A PIPELINED DISTRIBUTED ARITHMETIC PFFT PROCESSOR

P. Chow, Z.G. Vranesic, and J.L. Yen

Department of Electrical Engineering
University of Toronto
Toronto, Ontario, Canada

ABSTRACT

Previous experience in implementing the Prime Factor Fourier Transform showed that it was much more difficult to do than the FFT because of its complicated structure. In most FFT implementations the "butterfly" structure is the basic arithmetic unit implemented. It is much simpler than the equivalent PFFT unit.

This paper describes a different architecture for implementing PFFT machines using distributed arithmetic and ROM's to perform the computations. It is found to be much simpler and more modular than a design which uses multipliers and adders. The implementation of a PFFT processor with a throughput of 104 kHz for complex data points is described.

1. INTRODUCTION

One of the major factors limiting the computation speed of the Discrete Fourier Transform (DFT) has been the number of multiplications required. The work of Winograd, [1,2], has resulted in two new algorithms which have fewer multiplications than the Fast Fourier Transform, currently the most popular algorithm. These two algorithms are the Winograd Fourier Transform (WFT), and the Prime Factor Fourier Transform (PFFT), [3]. A previous paper, [4], described several possible architectures using microprocessors to implement the PFFT. The results showed that the reduction in the number of multiplications was gained at the loss of the simplicity of the structure of the FFT. Managing the data was much more complicated using the new algorithms.

In this paper, we will first explain briefly the structure of the PFFT and distributed arithmetic. Then we will describe an architecture using read only memories (ROM's) and the techniques of "distributed arithmetic" to implement the PFFT. Previous work by Liu and Peled, [5], has shown that using distributed arithmetic to do the multiplications in an FFT butterfly unit can result in significant savings in hardware and power consumption. We have implemented the PFFT using distributed arithmetic to do all of the computations in the PFFT units instead of just the multiplications. This results in a fairly modular system and a practical way of implementing pipelined PFFT processors. Finally, the performance of FASTOR II, a machine being built to compute a 504-point PFFT, will be evaluated, and some comparisons with FFT processors will be made.

This work was sponsored by the National Science and Engineering Research Council of Canada under research grants A3951, A5280.

2. THE PRIME FACTOR FOURIER TRANSFORM

The Prime Factor Fourier Transform is described in detail in [3], but we will give a short description here. The PFFT is based on the use of a set of relatively prime length short transforms to form the complete transform. For example, a 504-point transform consists of a seven-point, an eight-point and a nine-point transform where 7*8*9=504, (see Fig. 1). Each of the short transforms is implemented using the techniques of Winograd to minimize the number of multiplications and to derive the constants to be used for the multiplications. The ordering of the data for the short transforms is determined using the indices of the data points modulo the length of each of the short transforms. The result is that accessing the data is not completely straight forward. The PFFT is computed by first doing an input ordering and then doing the short transforms in turn. The order in which the short transforms are computed is only important when optimizing the memory required in the intermediate stages. It is possible to begin the next transform stage without computing all of the previous short transforms. There is also a stage at the end required to correctly order the data for output.

The PFFT has several properties which can be exploited when implementing the algorithm. It is more modular than the WFT because it computes the short transforms independently

CH1630-3/81/0000/0198$00.75 © 1981 IEEE

instead of nesting the transforms like the WFT. It is also possible to organize the data so that the constants used in the short transforms do not vary if the length of the full transform is changed. This means that it is possible to produce short transform modules and put together the modules needed to get the full transform required. The major change required to change from one length to another length would be the way the data is permuted. This can be done at the input and output stages.

One of the main difficulties of implementing a hardwired PFFT in the straight forward way using adders and multipliers is the complexity of the short transforms. The short transforms require a complicated routing algorithm for the data as the transform is being computed. If however, the short transforms are implemented using a table lookup, then the data routing in the short transform is not a problem. This is the basis of the structure for FASTOR II.

3. DISTRIBUTED ARITHMETIC

When using distributed arithmetic it is necessary to look at the arithmetic operations being performed at the bit level. At this level it is possible to mix the operations of convolution and multiplication giving rise to the name "distributed arithmetic". This leads to different ways of realizing digital filters and in this case, a different structure for implementing the PFFT. Distributed arithmetic and some of the structures suggested by its use is described in [6].

The basic principle of distributed arithmetic is to look at the N input B-bit data words one bit at a time, starting with the least significant bits (see Fig. 2). Use these bits as the operands for a linear function f and compute the output which will be a word of B bits. Then shift the input and output registers one bit and compute the next output word. Add this to the previous value in the output register. Continue this cycle for each bit in the input words. The final result will be in the output register.

The function f can be implemented using discrete adders and multipliers or more simply by using a lookup table to store the precalculated partial sums. This is the method that we used in our implementation of FASTOR II. The lookup tables are used to store precalculated values for the short transforms used in the PFFT.

4. THE INPUT AND OUTPUT STAGES

One of the problems with the PFFT mentioned previously is the ordering of the data required at the input and the output stages. In FASTOR II, the permutations are done with tables held in EPROM's which are used as address translators to find the location in memory where the data is to be stored on input or read on output (Fig. 3). The input and output stages run at relatively slow speeds compared to the transform stages because they only have to keep up with the maximum sample rate of about 9.6 μs per point in FASTOR II. Therefore, there is no problem with the address translation process.

With the proper ordering in memory the transform modules can access the data using counters. For different length transforms, it is necessary to reprogram the EPROM's and adjust the counters used by the transforms.

5. THE SHORT TRANSFORM MODULE

The basic structure of a transform module is shown in Fig. 4. Assume that a word is B bits long and the transform is N points long. The first step is "corner turning" which is used to arrange the input data words into slices of bits which can be used as inputs to the ROM lookup table. Each slice consists of one bit from each of the input data words. The next stage is a double buffer RAM. One side is used to store the bit slices after the corner turning while the bit slices in the other side are used to do the table lookups. The pipeline latch between the RAM and the ROM is used to hold the current inputs to the ROM while the next inputs are being accessed in the RAM. The word select counter selects the output point being computed. Its outputs also form part of the input to the lookup table. At the output of the ROM is an adder/subtracter used to accumulate the partial sums generated for each input. The output of the ROM is held in the pipeline latch to allow the next value to be accessed in the ROM while the value just read is used by the adder/subtracter. At the other input to the adder/subtracter there is a shifter and another latch. The shifter is used to shift the current sum when necessary and the latch is used to store the sum just computed.

The operation of corner turning required by FASTOR II is shown in Fig. 5. The data is naturally sampled or accumulated in words but the table lookup requires the ability to read slices of bits. FASTOR II implements corner turning using the method shown in Fig. 6. Fig. 6 shows a B:1 multiplexer and one half of the double buffer RAM. The figure shows 4K by 1 RAM's instead of much smaller ones because the outputs

from the previous transform stage are stored in the double buffer RAM, so it must be large enough to hold these points. In Fig. 1, all 8 output points of the first 8-point transform are stored in RAM 0 of the 9-point stage. The next 8 points are stored in RAM 1 and so on. Each of the RAM chips stores the data corresponding to one of the input points of the transform. For example, RAM 0 will contain data words which must be used as point 0 when computing the transform. The data is stored in the RAM by first enabling the chip corresponding to the correct point and then by using the counter to select all the bits of the data word for writing in the RAM. When all the data has been stored in the RAM's, then reading all the RAM's at once will give the bit slices required.

The data representation used by FASTOR II is composed of two 12-bit, two's complement integers, one for the real part and one for the imaginary part. In order to generate a real or an imaginary part of an output point, 24 partial sums must be accumulated. There are 12 partial sums due to the real inputs and 12 partial sums due to the imaginary inputs. The data is presented to the ROM in the order shown in Fig. 7, starting with the least significant bit of the real part and alternating with the imaginary part. This is done so that the output value can be accumulated all at once. This also means that the accumulated output should only be shifted every two cycles instead of every cycle. If all the real bits were done first and then all the imaginary bits, it would be necessary to have an extra register to store the result from the real part. This is then added to the result from the imaginary part after it is computed.

The word select counter is used to select the value being computed, for example, the imaginary part of point 3. This counter is fixed for the 24 cycles required to compute a word and then it is changed to select the next word.

The size of the ROM is determined by the number of points in the transform. If the short transform length is N and the output is complex, then there are 2N output words possible. There are N bits of address due to the data and $\lceil \log_2 2N \rceil$ bits from the word select counter to specify the output word being computed. As N gets large, the ROM becomes excessive and expensive. Large bipolar ROM's were found to be rare and quite costly at the time FASTOR II was being designed. The structure shown in Fig. 8 is a way of reducing the size of the ROM by adding extra adders. This method is discussed in [6]. The reason this can be done is because the computation of the transform is a linear function. In Fig. 8 the output point is computed by doing three table lookups and then combining the outputs to get the final result. For example, the output of ROM 0 is the value of the transform assuming the input points N/3 to N-1 are set to zero. Points 0 to N/3-1 are used as actual inputs to the ROM.

As an illustration of the difference in the size of ROM required, consider the case for N=9. In the first case the number of input bits is $N+\lceil \log_2 2N \rceil = 9+5 = 14$ bits. This means a ROM of 16K 12-bit words is required. Using the second method there are 3 sections of ROM with the number of input bits being $N/3+\lceil \log_2 2N \rceil = 3+5 = 8$ bits. Therefore, only three ROM's of 256 words are required. The ROM can be sectioned as many times as necessary with the tradeoff being a smaller ROM for extra adder delays.

In order to see how the tables in the ROM's are generated, let

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk}, \quad W_N = e^{j\frac{2\pi}{N}} \tag{1}$$

$$, \quad x_n = \sum_{b=0}^{B-1} x_{nb} 2^b \tag{2}$$

$(X_k)_{k=0,N-1}$ is the DFT of $(x_n)_{n=0,N-1}$ and (2) is the representation of x_n in bits where there are B bits in a word and x_n is real. Substituting,

$$X_k = \sum_{n=0}^{N-1} \sum_{b=0}^{B-1} x_{nb} 2^b W_N^{nk}$$

$$= \sum_{b=0}^{B-1} \sum_{n=0}^{N-1} x_{nb} W_N^{nk} 2^b \tag{3}$$

$$= \sum_{s=0}^{2} \sum_{b=0}^{B-1} \sum_{n=s\frac{N}{3}}^{(s+1)\frac{N}{3}-1} x_{nb} W_N^{nk} 2^b \tag{4}$$

The inner sum of (3) is used to generate the ROM with k and 1 other bit for the real or imaginary part forming the word select address and the x_{nb} forming the data input address bits. For complex x, let

$$x_n = x_n^r + jx_n^i$$

x_n^r transforms to a + jb

jx_n^i transforms to j(c+jd)

Then x_n transforms to (a-d)+j(b+c). Therefore, to compute the real part of an output point sub-

tract the imaginary parts due to the contributions from the imaginary input bits from the real parts due to the contributions from the real input bits. A similar procedure is used for computing the imaginary part of the output.

Equation (4) is used to generate the ROM tables for the configuration shown in Fig. 8, where s is the ROM number being generated. The example shown is for the specific case of dividing the ROM into three sections.

6. PERFORMANCE AND GENERALIZATIONS

FASTOR II is constructed with standard TTL technology without any attempt to push the limits of this technology. The sampling frequency of 20 kHz required in an initial specification for the machine was easily achieved. The cycle time used in the accumulator loop at the output of the lookup tables is 200 ns which is not very difficult to achieve in TTL. A complex output point composed of two 12-bit integers is generated every 48 cycles. This gives a possible sampling frequency of 1/(48*200 ns) = 104 kHz. This is greater than ten times the performance of FASTOR I, [4].

FASTOR I is a processor constructed with AM2903 bit slices, [7], which is microprogrammed to compute the short transforms of a 252-point PFFT. It is attached to an LSI-11, [8], which is used to perform the data manipulation required. The bottleneck in the system is the LSI-11 because of the time needed to do the data permutations. The system throughput is about 6 kHz for complex data.

In FASTOR II, the slowest part of the pipeline is in the ROM lookup stage where a level of addition at the output of the ROM sections has been used to reduce the amount of ROM required. Easy ways of relieving this bottleneck are to use larger ROM's to eliminate the adders or by having a second ROM section in parallel (Fig. 9).

The basic factor limiting the speed in this approach to implementing the PFFT is the inherent bit serial nature of the computations. As the number of bits used to represent the data increases, the number of ROM lookup and accumulator cycles increases in proportion. Therefore, this technique does not appear practical for very high speed signal processing applications, like radar, but it does seem promising for slower applications such as speech.

One of the problems with the WFT and the PFFT is that it is not practical to implement large transforms using adders and multipliers because as the lengths of the short transforms increase, the number of additions required becomes very large and the complexity of computing the transform increases. Therefore, short transforms greater than 16-points are impractical to implement in this way. However, using the techniques implemented in FASTOR II it is possible to do larger transforms because the difficulty of computing the short transforms does not increase when the number of input points increases. It is only the size of the lookup table which gets larger.

When comparing the implementation of FASTOR II to traditional FFT processors, the first major difference is that there is no hardware required to perform any multiplications. The main complexity of computing the transforms is hidden in the generation of the lookup tables. The arithmetic in the processor is limited to just additions and subtractions which only require relatively cheap and simple chips. This results in significant savings in power and the number of chips as shown by Liu and Peled in [5] for an FFT realization using a similar technique.

One advantage of using the PFFT is that fewer stages are required to do a transform compared to the number of stages in an FFT of similar length. For example, a 512-point radix 2 FFT requires 9 stages of butterflies. A 504-point PFFT requires only 3 stages, consisting of 7, 8 and 9-point transforms. Each PFFT stage in FASTOR II uses about 90 chips. It is estimated that a 16-point stage would require about 115 chips with about half of the increase being for memory and the rest being required to increase buffer and counter sizes. In the simplest implementation described in [5], about 42 chips were required to implement a computation unit with a 2.5 MHz complex data throughput. However, the arithmetic used in that system was a modified floating point using an 8-bit mantissa and a 4-bit exponent and the clock rate was 40 ns. Therefore, the comparison is not exact but the complexity of both systems is probably similar.

As mentioned before, an advantage of the PFFT is that it is possible to arrange the data so that no changes are required in the short transform algorithms when changing the length of the full transform. The only changes required to the configuration of a machine are the input and output permutations, and the size of the double buffer memory shown in Fig. 4. The arithmetic sections remain the same. In an FFT, changing the length of the transform also requires a change in the constants used in the arithmetic units as well as the changes required in the data ordering and the memory required.

A disadvantage of the PFFT is that all of the data is required before a transform can begin. This is unlike the FFT which can begin without a complete set of data.

The short transform modules are all basically the same in FASTOR II, except for the

size of the ROM's, the size of the RAM at the inputs and the programming of some of the counters. Therefore, it is possible to consider the design of a general PFFT chip which could handle short transforms up to some fixed size (16 points does not seem unreasonable). The internal structure should look like the one shown in Fig. 8, except that four sections of ROM could be used to reduce the total amount of ROM required. A small double buffer cache in front of the arithmetic section should be included to make pipelining easier to do. To compute a specific short transform, the lookup tables and some counters would have to be programmed. External circuitry would be required to do the input and output data permutations and the intermediate buffering. It would be interesting to consider other applications of such a chip, if it existed.

7. CONCLUSIONS

The work done in the implementation of FASTOR II has shown that the use of distributed arithmetic and ROM's is a practical way to implement the PFFT. By using ROM's to store precalculated values of the short transforms, the problem of computing the short transforms in the hardware is reduced to accumulating the partial sums. However, performing the computations in a bit serial manner requires the ability to do corner turning on the data.

The major factor limiting the throughput of the system is that the computations are at the bit serial level. However, this implementation does show that it is practical to build PFFT systems with throughputs in the 100 kHz range with little problem. There is also the option of using faster technologies.

The modularity of the transforms suggests that it is possible to produce a programmable chip which can be used to compute the short transforms. The fact that the difficulties in computing the short transforms are hidden once the ROM is programmed also means that it is possible to implement transforms larger than it was previously practical to do with the PFFT. Therefore, PFFT machines would require fewer stages than similar length FFT machines.

The result of this work has shown that the approach taken to implement FASTOR II is a practical way of implementing the PFFT. It has also shown that there are possibilities for using the PFFT as an alternative to the FFT for moderate sample rates.

8. ACKNOWLEDGEMENTS

We would like to thank Dan Grassick for his work in doing the detailed implementation of the hardware and the useful suggestions that he made.

REFERENCES

1. S. Winograd, "On Computing the Discrete Fourier Transform," Proc. Nat. Acad. Sci., U.S.A., Vol. 73, No. 4, April 1976, pp. 1005-1006.

2. S. Winograd, "On Computing the Discrete Fourier Transform," IBM T.J. Watson Res. Ctr., Yorktown Heights, N.Y., IBM Res. Rep., RC-6291, Nov. 1976.

3. D.P. Kolba and T.W. Parks, "A Prime Factor FFT Algorithm using High-speed Convolution," IEEE Trans. Acoust. Speech, Signal Processing, Vol. ASSP-25, August 1977, pp. 281-294.

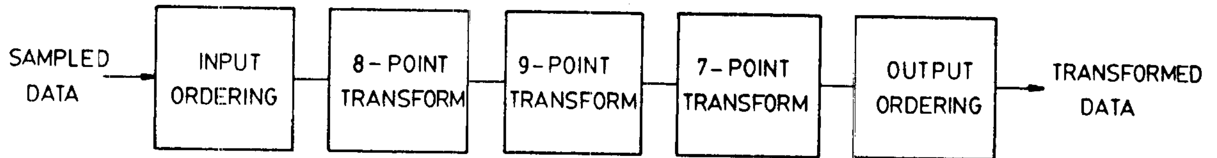
4. P. Chow, Z.G. Vranesic and J.L. Yen, "Microprocessor Implementations of Discrete Fourier Transform Machines," Compcon Fall, 1979, pp. 316-320.

5. B. Liu and A. Peled, "A New Hardware Realization of High-Speed Fast Fourier Transformers," IEEE Trans. Acoust. Speech, Signal Processing, Vol. ASSP-23, December 1975, pp. 543-547.

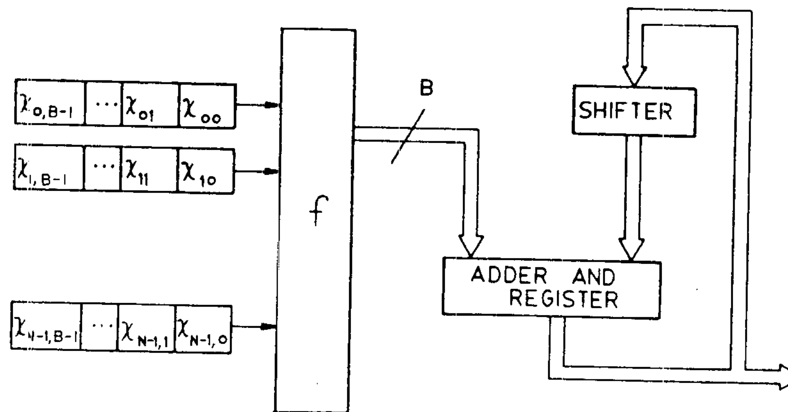
6. C.S. Burrus, "Digital Filter Structures Described by Distributed Arithmetic," IEEE Trans. Circuits and Systems, Vol. CAS-24, December 1977, pp. 674-680.

7. "The AM2900 Family Data Book," Advanced Micro Devices, Inc., Sunnyvale, California, 1979.

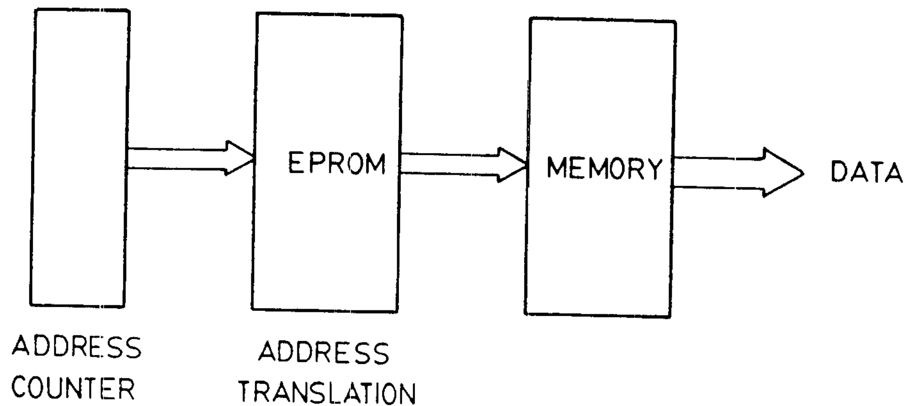
8. "Microcomputer Processor Handbook," Digital Equipment Corporation, Maynard, Mass., 1979.



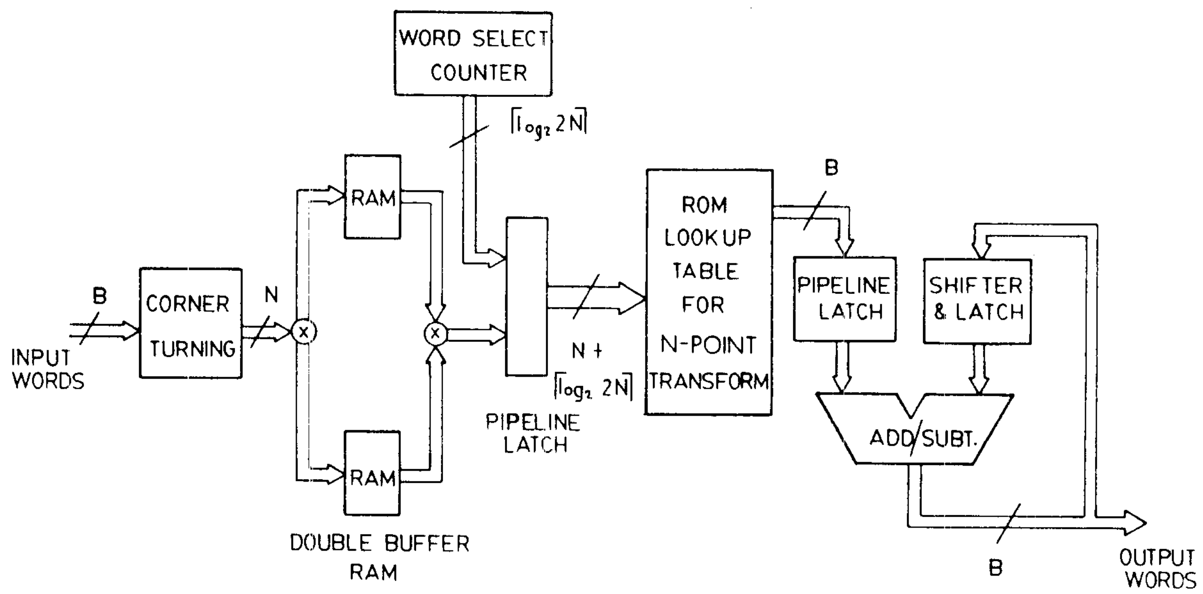
A 504 - POINT PFFT
Fig. 1



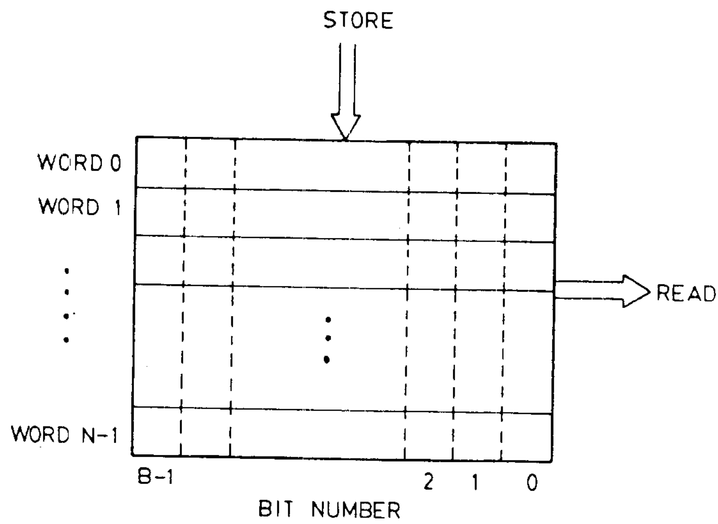
A Function Implemented with Distributed Arithmetic
Fig. 2



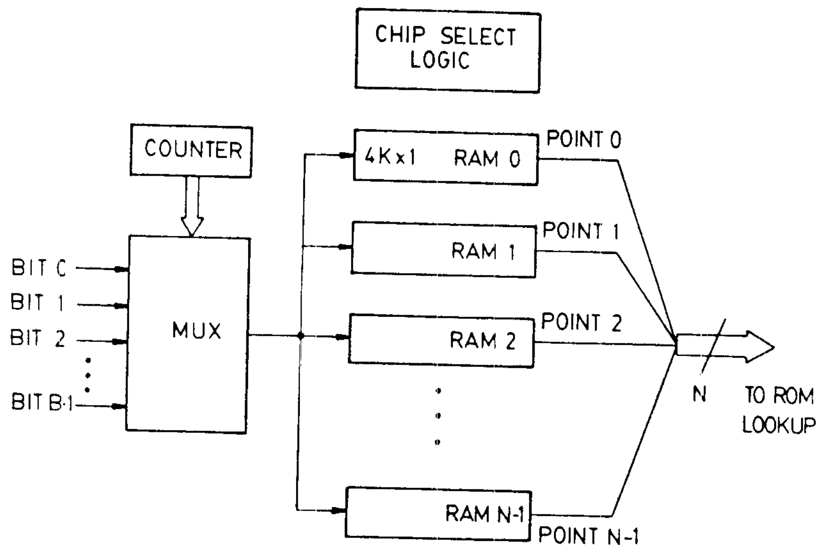
Address Translation
Fig. 3



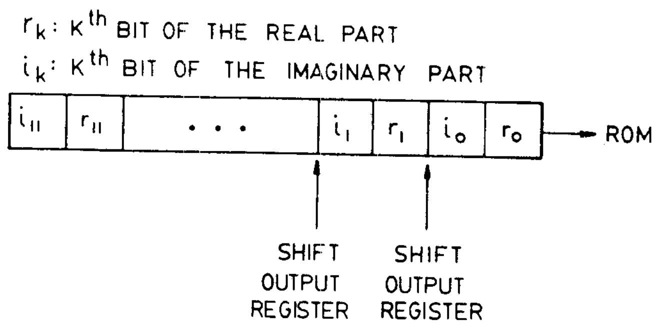
Basic Structure of Short Transform Module
Fig. 4



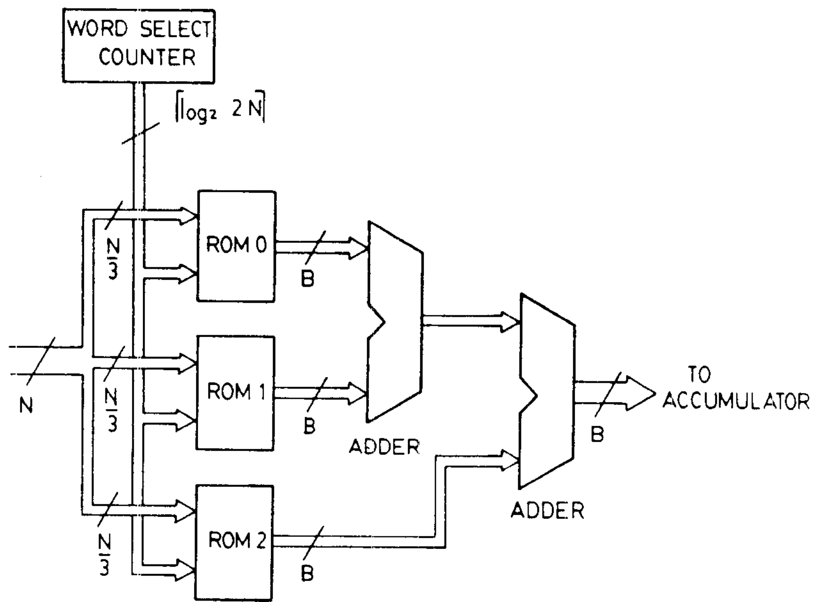
Corner Turning Operation Required
Fig.5



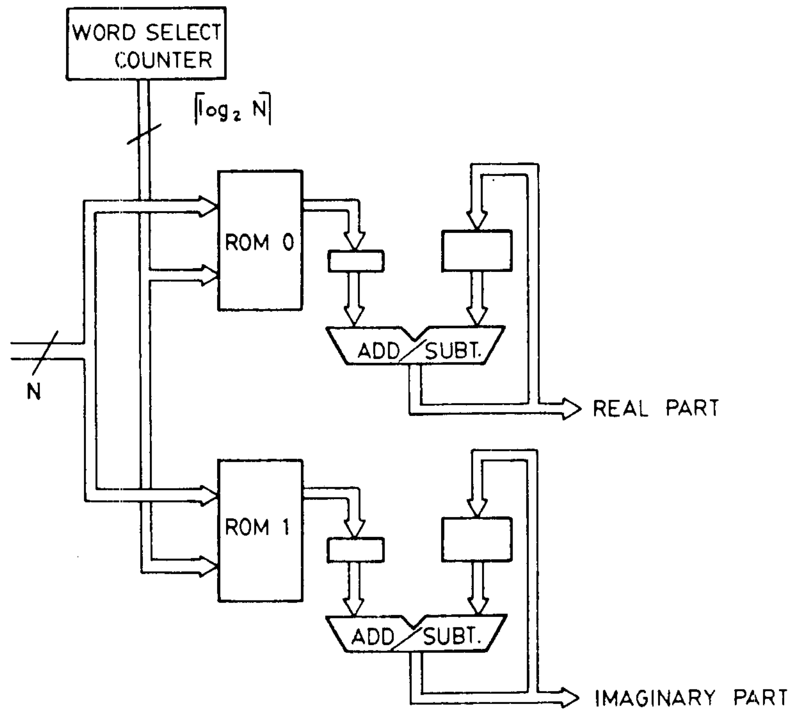
Corner Turning Used in FASTOR II
Fig. 6



Order of Data Presented to ROM
Fig. 7



Structure Using Smaller ROM's
Fig. 8



Using a Second Accumulator
to Achieve Higher Throughput
Fig. 9