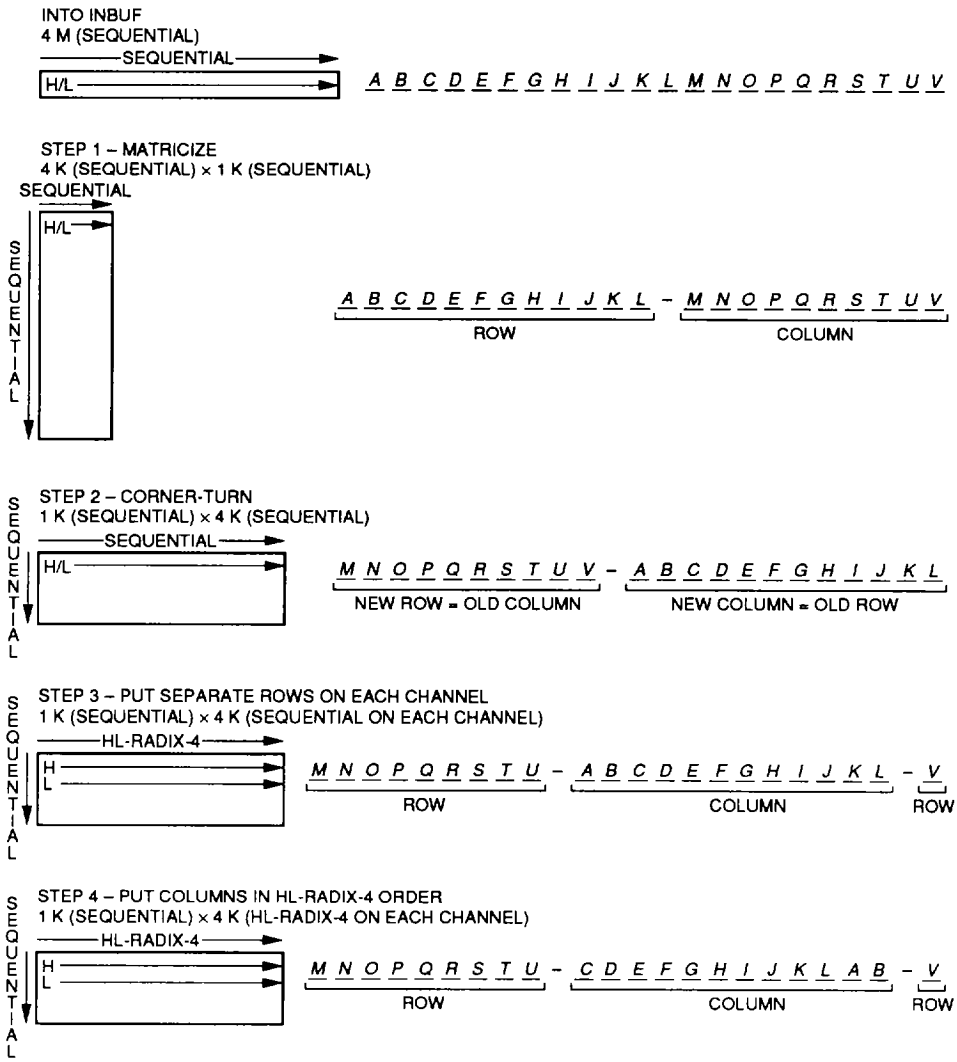

Fig. 7. INBUF data reordering.

	C_{20}	C_{19}	C_{18}	C_{17}	C_{16}	C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0	HI/LO
SPECTRUM = 0	***A***	***B***	***C***	***D***	***E***	***F***	***G***	***H***	***I***	***J***	***K***	***L***	***M***	***N***	***O***	***P***	***Q***	***R***	***S***	***T***	***U***	***V***
SPECTRUM = 1	*M*	*N*	*O*	*P*	*Q*	*R*	*S*	*T*	*U*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*A*	*B*	*V*
SPECTRUM = 2	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*A*	*B*	*O*	*P*	*Q*	*R*	*S*	*T*	*U*	*C*	*D*	*E*	*M*	*N*	*V*
SPECTRUM = 3	*R*	*S*	*T*	*U*	*C*	*D*	*E*	*M*	*N*	*H*	*I*	*J*	*K*	*L*	*A*	*B*	*O*	*P*	*Q*	*F*	*G*	*V*
SPECTRUM = 4	*K*	*L*	*A*	*B*	*O*	*P*	*Q*	*F*	*G*	*T*	*U*	*C*	*D*	*E*	*M*	*N*	*H*	*I*	*J*	*R*	*S*	*V*
SPECTRUM = 5	*D*	*E*	*M*	*N*	*H*	*I*	*J*	*R*	*S*	*A*	*B*	*O*	*P*	*Q*	*F*	*G*	*T*	*U*	*C*	*K*	*L*	*V*
SPECTRUM = 6	*P*	*Q*	*F*	*G*	*T*	*U*	*C*	*K*	*L*	*M*	*N*	*H*	*I*	*J*	*R*	*S*	*A*	*B*	*O*	*D*	*E*	*V*
SPECTRUM = 7	*I*	*J*	*R*	*S*	*A*	*B*	*O*	*D*	*E*	*F*	*G*	*T*	*U*	*C*	*K*	*L*	*M*	*N*	*H*	*P*	*Q*	*V*
SPECTRUM = 8	*U*	*C*	*K*	*L*	*M*	*N*	*H*	*P*	*Q*	*R*	*S*	*A*	*B*	*O*	*D*	*E*	*F*	*G*	*T*	*I*	*J*	*V*
SPECTRUM = 9	*B*	*O*	*D*	*E*	*F*	*G*	*T*	*I*	*J*	*K*	*L*	*M*	*N*	*H*	*P*	*Q*	*R*	*S*	*A*	*U*	*C*	*V*
SPECTRUM = 10	*N*	*H*	*P*	*Q*	*R*	*S*	*A*	*U*	*C*	*D*	*E*	*F*	*G*	*T*	*I*	*J*	*K*	*L*	*M*	*B*	*O*	*V*
SPECTRUM = 11	*G*	*T*	*I*	*J*	*K*	*L*	*M*	*B*	*O*	*P*	*Q*	*R*	*S*	*A*	*U*	*C*	*D*	*E*	*F*	*N*	*H*	*V*
SPECTRUM = 12	*G*	*A*	*U*	*C*	*D*	*E*	*F*	*N*	*H*	*I*	*J*	*K*	*L*	*M*	*B*	*O*	*P*	*Q*	*R*	*G*	*T*	*V*
SPECTRUM = 13	*L*	*M*	*B*	*O*	*P*	*Q*	*R*	*G*	*T*	*U*	*C*	*D*	*E*	*F*	*N*	*H*	*I*	*J*	*K*	*S*	*A*	*V*
SPECTRUM = 14	*E*	*F*	*N*	*H*	*I*	*J*	*K*	*S*	*A*	*B*	*O*	*P*	*Q*	*R*	*G*	*T*	*U*	*C*	*D*	*L*	*M*	*V*
SPECTRUM = 15	*Q*	*R*	*G*	*T*	*U*	*C*	*D*	*L*	*M*	*N*	*H*	*I*	*J*	*K*	*S*	*A*	*B*	*O*	*P*	*E*	*F*	*V*
SPECTRUM = 16	*J*	*K*	*S*	*A*	*B*	*O*	*P*	*E*	*F*	*G*	*T*	*U*	*C*	*D*	*L*	*M*	*N*	*H*	*I*	*Q*	*R*	*V*
SPECTRUM = 17	*C*	*D*	*L*	*M*	*N*	*H*	*I*	*Q*	*R*	*S*	*A*	*B*	*O*	*P*	*E*	*F*	*G*	*T*	*U*	*J*	*K*	*V*
SPECTRUM = 18	*O*	*P*	*E*	*F*	*G*	*T*	*U*	*J*	*K*	*L*	*M*	*N*	*H*	*I*	*Q*	*R*	*S*	*A*	*B*	*C*	*D*	*V*
SPECTRUM = 19	*H*	*I*	*Q*	*R*	*S*	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*T*	*U*	*J*	*K*	*L*	*M*	*N*	*O*	*P*	*V*
SPECTRUM = 20	*T*	*U*	*J*	*K*	*L*	*M*	*N*	*O*	*P*	*Q*	*R*	*S*	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*V*
SPECTRUM = 0	***A***	***B***	***C***	***D***	***E***	***F***	***G***	***H***	***I***	***J***	***K***	***L***	***M***	***N***	***O***	***P***	***Q***	***R***	***S***	***T***	***U***	***V***

Fig. 8. INBUF transformation sequence.

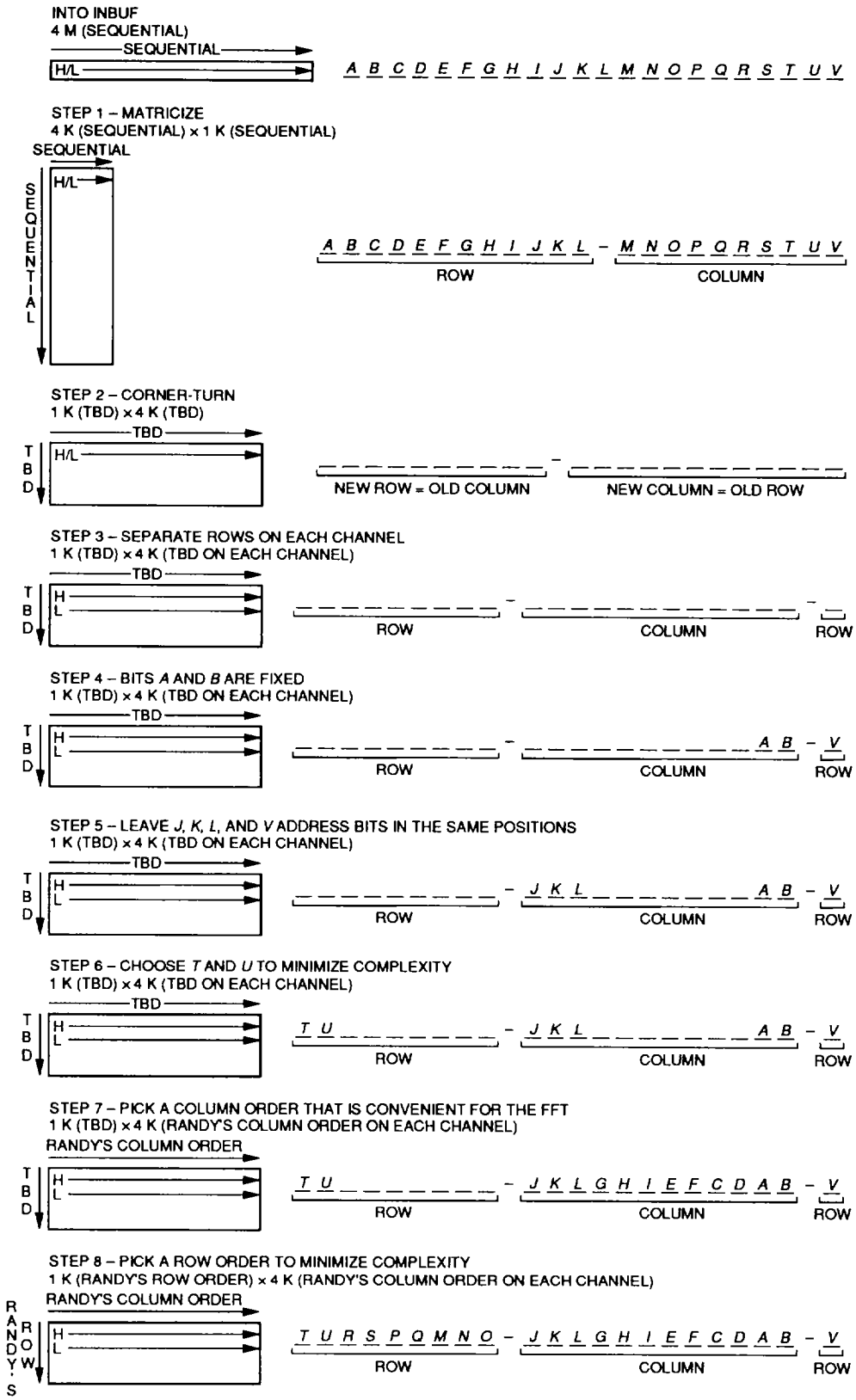

Fig. 9. Revised INBUF data reordering.

	C_{20}	C_{19}	C_{18}	C_{17}	C_{16}	C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0	HI/LO
SPECTRUM = 0	***A***	***B***	***C***	***D***	***E***	***F***	***G***	***H***	***I***	***J***	***K***	***L***	***M***	***N***	***O***	***P***	***Q***	***R***	***S***	***T***	***U***	***V***
	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	M_c	M_b	M_a
									$M_2 = M_a$													
									$M_1 = M_b$													
									$M_0 = M_c$													
SPECTRUM = 1	*T*	*U*	*R*	*S*	*P*	*Q*	*M*	*N*	*O*	*J*	*K*	*L*	*G*	*H*	*I*	*E*	*F*	*C*	*D*	*A*	*B*	*V*
	M_c	M_b	A_1	A_0	A_3	A_2	A_6	A_5	A_4	A_9	A_8	A_7	A_{12}	A_{11}	A_{10}	A_{14}	A_{13}	A_{16}	A_{15}	A_{18}	A_{17}	M_a
									$M_2 = M_a$													
									$M_1 = M_b$													
									$M_0 = M_c$													

Fig. 10. Revised INBUF transformation sequence.

	C_{20}	C_{19}	C_{18}	C_{17}	C_{16}	C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0	HI/LO
SPECTRUM = 0	***A***	***B***	***C***	***D***	***E***	***F***	***G***	***H***	***I***	***J***	***K***	***L***	***M***	***N***	***O***	***P***	***Q***	***R***	***S***	***T***	***U***	***V***
	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	M_c	M_b	M_a
	M_e	M_d																				
SPECTRUM = 1	*T*	*U*	*R*	*S*	*P*	*Q*	*M*	*N*	*O*	*J*	*K*	*L*	*G*	*H*	*I*	*E*	*F*	*C*	*D*	*A*	*B*	*V*
	M_c	M_b	A_1	A_0	A_3	A_2	A_6	A_5	A_4	A_9	A_8	A_7	A_{12}	A_{11}	A_{10}	A_{14}	A_{13}	A_{16}	A_{15}	A_{18}	A_{17}	M_a
																				M_e	M_d	
										$M_2 = M_a$												
										$M_1 = M_b \wedge M_d$												
										$M_0 = M_c \wedge M_e$												

Fig. 11. Final INBUF transformation sequence.