Parallel Processing of the Fast Fourier Transform

Ke Zhou

Follow this and additional works at: https://openprairie.sdstate.edu/etd

Recommended Citation

https://openprairie.sdstate.edu/etd/4548

This Thesis - Open Access is brought to you for free and open access by Open PRAIRIE: Open Public Research Access Institutional Repository and Information Exchange. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of Open PRAIRIE: Open Public Research Access Institutional Repository and Information Exchange. For more information, please contact michael.biondo@sdstate.edu.
Parallel Processing of the Fast Fourier Transform

A thesis submitted
in partial fulfillment of the requirements for the
degree Master of Science
Major in Electrical Engineering
South Dakota State University
1988
Parallel Processing of the Fast Fourier Transform

ACKNOWLEDGMENTS

By KE ZHOU

This thesis is approved as a creditable and independent investigation by a candidate for the degree, Master of Science, and is acceptable for meeting the thesis requirements for this degree. Acceptance of this thesis does not imply that the conclusions reached by the candidate are necessarily the conclusions of the major department.

In addition, I would like to thank Professor C. G. Carson for his

Finally, students in the making the time on a graduate program an enjoyable one.

Dr. D.B. Miron  
Thesis Advisor

Dr. V. G. Ellerbruch  
Head, Electrical Engineering Department
ACKNOWLEDGMENTS

I would like to express my sincere appreciation to Dr. D. B. Miron, Associate Professor of Electrical Engineering for his excellent advice and constant encouragement. His assistance was invaluable in both my course work and this thesis project.

I would also like express my appreciation for all the encouragement Dr. V. G. Ellerbruch, Professor and Head of the Electrical Engineering Department gave me during my studies.

In addition, I would like to thank Professor C. G Carson for his help in my programming work.

Finally, I want to thank the staff and graduate students in the Electrical Engineering Department for making the time spent on my graduate program an enjoyable one.

Ke Zhou
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Item</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Chapter One: Properties of the Discrete Fourier Transform</td>
<td>6</td>
</tr>
<tr>
<td>1.1 Properties of the DFT</td>
<td>6</td>
</tr>
<tr>
<td>1.1.1 The DFT and IDFT</td>
<td>7</td>
</tr>
<tr>
<td>1.2 Complex Input Numbers in A Program</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Floating-Point Input Numbers in A Program</td>
<td>11</td>
</tr>
<tr>
<td>1.4 Program of the DFT</td>
<td>13</td>
</tr>
<tr>
<td>Chapter Two: Fast Fourier Transform</td>
<td>24</td>
</tr>
<tr>
<td>2.1 The Cooley-Tukey Algorithm</td>
<td>26</td>
</tr>
<tr>
<td>2.1.1 Multidimensional Index Mapping</td>
<td>27</td>
</tr>
<tr>
<td>2.1.2 The Cooley-Tukey FFT Algorithm</td>
<td>29</td>
</tr>
<tr>
<td>2.2 The Winograd FFT Algorithm</td>
<td>37</td>
</tr>
<tr>
<td>2.3 A Fortran Program for the Cooley-Tukey FFT</td>
<td>39</td>
</tr>
<tr>
<td>Chapter Three: Vector Processor and Parallel Processing</td>
<td>43</td>
</tr>
</tbody>
</table>
Chapter Three: The Vector Controller

3.1 The Vector Controller

3.1.1 Scalar and Vector Modes

3.2 Vectorization of A Scalar Program

3.2.1 Program Dependence

3.2.2 Data Dependence

3.2.3 Control Dependence

3.2.4 Dependence Level

3.2.5 Dependence Interchange Problem

3.3 The Vectorization Analysis of the FFT

Chapter Four: Scalar and Vectorized Cooley-Tukey FFT Program

4.1.2 Incorporating the VC Instruction

4.2 The FFT Program Executed on IBM-PC

4.3 The Vectorized FFT Program

4.4 The VP Performance Estimation

4.4.1 Instruction Timing

4.4.2 Timing for the FFT Program on IBM-PC

4.4.3 Timing for the Vectorized FFT Program

4.5 Conclusion
Figures

Fig. 1.1 Fortran Program for the DFT with Table Look-Up........................................... 15

Fig. 1.2 Assembly Program for the DFT................................................................. 16

Fig. 1.3 Assembly Sin and Cos Subroutine......................................................... 20

Fig. 2.1 8-Point, Radix-2, In-Place FFT............................................................ 41

Fig. 2.2 Fortran Program for a Cooley-Tukey Radix-2 FFT........................................ 42

Fig. 3.1 The Architecture of Micro-Vector System........................................... 66

Fig. 3.2 Generic Architecture of a Shared Memory System...................................... 67

Fig. 3.3 Decomposed DO30 Loop................................................................. 68

Fig. 4.1 An Improved Fortran FFT Subroutine................................................. 86

Fig. 4.2 The FFT Program on IBM-PC................................................................. 88

Fig. 4.3 The Vectorized FFT Program.............................................................. 101

Fig. 4.4 BASIC IFFT Calling Program
The algorithms which evaluate Discrete Fourier Transforms (DFT) and Inverse DFT very efficiently are called Fast Fourier Transforms (FFT). The FFT has become a basic tool in various scientific and engineering disciplines, ranging from oil exploration to artificial intelligence. Efficient FFT computation is a requirement of any engineering/scientific computer.

The first FFT algorithms were reported by Huns and Oppen in 1965, and by Goodman and Heide in 1966. However, the FFT didn't receive much attention at all until Cooley and Tukey published their algorithm [6] in 1965. The Cooley-Tukey algorithm is simple and widely used in many application software packages. Winograd developed his FFT in 1978, which is based upon the prime factor theory.
Parallel Processing of the Fast Fourier Transform

INTRODUCTION

The algorithms which evaluate Discrete Fourier Transforms (DFT) and Inverse DFT very efficiently are called Fast Fourier Transforms (FFT). The FFT has become a basic tool in various scientific and engineering disciplines, ranging from oil exploration to artificial intelligence. Efficient FFT computation is a requirement of any engineering/scientific computer.

The first FFT algorithms were reported by Runge and Konig in 1924, and by Danielson and Lanczos in 1942. However, the FFT didn't receive much attention at all until Cooley and Tukey published their algorithm [1] in 1965. The Cooley-Tukey algorithm is simple and widely used in many application software packages. Winograd developed his FFT in 1976, which is based upon the prime factor theory
(see [7]). It is typically faster than the Cooley-Tukey Algorithm, if the computer system has no multiplication instructions. According to the book prepared by the Digital Signal Processing Committee of the IEEE in 1979, the speed difference among these FFT algorithms is around 40%.

My objective in this paper is to choose a proper algorithm, establish the appropriate programming techniques, and determine the sequence of steps required to implement a FFT both on a conventional IBM-PC and a Vector Processor (VP) system. I will demonstrate how to vectorize a FFT so that the algorithm can be performed under a VP system. The analysis of data dependence in an algorithm is another important part of this paper.

The paper includes the analysis of the Cooley-Tukey and Winograd FFT algorithms. The Prime factor method will be used in these two FFTs. It will be seen that the Cooley-Tukey Algorithm can be more easily implemented
on a vector system and needs fewer memory locations. The details of the Winograd FFT algorithm can be found in [2].

In addition, this paper has two Cooley-Tukey FFTs and one DFT program written in Assembly Language. One of two FFT programs has been tested and executed on a conventional IBM-PC which has an Intel-8088 processor as the Central Processing Unit (CPU), and one Intel-8087 Numeric Data Processor. The 8087 is specially designed to perform real number operations efficiently and quickly. Because of the special architecture of the 8087, single or double precision can be easily processed. The tested program was compiled and linked by Microsoft Assembly Language version 5.0 and the required results of both the FFT and Inverse FFT were obtained.

Another program of the Cooley-Tukey FFT is a modified version of the first one. The difference between these two programs is the inclusion of the vector
instructions for the VP. A VP system consists of multiple 8087s which can be activated sequentially or in parallel. Most of the mathematical calculations are performed using 8087s by loading the data elements of an array sequentially into them and executing the instructions in parallel. The results of the operations are sequentially retrieved from the 8087s and stored at the addresses pointed to by the CPU. A detailed discussion of the VP and its Instructions are given in Chapter Four. Since the required hardware using more than one 8087 is not currently working, the vectorized FFT program was only compiled without execution. Since the first program has the same algorithm and more than 70% of the same program code as the vectorized one, it could be executed using a working Vector Processor without any difficulty.

There is a data format of conversion problem between the CPU and I/O ports, the conversion uses a long Assembly program to realize the Input and Output of floating-point numbers. This program is even longer than
the FFT program. Thus, Microsoft Quick BASIC was used to call these Assembly program subroutines and to handle the Input and output of real numbers. Actually, other high level languages such as Microsoft C and FORTRAN could also be used without many modifications.

The VP performance will be finally mentioned in Chapter Four. The number of clock periods required to execute a particular form (register-to-register, immediate-to-memory, etc.) of instruction is counted and discussed. It will be summarized that the key point to vectorize a program is to eliminate the use of serial mode program segments and the parallel mode is very helpful in raising the execution speed of a program.

FORTRAN is a high-level language which was used to describe the FFT algorithms. All these algorithms have been tested and satisfactory results were obtained.
Chapter One

Properties of Discrete Fourier Transform

This Chapter describes the properties of the Discrete Fourier Transform (DFT), the mathematical preparation for later programming and the improved direct calculation of the DFT. At the end of this chapter, we will introduce a FORTRAN subroutine and a subroutine written in Microsoft Assembly Language. These two programs were tested both on an IBM 4381 main frame and an IBM-PC. The required results were obtained.

1.1 Properties of the DFT

The DFT is one of the most fundamental operations in digital signal processing. The DFT is used in the description, representation, and analysis of discrete-time signals. Efficient DFT algorithms make considerable use of digital signal processing practical.
1.1.1 DFT and IDFT

The DFT of a sequence of $N$ complex numbers \( \{x_0, x_1, x_2, \ldots, x_{N-1}\} \) is another sequence of complex numbers \( \{X_0, X_1, X_2, \ldots, X_{N-1}\} \). The relation between \( x_n \) and \( X_k \) is:

\[
X_k = \sum_{n=0}^{N-1} x_n \cdot \exp(-j2\pi nk/N), \quad k=0,1,2,\ldots,N-1
\]

\[\tag{1.1}\]

In most cases, the signal \( x(n) \) comes from samples of some continuous time signal; therefore, the index \( n \) is considered to be a sample of time.

The Inverse Discrete Fourier Transform (IDFT) of the sequence \( \{X_0, X_1, X_2, \ldots, X_{N-1}\} \) gives the original sequence \( \{x_0, x_1, x_2, \ldots, x_{N-1}\} \) by the next formula:

\[
x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot \exp(j2\pi nk/N), \quad n=0,1,2,\ldots,N-1
\]

\[\tag{1.2}\]
to simplify the notation, use \( W = \exp(-j2\pi/N) \). The definitions of the DFT and IDFT become:

\[
X_k = \sum_{n=0}^{N-1} x_n W^{nk}, \quad k=0,1,2,...,N-1
\]

\[\text{(1.3)}\]

\[
x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k W^{-nk}, \quad n=0,1,2,...,N-1
\]

\[\text{(1.4)}\]

1.2 Complex Input Numbers in A Program

In FORTRAN, complex variables can be defined on the top of a program and the compiler is able to handle the complex calculations. But it is impossible to define a complex number in Assembly Language. In other words, complex numbers can not be directly used in the Assembly program. Thus, the following mathematical method is necessary.
In the case where \( x(n) \) is complex, the real and imaginary parts of \( x(n) \) are denoted by \( X(n) \) and \( Y(n) \), respectively, and the real and imaginary parts of the transform by \( A(k) \) and \( B(k) \),

\[
x(n) = X(n) + jY(n) \]

\[
X(k) = A(k) + jB(k) \]

with \( X, Y, A \) and \( B \) being real-valued functions of variables.

I substitute these two definitions and \( \omega_n^k = \cos(Qnk) - j\sin(Qnk) \), \( Q = 2\pi/N \) into (1.3) and IDFT(1.4),

The DFT becomes:

\[
A(k) + jB(k) = \sum_{n=0}^{N-1} \left[ X(n) \cos(Qnk) + Y(n) \sin(Qnk) \right]
\]

or

\[
A(k) = \sum_{n=0}^{N-1} \left[ X(n) \cos(Qnk) + Y(n) \sin(Qnk) \right]
\]
\[ B(k) = \sum_{n=0}^{N-1} \left[ Y(n) \cos(Qnk) - X(n) \sin(Qnk) \right] \]

Similarly, the IDFT becomes

\[ X(n) + jY(n) = \frac{1}{N} \sum_{n=0}^{N-1} \left[ A(k) + jB(k) \right] \left[ \cos(Qnk) + j\sin(Qnk) \right] \]

Thus,

\[ X(n) = \frac{1}{N} \sum_{n=0}^{N-1} \left[ (A(k) \cos(Qnk) - B(k) \sin(Qnk)) \right] \]

\[ Y(n) = \frac{1}{N} \sum_{n=0}^{N-1} \left[ -(B(k) \cos(Qnk) - A(k) \sin(Qnk)) \right] \]

The only difference in the DFT and IDFT is the factor of \((1/N)\) and negative parts in the IDFT. In other words, if an algorithm exists for calculation of the DFT, multiplication of the DFT by \(1/N\) and reversing the signs of input and output imaginary parts give an algorithm for
When I developed the programs, I didn’t particularly define any complex numbers in the programs. In all the later programs of this paper, real and imaginary parts are processed independently in two separate real arrays. This is very effective, particularly when I developed the DFT and FFT in Assembly language.

1.3 Floating-Point Input Numbers in A Program

In many practical applications, the data to be processed may be floating-point numbers. The next procedure could be followed so that the general FFT algorithm may be used.

1. Function \( x(k) \) is floating-point \( k=0,1,\ldots,2N-1 \)

2. Divide \( x(k) \) into two functions

\[
h(k) = x(2k)
\]
g(k)=x(2k+1) \quad k=0,1,\ldots,N-1

3. Form the complex function

\[ y(k)=h(k)+jg(k) \quad k=0,1,\ldots,N-1 \]

4. Compute

\[ Y(n)=\sum_{k=0}^{N-1} y(k)\exp(-j2\pi nk/N) \]

\[ =R(n)+jI(n) \quad n=0,1,\ldots,N-1 \]

where \( R(n) \) and \( I(n) \) are the real and imaginary parts of \( Y(n) \), respectively.

5. Compute

\[ X_r(n)=[R(n)/2 + R(N-n)/2] + \]

\[ \cos\left(\frac{n}{N}\right)[I(n)+I(N-n)/2]-\]

\[ \sin\left(\frac{n}{N}\right)[R(n)/2-R(N-n)/2]) \]

\[ n = 0,1,\ldots,N-1 \]

\[ X_i(n)=[I(n)/2-I(N-n)/2] - \]

\[ \sin\left(\frac{n}{N}\right)[I(n)/2+I(N-n)/2]-\]

\[ \cos\left(\frac{n}{N}\right)[R(n)/2-R(N-n)/2] \]

\[ n=0,1,\ldots,N-1 \]

where \( X_r(n) \) and \( X_i(n) \) are respectively the real and imaginary parts of the 2N points discrete transform of
1.4 Program of the DFT

An obvious way to calculate the DFT of a signal \( x(n) \) is to implement the definition of the DFT directly. Fig. 1.1 is a FORTRAN program composed of two parts - a calling program and DFT subroutine. Four arrays were defined in the program, two for input and two for output. DO10 loop is an initializing process to set up SIN and COS tables. It is not possible to save the output data back to input arrays with this approach. This algorithm uses more memory locations. The inner most loop performs DFT calculations. It needs four multiplications every time it goes through the loop.

Fig. 1.2 is a DFT program written in Assembly language. It can be used under Microsoft Quick BASIC. It has the same algorithm as Fig. 1.1. Fig. 1.3 is a subroutine which will be called by the program in Fig. 1.2.
to find the sin and cos values with the same angle. The 8087 and 8088 instructions used in Fig. 1.2 will be discussed in Chapter Four.
FIG. 1.1

FORTRAN PROGRAM FOR THE DFT WITH TABLE LOOK UP

DIMENSION X(128), Y(128), A(128), B(128)
N=0
PRINT,' DATA BEFORE DFT'
20 READ(5,*,END=80)X(I),Y(I)
N=N+1
GOTO 20
80 CALL DFT(X,Y,A,B,N)
PRINT,' DATA AFTER DFT'
DO 30 I=1,N
PRINT,A(I),B(I)
30 CONTINUE
STOP
END

SUBROUTINE DFT(X,Y,A,B,N)

PI: 3.1415926
Q=2*PI/N
DO 10 J=1,N
C(J)=COS(Q*(J-1))
S(J)=SIN(Q*(J-1))
10 DO 20 J=1,N
AT=X(1)
BT=Y(1)
K=1
DO 30 I=2,N
K=K+J-1
IF (K.GT.N) K=K-N
AT=AT+C(K)*K(I)+S(K)*Y(I)
30 BT=BT+C(K)*Y(I)-S(K)*X(I)
A(J)=AT
B(J)=BT
RETURN
END
ASSEMBLY PROGRAM FOR THE DFT

PUBLIC DFT

DFT PROC FAR
EXTRN GETSC:NEAR
JMP START

ARRAYX DD 60 DUP(?) ;KEEP 60*4 BYTES FOR ARRAY X
ARRAYY DD 60 DUP(?) ;KEEP 60*4 BYTES FOR ARRAY Y
ARRAYA DD 60 DUP(?) ;KEEP 60*4 BYTES FOR OUTPUT ARRAY A
ARRAYB DD 60 DUP(?) ;KEEP 60*4 BYTES FOR OUTPUT ARRAY B
ARRAYC DD 60 DUP(?) ;KEEP 60*4 BYTES FOR COS TABLE
ARRAYS DD 60 DUP(?) ;KEEP 60*4 BYTES FOR SIN TABLE
N DW 16 ;SAVE THE NO. OF ELE. HERE
PItwo DD 6.283186 ;SAVE 2*PI HERE FOR LATE CALCULATION
Q DD 0.3927 ;Q=PI2/N
N1 DD ? ;SAVE REAL N HERE
A DD ? ;SAVE ANGEL HERE
TEMP DW ? ;SAVE TEMPORARY RESULTS
TEMP1 DD ? ;KEEP CONSTANT FOUR HERE FOR LATER CALC.
AT DD ?
BT DD ?
K DW ?
J DW ?
J1 DW ?

START:
PUSH BP ;SAVE ORIGINAL BP
MOV BP,SP ;SAVE CURRENT SP INTO BP
MOV BX,[BP]+14;GET ADDR(N)
MOV AX,[BX] ;SAVE N INTO AX
MOV N,AX ;SAVE N INTO N LOCATION
FNINIT ;INITIALIZE REGISTERS OF 8087
MOV CX,N ;SAVE N INTO CX
MOV BX,[BP]+12; GET ADDR(ARRAYX)
MOV SI, 4
I1:
FLD DWORD PTR [BX]
FSTP DWORD PTR ARRAYX [SI]
ADD SI, 4 ; GET OFFSET FOR NEXT ELE.
ADD BX, 4
DEC CX
JNZ I1
MOV CX, N ; GET COUNTER
MOV BX, [BP] + 10; GET ADDR(ARRAYY)
MOV SI, 4
I2:
FLD DWORD PTR [BX]
FSTP DWORD PTR ARRAYY [SI]
ADD SI, 4
ADD BX, 4
DEC CX
JNZ I2
FLD PI TWO ; GET 2*PI ON TOS
FLD N ; GET THE VALUE OF N HERE
FSTP DWORD PTR N1 ; POP INTO N1 AS A REAL
FDIV N1 ; FIND 2*PI/N IN TOS
FSTP Q ; POP IT INTO Q LOCATION
FLDZ
FSTP A ; SAVE 0 INTO A
FWAIT ; MAKE SURE 8087 HAS DONE
MOV CX, AX ; MOV N INTO CX FOR COUNTING
MOV SI, 4
D010:
CALL GETSC
ADD SI, 4
FLD Q
FLD A
FADD ST, ST(1)
FSTP A
DEC CX
JNZ D010 ; NEXT LOOP
MOV J, 1 ; MOVE 1 TO J
MOV J1, 4
MOV AX, N ; GET N
PUSH AX ; SAVE N ONTO STACK
D020:
MOV SI, 4 ; GET OFFSET OF ARRAY
FLD DWORD PTR ARRAYX [SI] ; GET X(1)
FSTP DWORD PTR AT ; SAVE X(1) INTO AT
FLD DWORD PTR ARRAYY [SI] ; GET Y(1)
FSTP DWORD PTR BT ; SAVE IT INTO BT
MOV K, 1 ; SAVE 1 INTO K

; STARTING OF DO30
MOV SI, 8 ; SAVE 2 INTO I
MOV AX, N
PUSH AX ; SAVE N FOR LATER COUNTING
DO30:
   MOV AX, K ; GET K
   MOV BX, J
   ADD AX, BX
   SUB AX, 1 ; PERFORM K+J-1
   MOV K, AX ; SAVE IT INTO K
   MOV BX, N ; GET N IN BX
   CMP AX, BX ; K.GT.N ?
   JLE B1
   MOV AX, K ; GET K IN AX
   MOV BX, N ; GET N IN BX
   SUB AX, BX ; K = K-N
B1:
   MOV K, AX ; SAVE NEW K INTO K
   IMUL FOUR
   MOV BX, AX
   FNINIT ; RESET 8087
   FLD DWORD PTR ARRAYC [BX]
   FLD DWORD PTR ARRAYX [SI]
   FMUL ST,ST(1) ; GET C(K)*X(I) ON TOS
   FLD DWORD PTR ARRAYS [BX]
   FLD DWORD PTR ARRAYY [SI]
   FMUL ST,ST(1)
   FADD ST,ST(2)
   FLD DWORD PTR AT
   FADD ST,ST(1) ; GET NEW AT
   FSTP DWORD PTR AT ; SAVE IT INTO AT NOW
   FNINIT ; RESET 8087 COPROCESSOR
   FLD DWORD PTR ARRAYC [BX] ; GET C(K)
   FLD DWORD PTR ARRAYY [SI] ; GET Y(I)
   FMUL ST,ST(1)
   FLD DWORD PTR ARRAYS [BX] ; GET S(K)
   FLD DWORD PTR ARRAYX [SI] ; GET X(I)
   FMUL ST,ST(1) ; GET S(K)*X(I)
   FSUB ST(2),ST
   FLD DWORD PTR BT
   FADD ST,ST(3)
   FSTP DWORD PTR BT ; POP TO BT
POP CX
DEC CX ; DONE THE LOOP DO30?
JNZ PP ; GO TO PP IF NO
MOV AX, J1
MOV DI, AX
FLD DWORD PTR AT ;GET AT
FSTP DWORD PTR ARRAYA [DI]
FLD DWORD PTR BT ;GET BT
FSTP DWORD PTR ARRAYB [DI]
POP CX
DB R CX ;DONE DO20 ?
JNZ PP1 ;NO, GO TO PP1
JMP OUTPUT
PP:
PUSH CX ;SAVE CHANGED N BACK STACK
ADD SI, 4 ;INCREMENT SI
JMP DO30 ;GO BACK DO30 LOOP
PP1:
PUSH CX ;SAVE NEW COUNTER BACK TO STACK
MOV AX, J
ADD AX, 1
MOV J, AX
MOV AX, J1
ADD AX, 4
MOV J1, AX
JMP DO20 ;GO BACK TO DO20
OUTPUT:
MOV SI, 4 ;GET THE STARTING ADDR. OF ARRAY
MOV BX, [BP] + 6 ;GET ADDR(ARRAYB)
MOV DI, [BP] + 8 ;GET ADDR(ARRAYA)
MOV CX, N ;GET THE COUNTER
AGAIN:
FLD DWORD PTR ARRAYB [SI]
FSTP DWORD PTR [BX]
FLD DWORD PTR ARRAYA [SI] ;OUTPUT ARRAY A
FSTP DWORD PTR [DI]
ADD SI, 4
ADD DI, 4
ADD BX, 4
DEC CX ;DONE ALL ELE.
JNZ AGAIN ;NO, GO TO AGAIN
POP BP
RET 10
DFT ENDP
END
Fig. 1.3

ASSEMBLY SIN AND COS SUBROUTINE

.DECLARATION

PUBLIC GETSC

.PROC

JMP START

LOOP1:

MOV SIGN_STORE,0 ;ASSUME POSITIVE
FTST ;TEST STACK TOP
FSTSW STATUS_WORD ;GET STATUS WORD
FWAIT

MOV AH,BYTE PTR STATUS_WORD+1
SAHF ;SET STATUS BIT
JNC NON_NEGATIVE ;JMP IF CF =0
MOV SIGN_STORE,-1;ITS NEGITIVE
FABS ;CHANGE TO POSITIVE,

NON_NEGATIVE:

MOV REALLY_COS,0 ;SIN ,NOT COS
FILD MINUS2 ;LOAD MINUS INTEGER
FLDPI ;LOAD PI
FSSCALE
FSTP ST(1) ;DUMP -2
FXCH

RANGE:

FPREM ;FIND PARTIAL REMAINDER
FSTSW STATUS_WORD+D
FWAIT

MOV AH,BYTE PTR STATUS_WORD+1

PUBLIC GETSC

PROC NEAR

JMP START

LOOP1:

MOV SIGN_STORE,0 ;ASSUME POSITIVE
FTST ;TEST STACK TOP
FSTSW STATUS_WORD ;GET STATUS WORD
FWAIT

MOV AH,BYTE PTR STATUS_WORD+1
SAHF ;SET STATUS BIT
JNC NON_NEGATIVE ;JMP IF CF =0
MOV SIGN_STORE,-1;ITS NEGITIVE
FABS ;CHANGE TO POSITIVE,

NON_NEGATIVE:

MOV REALLY_COS,0 ;SIN ,NOT COS
FILD MINUS2 ;LOAD MINUS INTEGER
FLDPI ;LOAD PI
FSSCALE
FSTP ST(1) ;DUMP -2
FXCH

RANGE:

FPREM ;FIND PARTIAL REMAINDER
FSTSW STATUS_WORD+D
FWAIT

MOV AH,BYTE PTR STATUS_WORD+1
FSTP ST ; DUMP ST
FLDZ       ; READ RATIO 0 TO 1
FLD1

SINDONE:

MOV BX, 0 ; ASSUME C3 OFF
TEST AH, 01000000B
JZ NOC3   ; JUMP IF OFF
MOV BX, 1 ; NOTE C3 ON

NOC3:

TEST AH, 10B
JZ NOC1   ; JUMP IF OFF
XOR BX, 1
JMP DOSINB

NOC1:

XOR BX, 0

DOSINB:

CMP BX, 1
JNE SINFUNC
FXCH

SINFUNC:

FMUL ST(0), ST(0) ; ST(0) = Y*Y
FLD ST(1)         ; ST(0) = X
FMUL ST(0), ST(0) ; ST(0) = X*X
FADDP ST(1), ST(0) ; ST(0) = X*X + Y*Y
FSQRT
FDIVP ST(1), ST(0)
TEST AH, 1B
JZ COOFF
NOT SIGN_STORE

COOFF:

; DO WE NEED CHANGE SIGN?
CMP SIGN_STORE, 0
JE POS
FXCHS

POS:

CMP SIGNAL, 1
JZ SAVESIN
FST QWORD PTR C
JMP ALLDONE

SAVESIN:

FST QWORD PTR S
SAHF
JP RANGE ; THIS TESTS BIT C2
CMP REALLY_COS,0
JE ITS_SINE
XOR AH,01000000B
TEST AH,01000000B
JNZ NOCARRY
XOR AH,1B

NOCARRY:
ITS_SINE:

FTST
FSTSW STATUS_WORD
FWAIT
MOV BX,0
AND BYTE PTR STATUS_WORD+1,01000001B
CMP BYTE PTR STATUS_WORD+1,01000001B
JNE NOT_ZERO
MOV BX,-1

NOT_ZERO:
TEST AH,10B ; IS C1 ON
JZ C1ISOFF ; JUMP IF OFF
CMP BX,0 ; ST EXACTLY ZERO
JNE STOANDC1 ; JUMP IF YES
FSUBP ST(1),ST ; NOW PI/4-ST
FPTAN
JMP SINDONE

STOANDC1:
FSTP ST ; POP ST
FSTP ST ; AND PI/4
FLD1 ; LOAD RATIO 1 TO 1
FLD1
JMP SINDONE

C1ISOFF:
FSTP ST(1) ; GET RID OF PI/4
CMP BX,0 ; ST EXACTLY ZERO ?
JNE STOANDNOC1 ; JUMP IF YES
FPTAN
JMP SINDONE

STOANDNOC1:
; FIND COSINE

```
FLD PI2 ; GET PI/2
FLD A ; GET THE ANGEL
FSUBP ST(1),ST ; GET PI/2 - A ON TOS
MOV SIGNAL,0
JMP LOOP1
POP BX
POP AX
RET
ENDP
END
```

The direct computation of a single value or element of the DFT of a sequence \( x[n] \) requires \( 4N \) multiplications and \((4N-2)\) real additions. Since \( N \) must be computed for \( N \) different values of \( N \), the direct computation of the DFT of a sequence \( x[n] \) requires \( 4N^2 \) real multiplications and \( N(4N-2) \) real additions or, alternatively, \( N^2 \) complex multiplications and additions. The implementation of the computation of the DFT on a general-purpose digital computer or a system with special hardware requires provision for storing and accessing the input sequence values \( x[n] \) and the values of the coefficients \( \frac{1}{\sqrt{N}} \). Since the amount of accessing and storing of data in numerical computation algorithms is generally proportional to the number of arithmetic operations, it is generally accepted that a meaningful measure of complexity, or, of the time
Chapter Two

Fast Fourier Transforms

From the discussion of the DFT of Chapter one, the direct computation of a single value or element of $X(K)$ requires $4N$ real multiplications and $(4N-2)$ real additions. Since $X(K)$ must be computed for $N$ different values of $K$, the direct computation of the DFT of a sequence $x(n)$ requires $4N^2$ real multiplications and $N(4N-2)$ real additions or, alternatively, $N^2$ complex multiplications and additions. The implementation of the computation of the DFT on a general-purpose digital computer or a system with special hardware requires provision for storing and accessing the input sequence values $x(n)$ and the values of the coefficients $W_k^n$. Since the amount of accessing and storing of data in numerical computation algorithms is generally proportional to the number of arithmetic operations, it is generally accepted that a meaningful measure of complexity, or, of the time
required to implement a computational algorithm, is the number of multiplications and additions required. Thus, for the direct calculation of the DFT, a convenient measure of the efficiency of the computation is the fact that $4N^2$ real multiplications and $N(4N-2)$ real additions are required. Since the amount of computation, and thus the computation time, is approximately proportional to $N^2$, it is evident that the number of arithmetic operations required to compute the DFT by the direct method becomes very large for large values of $N$. For this reason, computational procedures, that reduce the number of multiplications and additions are of considerable interest.

Cooley and Tukey published their algorithm for the computation of the DFT that is applicable when $N$ is composite number; i.e., $N$ is the product of two or more integers. The principle of the algorithm is most conveniently illustrated by considering the special case of $N$ an integer power of two; i.e., $N=2^M$. This algorithm
resulted in the discovery of a number of computational algorithms which have come to be known as FFTs. In 1975, S. Winograd published his theory for efficient calculation of Prime-Length cyclic convolution using a minimum number of multiplications. The most important part of the theory is Multidimensional Index Mapping. The Cooley-Tukey algorithm can also be developed by this theory instead of obtaining the algorithm from the matrix calculation. Winograd’s FFT will be also introduced in this chapter. The key to these methods lies in their exploitation of the possibilities for factoring the number of values of the series to be transformed. They decompose the computation of the DFT of a sequence of length $N$ into successively smaller DFTs. The manner in which this principle is implemented leads to a number of different algorithms, all with comparable improvements in computational speed.

2.1 The Cooley-Tukey Algorithm
2.1.1 Multidimensional Index Mapping

Index mapping is one of the most practical methods of reducing the arithmetic necessary to calculate the DFT. The basic idea of this mapping is to decompose a one dimensional problem into a multi-dimensional one.

Apparently, if the length of data sequence is not prime, $N$ can be factored as $N = N_1 N_2$ with the time index $n$ of (1.3) taking on values of

$$n = 0, 1, 2, 3, \ldots, N-1$$  \hspace{1cm} (2.1)

Two new independent variables are defined here as the following,

$$n_1 = 0, 1, 2, \ldots, N_1-1$$

$$n_2 = 0, 1, 2, \ldots, N_2-1$$  \hspace{1cm} (2.2)

and the completely general linear equation which maps $n_1$ and $n_2$ to $n$ is given by

$$n = \langle K_1 n_1 + K_2 n_2 \rangle_N$$
This defines a relation between all allowed $n_1$ and $n_2$ in (2.1) and a value for $n$. The subscript $N$ here means the modular of $N$. It has been proved in [6] that a pair $k_i$ always exists such that the map in (2.3) is single value.

A notation in this chapter will be used,

$$(N,M) = L$$

It means that the greatest common divisor of two numbers $M$ and $N$ is $L$.

Case 1: $N_1$ and $N_2$ are relatively prime, i.e. $(N_1, N_2) = 1$

The integer map of (2.3) is unique if and only if the following is true:

$$(K_1 = aN_2 \text{ and/or } K_2 = bN_1 \text{ and } (K_1, N_1) = (K_2, N_2) = 1$$
where $a, b$ are integers.

**Case 2:** $N_1$ and $N_2$ are not relatively prime, 

i.e., $(N_1, N_2) > 1$

The map of $(2.3)$ is called a prime factor Map (PFM) when

$$\text{PFM: } K_1 = aN_2 \text{ and } K_2 = bN_1$$

The map of $(2.3)$ is called a common factor map (CFM) when

$$\text{CFM: } K_1 = aN_2 \text{ or } K_2 = bN_1 \text{, but not both}$$

### 2.1.2 The Cooley-Tukey FFT Algorithm

The Cooley-Tukey FFT uses the CFM on both the time and frequency index of $(2.3)$,

$$n = \langle K_1 n_1 + K_2 n_2 \rangle_N$$

$$k = \langle K_3 k_1 + K_4 k_2 \rangle_N$$

\[ (2.4) \]

with conditions of CFM: i.e.,

$$k_1 = aN_2 \text{ or } k_2 = bN_1 \text{ but not both}$$
Next, these maps are applied to the definitions of the DFT by defining the two dimensional arrays for the input data and its DFT as

\[ x(n_1, n_2) = x(K_1 n_1 + K_2 n_2) \]
\[ X(k_1, k_2) = X(K_3 k_1 + K_4 k_2) \]

An example of the CFM used in the Cooley-Tukey radix-4 FFT for a length-16 DFT is

\[ N = 16 = 4^2 \]
\[ n = 4n_1 + n_2 \]
\[ k = k_1 + 4k_2 \]

The resulting arrays of indices are
\[
\begin{array}{ccc}
\hline
n_1 & 0 & 1 & 2 & 3 \\
\hline
0 & 0 & 4 & 8 & 12 \\
1 & 1 & 5 & 9 & 13 \\
2 & 2 & 6 & 10 & 14 \\
3 & 3 & 7 & 11 & 15 \\
\hline
\end{array}
\]

\[
n_2 =
\begin{array}{ccc}
\hline
k_1 & 0 & 1 & 2 & 3 \\
\hline
0 & 0 & 1 & 2 & 3 \\
1 & 4 & 5 & 6 & 7 \\
2 & 8 & 9 & 10 & 11 \\
3 & 12 & 13 & 14 & 15 \\
\hline
\end{array}
\]

\[k_2 = n_1 \]

The substitution of these changes of variables into the definition of the DFT given in (1.3) result in

\[
\hat{\mathbf{x}} = \sum_{n_2=0}^{N_2-1} \sum_{n_1=-1}^{N_1-1} \hat{x} W_{k_1 k_2 n_1 n_2} W_{k_1 k_3 n_1 k_3} W_{k_2 k_4 n_2 k_3} W_{k_3 k_4 n_2 k_2}
\]

\[\hat{\mathbf{x}} \quad (2.6)\]

\(k_i\) can be chosen in such a way that the calculations are uncoupled and the arithmetic operation is reduced. The requirements for this are
\[ \langle K_1 K_4 \rangle_N \text{ or } \langle K_2 K_3 \rangle_N = 0 \text{ but not both} \]

\[ \text{--------- (2.7)} \]

In order that each short sum be a short DFT, the following must also hold:

\[ \langle k_1 k_3 \rangle_N = N_2 \text{ and } \langle k_2 k_4 \rangle_N = N_1 \]

the simplest set of coefficients that meets all these requirements are

\[ a = d = k_2 = k_3 = 1 \]

So the index maps become

\[ n = N_2 n_1 + n_2 \]

\[ k = k_1 + N_1 k_2 \]

\[ \text{--------- (2.8)} \]

and use (2.6), (2.7)

\[ \hat{X} = \sum_{n_2 = 0}^{N_2-1} \sum_{n_1 = 0}^{N_1-1} \hat{x} W_{N_1}^{n_1 k_1} W_{N_2}^{n_2 k_2} \]
where $W_N = \exp(-j2\pi/N)$,

The choice of the $k_1$ uncouples the calculations since the first sum over $n_1$ for $n_2=0$ calculates the DFT of the first row of the data array $x(n_1, n_2)$, and those data values are never needed in the succeeding calculations. Both the row and the column calculations are independent.

If one defines

$$f(k_1, n_2) = \sum_{n_1=0}^{N_1-1} x(n_1, n_2) W_{N_1}^{n_1 k_1}$$

and

$$g(k_1, n_2) = f(k_1, n_2) W_N^{k_1 n_2}$$

then the calculation of the DFT is complete with

$$\hat{x}(k_1, k_2) = \sum_{n_2=0}^{N_2-1} g(k_1, n_2) W_{N_2}^{n_2 k_2}$$

\[ (2.9) \]
The most common and most efficient forms of the FFT use all dimensions of the same length. This is called the radix of the algorithm. The DFT of the length $N$ is thus related to the radix $R$ by

$$N = R^M$$

which gives $M$ dimensions, each of length $R$. The short length-$R$ DFTs for $R=2$ and $4$ require no multiplications, and those for $8$ and $16$ requires very few. For a radix 2 FFT, $N = N_1 N_2 = (N/2) * 2$

so the index maps become

$$n = (N/2)n_1 + n_2$$

$$k = k_1 + 2k_2$$

and

$$f(k_1, n_2) = \sum_{n_1=0}^{N/2} x(n_1, n_2) (-1)^{n_1 k_1}$$

$$g(k_1, n_2) = f(k_1, n_2) W_N^{k_1 n_2}$$

--- (2.10)
The FORTRAN statements for this calculation could be:

\[ T = X(1) + X(2) \]

\[ X(2) = X(1) - X(2) \]

\[ X(1) = T \]

\[ T = Y(1) + Y(2) \]

\[ Y(2) = Y(1) - Y(2) \]

\[ Y(1) = T \]

--------- (2.12)

The calculations for X are the same as (2.12). The real parts are saved in array X and the imaginary parts are saved in array Y. The flowgraph for \( N = 2^3 \) is given in Fig. 2.1.

From Fig. 2.1, it is seen that CFM allows the intermediate results to be written back over the original
data in such a way that, after all the calculations are finished, the DFT will be in the array originally occupied by the input data sequence. Separate I/O arrays are then not required, thus allowing a substantial saving of memory.

Unfortunately, the use of In-Place calculations causes the order of the DFT terms to be permuted and the original input data are destroyed. A procedure called bit reversal to reorder the sequence is necessary in an FFT. In-place calculation of the transform elements is of some interest in systems with very small memories, which was the case in the '60s and early '70s, but memory is cheap today so that separate arrays for input and output, which doesn't destroy the input data, is generally more useful. If there are N complex data points, the input needs 16N bytes (double precision), the output needs 16N bytes, and the SIN table calculation needs 8N bytes. For a 1024-point transform, this is still only around 40 K bytes.
2.2 The Winograd FFT Algorithm

The Winograd Algorithm was introduced by Winograd in 1976. This algorithm uses the PFM for indexing. The idea is to operate on the length-\(N\) input data vector by simple additions to form a set of \(M\) intermediate variables where, in general, \(M > N\). These intermediate variables are multiplied by constants which are derived from a combination of the real and imaginary parts of \(W\) in the DFT. These \(M\) products are then combined with simple additions to give the \(N\) values of the DFT. From this point, WFTA uses substantially fewer multiplications than the Cooley-Tukey FFT, but at the expense of more additions. The index mapping is as follows:

\[
\hat{X}(k_1, k_2) = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} \hat{x}(n_1, n_2) W_{N_1}^{n_1 k_1} W_{N_2}^{n_2 k_2}
\]

In the execution of a WFTA, a variable would not first be multiplied by one constant and then by another in the
center two stages. Instead, the two multiplication stages would be premultiplied into one stage of constants by which the variables then are multiplied. This moving of the multiplications together in the centre of the algorithm is responsible for the small number of multiplications in the WFTA.

The FORTRAN program to calculate a WFTA can be found in [2]. From the program, it is obvious that the number of additions has been increased but the multiplications are very few. If the program is executed by a system which has no hardware to perform multiplication, the algorithm is very efficient, but my target system has 8087 math coprocessors which have both multiplication and division instructions, so this algorithm will not be so relatively efficient. Also, the WFTA program is longer, needs more arrays, and is more complicated when compared with the Cooley-Tukey FFT. It also needs more passes through the DO loops and more index calculations. On my target micro vector system, there are
limited memory locations. For all these reasons, the WFTA algorithm will be less suitable than the Cooley-Tukey FFT.

2.3 A FORTRAN Program for the Cooley-Tukey FFT

Figure 2.2 is a FORTRAN subroutine which performs the Radix-2 Cooley-Tukey FFT. The subroutine has actually two parts; the first part is to calculate an FFT and the second part is to do bit reversal. In the first part, the outer loop (DO10 loop) steps through the $M$ dimensions of (2.9), carrying out $f(k_1,n_2)$ and $g(k_1,n_2)$ $M$ times. The next loop (the DO20 loop) steps through the $N_2$ length-2 DFTs. The inner loop (the DO30 loop) steps through the columns. Inside the DO30 loop, the first statement is an address offset that is part of the index map. The next four statements are the length-2 DFT.

This program is very similar to most of the FFT programs being used in practical. Because $N=2^M$, there are
\( M = \log_2 N \) stages. It is easy to select some sample numbers to trace the program and one finds that each stage has approximately \( N/2 \) multiplications. This leads to the next formula which can easily be found:

\[
\text{Complex Multiplications} = (1/2)N \log_2 N
\]

An improved Cooley-Tukey FFT program will be described in Chapter Four. In that program, the sin and cos tables are precalculated before the DO20 loop. Only the sin function is calculated and COS will be found by proper index and sign manipulation. Also, the input data will not be destroyed by means of using additional two arrays for saving output data.
Figure 2.1
8-points, radix-2, in-place FFT

SUBROUTINE FFT(X.Y.N,M)
REAL X(11,Y(11)
N2=N
DO 10 K=1,M
N1=N2
N2=N2/2
E=6.28318530718
A=0
J=1
CONTINUE
DO 104 I=1,M
IF (J.GE.J) GO TO 101
XT=X(I,J)
X(J)AX(I)
X(I)=XT
CONTINUE
J=J+1
100 RETURN
END

N=2*4
n= 4*n1 + n2
k= k1 + 2*k2

k1 = 0 1
0 0 1
1 2 3
2 4 5
3 6 7

k2 =
n1 = 0 1
0 0 4
1 1 5
2 2 6
3 3 7

(A+B)W^k

(A-B)W^k
SUBROUTINE FFT(X, Y, N, M)
REAL X(1), Y(1)

N2 = N
DO 10 K = 1, M
   N1 = N2
   N2 = N2 / 2
   E = 6.283185307179586 / N1
   A = 0.0
   DO 20 J = 1, N2
      C = COS(A)
      S = SIN(A)
      A = J * E
      DO 30 I = J, N, N1
         L = I + N2
         XT = X(I) - X(L)
         X(I) = X(I) + X(L)
         YT = Y(I) - Y(L)
         Y(I) = Y(I) + Y(L)
         X(L) = C * XT + S * YT
         Y(L) = C * YT - S * XT
      30 CONTINUE
   20 CONTINUE
10 CONTINUE
100 J = 1
N1 = N - 1
DO 104 I = 1, N1
   IF (I .GE. J) GO TO 101
   XT = X(J)
   X(J) = X(I)
   X(I) = XT
   XT = Y(J)
   Y(J) = Y(I)
   Y(I) = XT
101 K = N / 2
102 IF (K .GE. J) GO TO 103
   J = J - K
   K = K / 2
   GO TO 102
103 J = J + K
104 CONTINUE
RETURN
END
Chapter Three
Vector Processor and Parallel Processing

A vector system has multiple processors which allow the machine to process a number of data elements in parallel. The Vector processor developed by Dr. Miron has a general purpose processor which is an Intel 8088, a vector controller and a number of Intel 8087 coprocessors. The architecture of the system is shown in Fig. 3.1. The vector controller is designed to coordinate the activities of the math processors. The vector operations are done in the math coprocessors by loading the data elements sequentially into the math coprocessors and executing an 8087 instruction in parallel. Parallel, Scalar and Sequential are the three operating modes of the vector processor. These modes are controlled by the vector controller which recognizes the vector instructions. When there is no instruction involving vector quantities, the system is in scalar mode and the vector controller will have one of the 8087s connected to the general processor.
as a conventional IBM-PC system.

3.1.1 Scalar and Vector Modes

3.1 The Vector Controller

The VC permits a programmer to enter a vector mode or return to scalar mode through vector instructions. The three main functional units forming the vector controller (VC) are:

1) Vector Instructions decoder

It decodes a vector instruction and takes appropriate actions.

2) Sequential LOAD/STORE Control

The effect of the actions of 1) is to load the elements of the required vectors sequentially from Memory into each math coprocessor, or offer the required operations in parallel, write the results out sequentially to memory.

3) Parallel Execution Control
3.1.1 Scalar and Vector Modes

The VC permits a programmer to enter a vector mode or return to scalar mode through vector instructions. The vector instructions are intermixed with the 8088 CPU and 8087 coprocessor instructions. The VC ignores the CPU and the 8087 instructions but decodes the vector instructions and takes necessary actions depending on the type of instruction. The vector processor in the system has two basic modes of operation:

1) the scalar mode, and

2) the vector mode

In the scalar mode, the VC deactivates all the 8087s, except the first one. This single 8087 operates in parallel with the 8088 CPU. This is the normal mode of operation of the system without the intervention of the VC. Every time there is a JUMP instruction in the program,
the VC decodes the next instruction. The JUMP instruction clears the CPU instruction queue in order to provide the signal to the VC to examine the next bus byte, to see if it is a vector instruction. If the target instruction is a vector instruction then the VC causes the system to enter the requested mode, otherwise the VC ignores the instruction. We note from the above statements that whenever a vector instruction is to be included in the program, the vector instruction must be the target of a JUMP instruction.

3.2 Vectorization of A Scalar Program

The starting point of vectorizing a scalar is to study the data dependences in a FORTRAN program. My intention is to apply this study to the parallel processing of the FFT. Both the scalar and vectorized programs will be given and discussed in Chapter Four.

Many programs written for computer systems
contain varieties of loops, and nests of loops, for performing repetitive operations on sequences of data. The Cooley-Tukey FFT stated in Chapter Two was composed of three loops, in the first part, which were nested. These programs direct that operations be done in a well-defined order. Because scalar machines have historically been the most widely available type of machines, the order is one that is readily executable on a scalar system. On a vector system, however, where successive elements in the order are processed in parallel, this very same order may not be valid. There may exist other orders in which the elements may be processed correctly, but the analysis is required to discover both the valid orders and the parts of the program for which the vector machine may be used. This analysis and its result is commonly known as vectorization. My aim in this thesis paper is to begin with a program written for a scalar system which is written for conventional IBM-PC or IBM 4381 main frame and then vectorize the algorithm to make it execute on a multi-processor system.
Generally speaking, the steps to vectorize a program are,

1. To identify as many as possible of the source program statements which may be vectorized.

2. To identify which of the loops surrounding these statements may be used to vectorize them.

3. Use proper vector instructions to program the whole or part of the program.

4. To leave the rest of the program undisturbed in a highly optimized scalar object code.

3.2.1 Program Dependence

The vectorization is based upon the program
dependence theory. In general, a statement in a nest of DO-loops may be vectorized if it does not require, as an input on one iteration of a loop, a value it computed on an earlier iteration of the loop. When a value computed in one iteration of a DO-loop is not used in a later iteration, all of the data values can be computed in parallel. This independence of data values from one DO-loop iteration to the next is a key factor in allowing execution of the statement on a vector machine.

3.2.2 Data Dependence

The dependence that may arise when two statements reference the same storage location is called data dependence. Data dependencies arise in one of three ways:

1. A statement T depends upon a statement S when S defines a value and T references it. This is called true dependence.
Clearly, $S$ must execute before $T$ can execute because $S$ defines a value used by $T$.

2. A statement $T$ depends on a statement $S$ when $S$ references a value and then $T$ redefines it. This is called anti-dependence.

Again, $S$ must execute before $T$ because otherwise $T$ would change the variable $X$ and $S$ would use the wrong value.

3. A statement $T$ depends on a statement $S$ when $S$ stores a value which $T$ also stores. This is called output
dependence. Presumably, some statement between S and T needs the value in X produced by S.

\[ S: X = \]
\[ T: X = \]

S must execute before T or else the wrong value will be left behind in the variable X.

### 3.2.3 Control Dependence

A dependence may arise when one statement determines whether a second statement will be executed. This is called control dependence.

\[ \text{DO } 10 \text{ I}=1,N \]

1. \[ \text{IF(A(I).GT.0.0) GOTO 3} \]
2. \[ \text{A(I)=B(I) + 1.0} \]
3. \[ \text{CONTINUE} \]

Clearly, statement 1 must execute before statement 2 can
execute. Statement 2 depends on statement 1.

3.2.4 Dependence Level

Dependencies attach to a particular DO-loop levels in the loops surrounding a group of statements. Some dependencies are always present

DO 5 J=

DO 5 I=

S: V(I,J) = A(I,J) * B(I,J)

T: Z(I,J) = V(I,J)

5 CONTINUE

T always depends on S because, on each iteration in every loop, there is a true dependence involving the variable V.

DO 5 I=2,N

A(I) = A(I-1) + 1
There are true dependencies at the level of the loop with index I; an element of the array on iteration 2, for example, will be fetched on iteration 3. But there is no dependence at level J, since no element stored on one iteration of the loop is referenced on any other iteration.

3.2.4 Dependence Interchange Problem

When a given loop in a nest of DO-loops is chosen for execution in vector hardware, each vector
instruction will operate on successive data elements selected by that given DO-loop index. For example, if the loop with J index was vectorized in the nest

\[
\begin{align*}
\text{DO 1 } & \text{ K=1,N} \\
\text{DO 1 } & \text{ J=1,N} \\
\text{DO 1 } & \text{ I=1,N} \\
1 & \text{ A(I,J,K) = A(I+1,J+2,K+3)}
\end{align*}
\]

the vector instructions would fetch the elements of A in the order \((2,3,4),(2,4,4),\ldots,(2,N+2,4)\) and store them in the order \((1,1,1),(1,2,1),\ldots,(1,N,1)\). This is a different order from that which would be used in scalar mode, where the innermost DO-loop, with index I, would cycle most rapidly.

In fact, the vector order is exactly what would be seen in scalar mode if the J-loop was interchanged with the I-loop.
DO 1 K=1,N
DO 1 I=1,N
DO 1 J=1,N

1: A(I,J,K) = A(I+1,J+2,K+3)

In order for a given loop to be chosen for execution in vector hardware, this interchange must be valid. That is, it must preserve the semantics of the program.

For the k-loop in the original example to be vectorizable, the loop ordering

DO 1 J=1,N
DO 1 I=1,N
DO 1 K=1,N

1: A(I,J,K) = A(A(I+1,J+2,K+3)

would have to generate the same program results as the original. Note that the other loops are not permuted. It
is necessary to ask only if the loop of interest may be moved inside all of the others.

Sometimes this loop interchange is not possible. In the nest

```
DO 1 J=1,N
DO 1 I=1,N
1: A(I-1,J+1) = A(I,J)
```

there is a dependence at the level of the J-loop. A value stored on one iteration of J is fetched on the next. Many dependencies do not affect the results of the program when loops are interchanged. But this one does, and the J-loop cannot be interchanged with the I-loop because the answers would change.

In a multi-level nest, a dependence for a loop at some level might be interchangeable part of the way into the inner most level, but then be blocked. Such a
dependence is called "innermost preventing" because the loop at that level can not be the innermost level. If the loop can not be interchangeable into the innermost level then it can not be vectorized.

3.3 The Vectorization Analysis of the FFT

Recall the FFT algorithm written in FORTRAN in Chapter Two. This program takes complex input data in two arrays, the real parts in array X and the imaginary parts in array Y and calculates the DFT in-place, i.e., writes the output back into the X and Y arrays over the input data, which is destroyed. The length of the data must be \( N = 2^M \), and the outer DO10 loop steps through the M stages. The Ws are evaluated inside the DO20 loop by the cosine and sine functions. The actual DFT evaluation is done in the innermost loop, the DO30 loop. The first statement calculates an address offset for the data element. The next four statements calculate the length-2 DFT. The last two statements in the loop are the W factor.
complex multiplications using four real adds.

The in-place output of the basic FFT algorithm is in a scrambled order, as explained in Chapter Two. The last part of the program uses an in-place unscrambler. As the index I steps normally from one to N-1, the index J steps in bit-reverse order as shown in Fig.2.3 and the output is reordered with these two indices.

Apparently, one of the time-consuming operations in the FFT program is the calculations of the cosine and sine functions. Actually, it is not necessary to calculate cosine functions because they use the same angles in radians and the value of cosine could be obtained from sine. Also, this could be eliminated by the use of a precomputed table generated by next subroutine

```
SUBROUTINE INI(N,C,S)
REAL C(1),S(1)
P =6.28319/N
DO 10 K=1,N
    A=(K-1)*P
```
It is seen that only one function needs to be calculated.

For example, if the table is the sine function, the COS table can be found by using the next formula:

\[ \cos(J \cdot Q) = C(J) = \sin(\left(\frac{N}{4} \right) - J) \]

where \( Q = \frac{2\pi}{N} \), and \( N=2 \) will be treated as a special case.

Thus, the above subroutine can be improved as the follows:

```fortran
SUBROUTINE INI(N,C,S)
REAL S(1),C(1)
DATA C(0),C(1),S(0),S(1)/1,-1,0,0/
IF (N.EQ.2) GO TO 100
P=6.28319/N
DO 10 K=0,N-1
   A=K*P
   S(K)=SIN(A)
10 CONTINUE
DO 20 K=0,N-1
   MC=N/4-K
   IF(MC.LT.0) THEN
      MC=-MC
      C(K)=-S(MC)
   ELSE
      C(K)=S(MC)
   ENDIF
20 CONTINUE
```
In the outer DO10 loop, there are four assignment statements and a DO20 inner loop. The four assignments are

1. \( N1 = N2 \)
2. \( N2 = N2/2 \)
3. \( E = 6.28319/N1 \)
4. \( A=0 \)

According to the study of data dependence, it is easily found that statement 1 and statement 3 have true data dependence because statement 1 has to execute before statement 3 execute. Also, statement 1 and statement 2 belong to anti-dependence. Statement 1 must execute before statement 2 because otherwise statement 2 would store the variable \( N2 \) and \( N1 \) would use the wrong value.

Alan Norton and Allan J. Silberger in May,
1987[9] described a Generic Architecture for a shared memory system for parallel processing to support their programming. The structure is shown in Fig. 3-2. From their paper, I found that if a parallel processing system has local memories, it would be very helpful. Statement 1 and 2 can not be processed in parallel because they need the same memory location N2. In a parallel system with local memories, it is quite possible to define some temporary variables. For example, N2/2 could be calculated and assigned to T which is a temporary variable. By the end of processing, T could be saved into N2 to replace the old N2.

Statement 1 and statement 3 could be executed sequentially in the same processor. Thus, the local memory system makes the parallel processing more practical. But since the data dependence exists, more statements may be needed and the processing would not be very efficient. This kind of parallel processing system needs more hardware support and it is not suitable for
microcomputers. The system is very expensive.

The VP system developed by Dr. Miron has no local memories for the coprocessors other than the 8087 stack registers. The coprocessors could access the IBM-PC conventional memory. The Intel 8087 itself has eight stack registers. This means if there are more than eight variables needed when processing the statements, memory locations or registers of the conventional system would be used. It is obvious that needs many time-consuming operations. These problems will be explained in detail in Chapter Four.

DO20 has three statements as the following:

1. \( C = \cos(A) \)
2. \( S = \sin(A) \)
3. \( A = J \times E \)

and DO30, the innermost loop.

As mentioned above, the SIN and COS may be
precomputed. Thus DO20 loop can be modified into:

1    CC=C(IA)
2    SS=S(IA)
3    IA=IA+IE

It is obvious that these statements and the statements in subroutine INI are data independent. The table calculations could be vectorized.

DO30, the innermost loop of the FFT program, actually performs DFT and is of greatest interest for parallel processing. The reasons are based upon the following three points:

1) DO30 may be decomposed into different program segments for each iteration. These segments are having no data dependence. Therefore the data elements could be loaded onto the 8087 stack registers in serial mode and the instructions could be executed in parallel.

2) The DO30 loop mainly performs floating-point additions and multiplications. It will be seen in Chapter
Four that floating-point calculations on the Intel 8087 take much more time than the integer operation performed by the Intel 8088. If the DO30 loop could be executed in parallel, it would save even more time.

3) There are eight different variables in the DO30 loop and they can all be kept in the 8087 stack. The instructions to perform calculations can be carried out by the means of the moving Stack Register up and down in a parallel mode. The loading or storing operation in serial mode is relatively less than other operations in the algorithm.

Before the DO30 loop is decomposed, it is necessary to count the number of loops which will be executed. A simple formula is

\[ n = \text{INT} \left( \frac{(N-J)}{N1} \right) + 1 \]
where n is the number that the loop will be execute. INT means only the integer portion of the result will be used. The DO30 loop could be decomposed into n program segments with the same statements but different array elements as illustrated on Fig. 3-3. After the study of these segments, it is determined that they are independent of each other. In the system shown on Fig. 3-2, these segments could be loaded into each local memory before starting to execute them. After the execution, the results could be sent back to global memory. In the VP processor as mentioned above, the local memory is composed of eight stack registers. Because there are only eight different variables in each segment, it will have no difficult in keeping necessary data in each 8087 math coprocessor. After the process, the results will be saved back to global memory for later calculation.
Fig. 3.1
The Architecture of Micro-Vector System

Intel 8087-1

Intel 8087-2

Intel 8087-3

Vector Controller

Intel 8088 Processor
Figure 3-2
Generic Architecture of a Shared Memory System
Decomposed DO30 Loop

ARRAYI: J, J+N1, J+2N1, ...

ARRAYL: J+N2, J+2N2, J+3N2, ...

n = INT [ (N-J)/N1 ] + 1

\[
\begin{align*}
XT &= X(ARRAYI(1)) - X(ARRAYL(1)) \\
X(ARRAYI(1)) &= X(ARRAYI(1)) + X(ARRAYL(1)) \\
YT &= Y(ARRAYI(1)) - Y(ARRAYL(1)) \\
Y(ARRAYI(1)) &= Y(ARRAYI(1)) + Y(ARRAYL(1)) \\
X(ARRAYL(1)) &= C*XT + S*YT \\
Y(ARRAYL(1)) &= C*YT - S*XT
\end{align*}
\]

I = ARRAYI(1) 
L = ARRAYL(1)

( 8087-1 )

\[
\begin{align*}
XT &= X(ARRAYI(n)) - X(ARRAYL(n)) \\
X(ARRAYI(n)) &= X(ARRAYI(n)) + X(ARRAYL(n)) \\
YT &= Y(ARRAYI(n)) - Y(ARRAYL(n)) \\
Y(ARRAYI(n)) &= Y(ARRAYI(n)) + Y(ARRAYL(n)) \\
X(ARRAYL(n)) &= C*XT + S*YT \\
Y(ARRAYL(n)) &= C*YT - S*XT
\end{align*}
\]

I = ARRAYI(n) 
L = ARRAYL(n)

( 8087-n )
Chapter Four
Scalar and Vectorized Cooley-Tukey FFT Programs

4.1 Vector Instructions

4.1.1 Serial and Parallel Modes

Either of the three vector modes can be entered in the Vector Controller (VC) as shown in Fig. 3.1:

1) Serial mode
2) Parallel mode, or
3) Scalar mode

In the serial mode, the VC activates the 8087s in series. Each of the 8087s in the system is activated one after another. While one of the 8087s is activated, the others are in a wait state. This process continues until the serial mode is terminated by a "return to scalar mode" instruction. The serial mode is used to LOAD the elements of the argument into the 8087s in series.

In the parallel mode all the 8087s are activated simultaneously. All 8087 instructions encountered in the parallel mode will be executed by all 8087s simultaneously. After the completion of the required operations, we again return to the scalar mode.
activated simultaneously. All 8087 instructions encountered in the parallel mode will be executed by all the 8087s simultaneously. After the completion of the required operations, we again return to the scalar mode. Once again a serial mode is entered to retrieve the results of the parallel execution and store it at the address pointed to by the appropriate registers of the CPU. The VP has its own instructions, which currently consists of three instructions although sixteen are potentially available. The hexadecimal codes of the three instructions are:

1) DF FD ; Enter parallel mode
2) DF FE ; Enter scalar mode
3) DF FF ; Enter serial mode

In Assembly language, these were named by Dr. Miron as given below:

1) FVECTOR-OP (DF FD)
2) FSCALAR (DF FE)

3) FVECTOR-SQ (DF FF)

4.1.2 Incorporating the VC Instructions

At present, the VC instructions defined by Dr. Miron are not recognized by the Microsoft Assembler. Hence, I used directives and empty instructions to solve this problem.

I define the VC instructions using the DW directive to keep the hex code of the instruction. For example, the next program is to double three pairs of data elements on the top of the 8087s.

```
81: MOV CX, 3; set counter register
82: JMP S1 ;clear the queue
S1: ;empty instruction
    DW 0DFFFH; Enter serial mode
NEXT1: FLD X[CX];load x on the top register of the 8087
    DEC CX ;CX-1 => CX
    JNZ NEXT1;if CX not zero, go to load next data
    JMP S2 ;empty instruction
```
The program first loads each of the three numbers onto the top registers of each of the three 8087s in turn. Then, the VP simultaneously doubles the contents of all the top registers. Finally, these doubled numbers will be returned to the original memory locations one at a time. The program can not JMP to a DIRECTIVE like DWs. An empty instruction with a label was used in front of each vector instruction.
4.2 FFT Program Executed on Conventional IBM-PC

Fig. 4.1 is an improved version of the program in Fig.2.3 which describes the logic of the Cooley-Tukey FFT algorithm. Fig.4.2 has two programs which are used to execute FFT algorithm on IBM-PC. Fig. 4.3 is a BASIC calling program for Inverse FFT. It uses the same FFT assembly subroutine. The BASIC calling programs deal with the input and output of complex numbers for the FFT. N is the number of complex data. M must be a power of two. The BASIC programs read the complex numbers into arrays A and B. After the CALL statement, array X and Y will contain the output results of the Assembly program, i.e., another sequence of complex numbers. Thus, the original input and output data are not destroyed and can be used for other purposes.

The first part of the Assembly program in Fig.4.2 is composed of DD, DW or DB directives. The first two DDs are used to set up two arrays which will
keep single precision real numbers. Each of them keeps 128 * 4 bytes memory capacity. The program reads data from BASIC and save them into ARRAYA and ARRAYB. I used the JUMP instruction to skip this part so that the Assembler will not treat them as normal instructions.

After the data section, the SIN function will be calculated and saved into the SIN array defined in the data section by DD directives. After the SIN table is set up, COS functions are evaluated and the results are saved into the COS array. The input part is followed to receive data from the BASIC to the Assembly program. Note that the starting value of the index register is four because the program reads the first element of array into ARRAY(1) instead of ARRAY(0) and each single precision real number takes four bytes.

The next step is to push M and N2 onto the 8088 stack for later loop counting and perform four assignment
statements. The sin and cos values will be found from precomputed sin and cos tables.

The DO30 loop is the next program segment. Before the loop, the loop counter is calculated by the formula \( \text{INT}\left(\frac{(N-J)}{N1}\right) + 1 \) and pushed onto stack for DO30 loop counting. In DO30 loop, source index register SI and destination index register are used as index I and L in the loop. SI and DI are multiplied by four before the loop to get a correct address offset of each array element and divided by four later to restore original value to make correct index values. A variety of 8087 instructions are used in this work. The definitions of the instructions can be found in [4].

The DO30 loop is decomposed into program segments as described in Fig. 4.3, where the angles are calculated before entering the DO30 loop. \( \text{ARRAYL} \) and \( \text{ARRAYH} \) will keep these index values for I and L in the DO30 loop. NDP is a memory location to keep the number of the state in the system as an integer value of it is passed by

4.3 The Vectorized FFT Program
The vectorized program in Fig. 4.3 is a modified version of the program in Fig. 4.2. It uses Vector Instructions to vectorize some program segments in the Cooley-Tukey FFT. The general consideration to vectorize the program segment is based upon the next two points:

1) Vectorize the segments which perform floating-point calculation.

2) The segment should have less data Loading/Storing and more arithmetic operations.

Therefore, the segments to evaluate the SIN function value, set up the COS table and precalculate the angles are vectorized as shown in Fig.4.3.

The DO30 loop is decomposed into program segments as described in Chapter Three. The indices are calculated before entering the DO30 loop. ARRAYI and ARRAYL will keep these index values for I and L in the DO30 loop. NDP is a memory location to keep the number of the 8087s in the system and the value of it is passed by
the BASIC program. At the beginning of the procedure the VP is in scalar mode. The vector environment is entered through the FVECTOR-SQ upon which the system enters serial mode. To overcome the difficulty of the MASM being unable to process this Assembly language statement, the hexadecimal code is used instead. Once in serial mode the elements of ARRAYA and ARRAYB are loaded serially into all the available 8087s. The system then enters the scalar mode through the appropriate instruction which resets the VC. Then the statements inside DO30 loop are executed in parallel in 8087s. The number of 8087 in a system is machine dependent. The DO30 loop will then be executed \( \text{INT}[(\frac{(N-J)}{N2+1})/\text{NDP}] + 1 \) times, which is less than the counter value of the DO30 in the program of Fig.4.2.

4.4 The VP Performance Estimation

After I finished programming the Cooley-Tukey Algorithm both on conventional IBM-PC and VP system, it is necessary to evaluate the difference of their
performances. In this section, instruction timings of both programs are counted, analyzed and summarized.

4.4.1 Instruction Timing

The instruction timing is represented as the number of clock periods required to execute a particular form (register-to-register, immediate-to-memory, etc.) of instruction. At 5MHz clock, the clock period is 200ns; at 8MHz, the clock period is 125ns. For the Intel 8088 instruction timing which uses memory operands, and effective addressing time will be added to the instruction timing. For the 8088 instructions on the 16-bit operation, the transfer time needs to be calculated by means of the formula \( n + (4 \times \text{Transfers}) \), where \( n \) is the number of clocks required to execute a given instruction.

The execution of an 8087 instruction involved three principal activities, each of which may contribute to the total execution time of the operation:
1) Instruction fetch
2) Instruction execution
3) Operand transfer

The typical execution time and a range for each instruction can be found in [4].

4.4.2 Timing for the FFT Program on IBM-PC

The following formula was obtained after counting the number of clocks of the FFT program in Fig. 4.2.

Number of Clocks =

\[
1208 + N^2 \sum_{K=1}^{M} 7965 + \sum_{J=1}^{N_2} (594 + \sum_{J=1}^{N_1} (953 + \text{INT}((N-J)/N_1+1) * 4085))
\]

where \(N_2 = N/2, N/4, N/8, \ldots,\)

\(N_1 = N, N/2, N/4, \ldots,\)

That is, when \(K=1, 2, \ldots, M,\) the combinations of \(N_2\) and \(N_1\) will be \((N/2, N), (N/4, N/2), \ldots,\) individually. The details could be found in Appendix 1.
4.4.3 Timing for the Vectorized FFT program

The counting of clock periods in the Fig.4.3 result in the next formula:

Number of Clocks =

\[ N \times 5800 + 1332 \times \frac{N}{NDP} + NDP \times 299 + 1614 + \]

\[ \sum_{M}^{N} \sum_{N2}^{N} \left( 527 + \left( \text{INT}\left( \frac{N-J}{N1+1} \right) \right) \times 319 + \right. \]

\[ \left. \left( \text{INT}\left( \frac{N-J}{N1+1} \right) \right) / NDP \times (2365 + NDP \times 481) \right) \]

\[ N1 = N, N/2, N/4, \ldots, \]
\[ N2 = N/2, N/4, N/8, \ldots, \]

where NDP is the number of the 8087s in the system and see Appendix 2 for details.

4.5 Conclusion

The Table 4.1 is based on the two formulas given in section 4.4. From the Table 4.1, it would be noted that when \( N \) is relatively smaller, the difference between the scalar and vectorized program is around 20%. When the
value of $N$ is getting larger, the difference will be maintained around 32%. In order to improve the VP performance, the following two points could be considered:

1) Try to modify the structure of scalar program and vectorize as many of the segments as possible. Note that the serial modes must be use as little as possible.

2) The Loading/Storing is the bottle-neck in this system. The following small vectorized program is to load three integers onto the 8087 and subtract themselves.

```assembly
    MOV CX,3 ;SET COUNTER BECAUSE 3 NDPS
    MOV SI,2 ;ACCESS FROM DATA(1)
    JMP L1
L1:
    DW ODFFFH ;ENTER SERIAL MODE
L2:
    FILD DATA [SI]
    ADD SI,2
    DEC CX
    JNZ L2
    JMP L3
L3:
    DW ODFFEH ;RETURN TO SCALAR MODE
    MOV CX,3 ;RESET COUNTER REGISTER
    MOV SI,2
```
JMP L4

L4:
JMP L5

L5:

The Table 4.2 shows the number of clock periods for each segment:

<table>
<thead>
<tr>
<th></th>
<th>SUB Operation</th>
<th>Storing</th>
<th>Others</th>
<th>TOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loading</td>
<td>270</td>
<td>85</td>
<td>435</td>
<td>61</td>
</tr>
</tbody>
</table>

Obviously, the Loading and Storing took most of the execution time because they were in a serial mode. Thus, serial mode should be reduced. Table 4.3 is a scalar program which does exactly the same thing as the above vectorized one.

MOV CX, 3
From the Table 4-3, the Subtraction operation takes much more time than the parallel one. The total difference between these two programs is small. But, if we have more instructions being executed in parallel, we will certainly same more time than the scalar programs. Thus, a parallel processing system is not always faster. The vector processor is suitable for 1) floating-point operation with fewer memory accesses. 2) more than one instructions which can be executed in parallel. The more instructions that are executed in parallel, the more time will be
saved.
Table 4.1

<table>
<thead>
<tr>
<th>Single NDP</th>
<th>N</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>20770</td>
<td>45455</td>
<td>98401</td>
<td>212039</td>
<td>455401</td>
<td>974891</td>
<td>2079997</td>
<td>4423055</td>
<td>9575457</td>
<td>19813427</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(18.9)</td>
<td>(20.2)</td>
<td>(20.1)</td>
<td>(19.5)</td>
<td>(18.7)</td>
<td>(17.9)</td>
<td>(17.1)</td>
<td>(16.4)</td>
<td>(17.6)</td>
<td>(17.5)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>16842</td>
<td>36289</td>
<td>78621</td>
<td>170688</td>
<td>370155</td>
<td>800282</td>
<td>1723449</td>
<td>3696136</td>
<td>7894743</td>
<td>13324129</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(19.6)</td>
<td>(28.8)</td>
<td>(31.2)</td>
<td>(32.7)</td>
<td>(32.7)</td>
<td>(32.6)</td>
<td>(32.2)</td>
<td>(31.8)</td>
<td>(32.8)</td>
<td>(32.8)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16690</td>
<td>32373</td>
<td>67644</td>
<td>142743</td>
<td>306390</td>
<td>657301</td>
<td>1410692</td>
<td>3017331</td>
<td>6434658</td>
<td>13324129</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>(19.3)</td>
<td>(29.1)</td>
<td>(31.4)</td>
<td>(33.4)</td>
<td>(33.2)</td>
<td>(33.3)</td>
<td>(33.4)</td>
<td>(34.8)</td>
<td>(33.4)</td>
<td>(33.5)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>16774</td>
<td>32228</td>
<td>67536</td>
<td>141190</td>
<td>304101</td>
<td>650656</td>
<td>1385759</td>
<td>2944478</td>
<td>6239469</td>
<td>13185084</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(18.4)</td>
<td>(29.0)</td>
<td>(36.0)</td>
<td>(38.7)</td>
<td>(39.5)</td>
<td>(39.9)</td>
<td>(40.1)</td>
<td>(40.2)</td>
<td>(41.5)</td>
<td>(40.2)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>16939</td>
<td>32260</td>
<td>63013</td>
<td>130037</td>
<td>275630</td>
<td>585764</td>
<td>1244293</td>
<td>2643249</td>
<td>5605486</td>
<td>11854368</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>(17.4)</td>
<td>(28.8)</td>
<td>(36.0)</td>
<td>(38.7)</td>
<td>(40.3)</td>
<td>(41.0)</td>
<td>(41.4)</td>
<td>(41.6)</td>
<td>(43.0)</td>
<td>(42.0)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vector Processor:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clocks</td>
</tr>
<tr>
<td>NDPs</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

Note: \( p_j = \frac{1 - \text{Clocks of VP/Clocks of Single NDP}}{\text{Number of Clocks}} \times 100\% \)

\( M \) = the number of Clocks
Appendix 1

1. The Calculation for the timing of Fig. 4.2

<table>
<thead>
<tr>
<th>Sin Function</th>
<th>Cos Function</th>
<th>I/O of Data</th>
<th>FFT</th>
<th>Bit Reversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 + N*5131</td>
<td>282+N*754</td>
<td>636</td>
<td>$77 + \frac{M}{k=1} \left[ 594 + \frac{N}{j=1} \left( 954 + \frac{N-J}{N-I} \right) \right] \times 4085$</td>
<td>$150 + N_I \times 1680 + N \times 500$</td>
</tr>
</tbody>
</table>

2. The calculation for the timing of Fig. 4.3

<table>
<thead>
<tr>
<th>Sin Function</th>
<th>Cos Function</th>
<th>I/O of Data</th>
<th>FFT</th>
<th>Bit Reversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{N}{(\text{NDP}+1)} \times (\text{NDP}+299+106)$</td>
<td>$1226 \times (\frac{N}{\text{NDP}+1})$</td>
<td>636</td>
<td>$209 + \frac{M}{k=1} \left[ 527 + \frac{N^2}{j=1} \left( 854 + \frac{N-I}{N-I} \right) \right] \times 319$</td>
<td>$150 + N_I \times 1680 + N \times 500$</td>
</tr>
</tbody>
</table>

$N$: the No. of Data
$\text{NDP}$: the No. of the Intel 8087s in the system
$N_I$: $N-1$
$A$: Integer division
$B$: Integer division
SUBROUTINE FFT(A,B,X,Y,N,M)
REAL A(1),B(1),X(1),S(1),C(1)
DATA C(0),C(1),S(0),S(1)/1,-1,0,0/
X=A
Y=B
IF (N.EQ.2) GO TO 600
P=6.28319/N
DO 70 K=0,N-1
  ANGLE=K*P
  S(K)=SIN(ANGLE)
  CONTINUE
70
DO 150 K=0,N-1
  MC=N/4-K
  IF(MC.LT.0)THEN
    MC=-MC
    C(K)=-S(MC)
  ELSE
    C(K)=S(MC)
  ENDIF
150
CONTINUE
600
N2=N
DO 10 K=1,M
  N1=N2
  N2=N2/2
  IE=N/N1
  IA=0
  DO 20 J=1,N2
    SS=S(IA)
    CC=C(IA)
IA=IA+IE

DO 30 I=J,N,N1
    L=I+N2
    XT=X(I)-X(L)
    X(I)=X(I)+X(L)
    YT=Y(I)-Y(L)
    Y(I)=Y(I)+Y(L)
    X(L)=CC*XT+SS*YT
    Y(L)=CC*YT-SS*XT

    CONTINUE

30 CONTINUE

CONTINUE

10 CONTINUE

J=1

N1=N-1

DO 104 I=1,N1
    IF(I.GE.J) GO TO 101
        XT=X(J)
        X(J)=X(I)
        X(I)=XT
        XT=Y(J)
        Y(J)=Y(I)
        Y(I)=XT

101 K=N/2

102 IF(K.GE.J) GO TO 103

    J=J-K
    K=K/2

103 GO TO 102

104 CONTINUE

RETURN

END
Fig. 4.2

The FFT Program as Programming for a IBM-PC

.8087
.MODEL MEDIUM
.CODE
;-----------------------------------
; Coolley Tukey FFT algorithm
; Programmer: Zhou Ke
; Date : January 13, 1988
; use MICROSOFT QUICKBASIC CALL FFT(X(1),Y(1),A(1),B(1),M,N)
;-----------------------------------

PUBLIC FFT

FFT PROC FAR
JMP START

ARRAYA DD 128 DUP(?) ;SET UP AN ARRAY A
ARRAYB DD 128 DUP(?) ;SET UP AN ARRAY B
COS DD 128 DUP(?) ;SET UP COS TABLE ARRAY
SIN DD 128 DUP(?) ;SET UP SIN TABLE ARRAY
A DD ? ;ANGEL LOCATION
N DW ? ;NUMBER OF THE COMPLEX NO.
N1 DW ? ;ONE OF THE FACTOR OF N
N2 DW ? ;THE SECOND FACTOR OF N
M DW ? ;POWER OF TWO
TWO DB 2 ;KEEP CONSTANT 2 HERE
E DW ? ;KEEP A LOCATION FOR 6.28/N!
L DW ?
C DD ? ;KEEP A LOC. FOR CONSTANT
S DD ? ;KEEP A LOC. FOR CONSTANT
J DW ? ;LOC. OF INDEX OF DO 20
STATUS_WORD DD ?
SIGN_STORE DB ?
MINUS2 DW -2
REALLY_COS DB ?
N11 DD ? ;DUMMY LOCATION
NONE DD ?
P DD ? ;P=2*PI/N
FOUR DW 4 ;KEEP CONSTANT 4 HERE
PITWO DD 6.28319 ;2*PI HERE
N4 DW ? ;SAVE N/4 HERE
NX DW ? ;INTEGER N
START:

PUSH BP ;SAVE CURRENT SP INTO BP
MOV BP,SP
PUSH AX ;SAVE CURRENT AX
PUSH BX ;SAVE CURRENT BX
MOV BX,[BP]+6 ;GET ADDR(N) AND SAVE IT IN BX
MOV AX,[BX] ;GET COUNTER IN AX
CMP AX,2 ;CHECK IF N=2
JNE NORMAL ;NO, GO TO NORMAL PROCESS
XOR DI,DI ;SET DI TO ZERO
XOR SI,SI ;SET SI TO ZERO
FLDZ ;SET TOS TO ZERO
FST DWORD PTR SIN [SI] ;SET SIN(0) TO ZERO
ADD SI,4 ;ADDRESS SIN(1)
FSTP DWORD PTR SIN [SI] ;SET SIN(1) TO 0
FLDI ;LOAD 1 ONTO TOS
FST DWORD PTR COS [DI] ;SET COS(1) TO 1
ADD DI,4 ;READY FOR NEXT ADDRESS
FCHS ;CHANGE SIE
FSTP DWORD PTR COS [DI];SET COS(2) TO -1
JMP FINISH ;GO TO OUTPUT DIRECTLY
NORMAL:
PUSH AX ;SAVE COUNTER INTO STACK
MOV NN,AX ;SAVE COUNTER INTO NN LOCATIO
MOV K,0 ;INITIALIZE K TO ZERO
FLD PI TWO ;GET 2 * PI
FILD NN ;LOAD N
FSTP NN1 ;CHANGE INTEGER TO REAL NUMBER
FDIV NN1 ;GET P ON TOS
FSTP P ;SAVE TO P
XOR SI,SI ;RESET SI TO ZERO
FLD P ;GET P
FILD K ;GET K
FMULP ST(1),ST ;LEAVE ANGEL ON TOP
MOV SIGN_STORE,0 ;ASSUME POSITIVE
FTST ;TEST STACK TOP
FSTSW STATUS_WORD ;GET STATUS WORD
FWAIT
MOV AH,BYTE PTR STATUS_WORD+1 ;GET 1ST BYTE
SAHF ;SET STATUS BIT
JNC NON_NEGATIVE ;JMP IF CF =0
MOV SIGN_STORE,-1 ;ITS NEGATIVE
NON_NEGATIVE:

```
FABS ;CHANGE TO POSITIVE, ABSOLUTE VALUE
     MOV REALLY_COS,0 ;SIN , NOT COS
     FILD MINUS2 ;LOAD MINUS INTEGER
     FLDPI ;LOAD PI
     FSSCALE
     FSTP ST(1) ;DUMP -2
     FXCH
```

RANGE:

```
FPREM ;FIND PARTIAL REMAINDER
     FSTSW STATUS_WORD
     FWAIT
     MOV AH,BYTE PTR STATUS_WORD+1
     SAHF
     JP RANGE ;THIS TESTS BIT C2
     CMP REALLY_COS,0
     JE ITS_SINE
     XOR AH,01000000B
     TEST AH,01000000B
     JNZ NOCARRY
     XOR AH,1B
     ITS_SINE:
```

NOCARRY:

```
     FTST
     FSTSW STATUS_WORD
     FWAIT
     MOV BX,0
     AND BYTE PTR STATUS_WORD+1,01000000B
     CMP BYTE PTR STATUS_WORD+1,01000000B
     JNE NOT_ZERO
     MOV BX,-1
     NOT_ZERO:
```

```
     TEST AH,10B ;IS C1 ON
     JZ C1ISOFF ;JUMP IF OFF
     CMP BX,0 ;ST EXACTLY ZERO
     JNE ST0ANDC1 ;JUMP IF YES
     FSUBP ST(1),ST ;NOW PI/4-ST
     FPTAN
     JMP SINDONE
     ST0ANDC1:
```

```
     FSTP ST ;POP ST
     FSTP ST ;AND PI/4
     FLD1 ;LOAD RATIO 1 TO
     FLD1
```
```
C1ISOFF:
    JMP SINDONE
FSSTP ST(1) ; GET RID OF PI/4
CMP BX,0 ; ST EXACTLY ZERO ?
JNE STOANDNOC1 ; JMP IF YES
FPTAN
JMP SINDONE
STOANDNOC1:
    FSTP ST ; DUMP ST
FLDZ ; READ RATIO 0 TO 1
FLD1
SINDONE:
    MOV BX,0 ; ASSUME C3 OFF
    TEST AH,01000000B
    JZ NOC3 ; JUMP IF OFF
    MOV BX,1 ; NOTE C3 ON
NOC3:
    TEST AH,10B
    JZ NOC1 ; JUMP IF OFF
    XOR BX,1
    JMP DOSINE
NOC1:
    XOR BX,0
DOSINE:
    CMP BX,1
    JNE SINFUNC
FXCH
SINFUNC:
    FMUL ST(0),ST(0) ; ST(0) = Y*Y
    FLD ST(1) ; ST(0) = X
    FMUL ST(0),ST(0) ; ST(0) = X*X
    FADDP ST(1),ST(0) ; ST(0) = X*X+Y*Y
    FSQRT
    FDIVP ST(1),ST(0)
    TEST AH,1B
    JZ C0OFF
C0OFF:
    CMP SIGN_STORE,0
    JE LEAVE_POS
    FCHS
LEAVE_POS:
    FSTP DWORD PTR SIN [SI]
    ADD SI,4 ; ADDRESS NEXT Elf.
    INC K ; K = K+1
    POP CX
    DEC CX ; CX-1 TO CX
    JZ L2
    PUSH CX ; PUSH NEW COUNTER
    FNINIT ; RESET 8087 REGISTERS
    FWAIT
L2:
    JMP LOOP1
```
XOR DX,DX ;SET DX TO ZERO
MOV AX,NN ;SAVE N INTO AX
IDIV FOUR ;FIND N/4 AND KEEP IT IN AX
MOV N4, AX
XOR DX,DX
MOV AX,NN ;SAVE N INTO COUNTER REGISTER
MOV CX,NN ;SAVE N INTO COUNTER REGISTER
MOV K, 0
MOV BX,K ;SAVE K INTO BX
MOV DX,N4 ;TRANSFER N/4 TO DX
SUB DX,BX ;FIND N/4 - K HERE
CMP DX,0 ;CHECK IF N/4 - K < 0
JGE L3 ;NO, GO TO L3
NEG DX ;YES, CHANGE THE SIGN
MOV AX,DX ;READY FOR MULTIPLICATION
XOR DX,DX
IMUL FOUR ;GET CORRECT ADDRESS OFFSET
MOV BX,AX ;READY FOR LOADING SIN ELEMENT
FLD DWORD PTR SIN [BX];LOAD PROPER SIN ELEMENT
Fchs ;CHANGE IT'S SIGN
FSTP DWORD PTR COS [DI]
JMP L4 ;JUMP TO READY NEXT ADDRESS
L5:
MOVCX,CX
XOR DX,DX ;CLEAR DX FOR MULTIPLICATION
IMUL FOUR ;READY NEXT ADDRESS
MOV BX,AX ;SAVE NEW ADDRESS
FLD DWORD PTR SIN [BX] ;GET PROPER SIN VALUE
FSTP DWORD PTR COS [DI];ASSIGN IT TO COS TABLE
ADD DI,4
INC K ;FIND K+1 TO K
DEC CX ;CHECK IF DONE
JNZ L3 ;NOT DONE, GO TO KEEP LOOPING
FINISH:

MOV BX,[BP]+8 ;GET ADDR(M)
MOV AX,[BX] ;SAVE M INTO AX REGISTER
MOV M,AX ;AND M LOCATION
MOV BX,[BP]+6 ;GET ADDR(N)
MOV AX,[BX] ;SAVE N FOR COUNTING
MOV N,AX ;SAVE AX TO N
MOV DX,AX ;SAVE N INTO DX REGISTER
MOV CX,AX
MOV BX,[BP]+12 ;GET ADDR(ARRAYA)
MOV SI, 4 ; START FROM ARRAY(1)
FNINIT

RC1:
FLD DWORD PTR [BX] ; LOAD NUMBER INTO ARRAYA
FSTP DWORD PTR ARRAYA [SI]
DEC CX ; DONE?
JNZ RR1 ; NO, GO BACK
JMP RB ; YES, GO TO RB

RR1:
ADD SI, 4 ; ADDRESS NEXT NUMBER
ADD BX, 4 ; ADDRESS NEXT NUMBER
JMP RC1

RB:
MOV BX, [BP]+10 ; GET ADDR (ARRAYB)
MOV CX, DX ; GET ORIGINAL COUNTER
MOV SI, 4 ; INITIALIZE THE COUNTER REGISTER

RR2:
FLD DWORD PTR [BX] ; INPUT NUMBERS FOR ARRAYB
FSTP DWORD PTR ARRAYB [SI]
DEC CX
JNZ TT
JMP DON

TT:
ADD SI, 4
ADD BX, 4
JMP RR2

DON:
MOV CX, M
PUSH CX
MOV AX, N ; SAVE N INTO AX
MOV N2, AX ; FINISH N TO N2

DO10:
XOR DX, DX ; INITIALIZE THE DX REGISTER
MOV AX, N2 ; SAVE N2 TO AX
MOV N1, AX ; FINISH SAVING N2 INTO N1
IDIV TWO ; FIND N2/2. Q IN AX, R IN DX
MOV N2, AX ; FINISH N2/2 TO N2

XOR DX, DX
MOV AX, N
IDIV N1
MOV E, AX

MOV A, 0

MOV J, 1 ; MOVE 1 TO J
MOV AX, N2
PUSH AX
R2:
XOR DX,DX
MOV AX,A
IMUL FOUR
MOV BX,AX
FLD DWORD PTR SIN [BX]
FSTP S
FLD DWORD PTR COS [BX]
FSTP C
MOV BX,A
MOV AX,E
ADD AX,BX
MOV A,AX

;START DO30
XOR DX,DX
MOV AX,N
SUB AX,J
IDIV N1
XOR DX,DX
INC AX
PUSH AX

: I=SI, L=DI
MOV SI,J
MOV AX,SI
ADD AX,N2
MOV DI,AX
FNINIT ; CLEAR REG. IN 8087
MOV AX,SI
MUL FOUR
MOV SI,AX
MOV AX,DI
MUL FOUR
MOV DI,AX
FLD DWORD PTR ARRAYA [SI]
FLD DWORD PTR ARRAYA [DI]
FLD DWORD PTR ARRAYA [SI]
FSUB ST,ST(1)
FXCH ST(2)
FADD ST,ST(1)
FSTP DWORD PTR ARRAYA [SI]
FSTP NONE ; GIVE UP OLD X(L

: GET YT=Y(I)-Y(L)
FLD DWORD PTR ARRAYB [SI]
FLD DWORD PTR ARRAYB [DI]
FLD DWORD PTR ARRAYB [SI]
FSUB ST,ST(1)
FXCH ST(2)
FADD ST,ST(1)
FSTP DWORD PTR ARRAYB [SI]
FSTP NONE ;GIVE UP Y(L)
FLD DWORD PTR S
FLD DWORD PTR C
FMUL ST,ST(3) ;C*XT
FINCSTP
FMUL ST,ST(1)
FDECSTP
FADD ST,ST(1)
FSTP DWORD PTR ARRAYA [DI]
FSTP NONE
FLD C
FLD S
FINCSTP
FMUL ST,ST(1)
FDECSTP
FMUL ST,ST(3)
FSUBP ST(1),ST
FSTP DWORD PTR ARRAYB [DI]
FSTP NONE
FSTP NONE
POP CX
DEC CX
JNZ SKIP1
JMP SKIP2

SKIP1:
PUSH CX
XOR DX,DX
MOV AX,SI
IDIV FOUR
MOV SI,AX
MOV AX,N1
ADD SI,AX
JMP R1

SKIP2:
POP CX
DEC CX
JNZ SKIP3
JMP SKIP4

SKIP3:
PUSH CX

MOV AX, J
INC AX
MOV J, AX
JMP R2

SKIP4:
POP CX
DEC CX
JNZ SKIP5
JMP SKIP6

SKIP5:
PUSH CX
JMP DO10

SKIP6:
MOV SI, 1
MOV AX, N ; SAVE N INTO AX
SUB AX, 1
MOV N1, AX
PUSH AX ; SAVE N1

DO104:
MOV DI, 1
R104:
CMP DI, SI
JGE R101
FNINIT
MOV AX, SI
IMUL FOUR
MOV SI, AX
MOV AX, DI
IMUL FOUR
MOV DI, AX
FLD DWORD PTR ARRAYA [SI]
FLD DWORD PTR ARRAYA [DI]
FXCH ST(1)
FSTP DWORD PTR ARRAYA [DI]
FSTP DWORD PTR ARRAYA [SI]
FLD DWORD PTR ARRAYB [SI]
FLD DWORD PTR ARRAYB [DI]
FXCH ST(1)
FSTP DWORD PTR ARRAYB [DI]
FSTP DWORD PTR ARRAYB [SI]
XOR DX,DX
MOV AX,SI
IDIV FOUR
MOV SI,AX
XOR DX,DX
MOV AX,DI
IDIV FOUR
MOV DI,AX
XOR DX,DX
MOV AX,N
IDIV TWO
R101:

R102:
CMP AX,SI
JGE R103
SUB SI,AX
XOR DX,DX
IDIV TWO
JMP R102

R103:
ADD SI,AX
POP CX
DEC CX
JNZ R
JMP OUT1

R:
PUSH CX
INC DI
JMP R104

OUT1:
MOV CX,N
MOV SI,4
MOV BX,[BP]+16
RE1:
FLD DWORD PTR ARRAYA [SI]
FSTP DWORD PTR [BX]
ADD BX,4
ADD SI,4
DEC CX
JNZ RE1
MOV CX,N
MOV SI,4
MOV BX,[BP]+14

RE2:
FLD DWORD PTR ARRAYB [SI]
FSTP DWORD PTR [BX]
ADD BX, 4
ADD SI, 4
DEC CX
JNZ RE2
JMP ALLEDONE

ALLEDONE:
POP BX
POP AX
POP BP
RET 12

FFT
ENDP
END
Fig. 4.3
Vectorized FFT program

.JM 8087
.MODEL MEDIUM
.CODE

;------------------------------------------
; Coolley Tukey FFT algorithm
; Programmer: Zhou Ke
; Date: MAY 3, 1988
; USE MICROSOFT QUICKBASIC CALL FFT(NDP,A(1),B(1),N,N)
; FILE NAME: PARA-FFT.ASM
; USE MULTI-PROCESSOR SYSTEM, N 8087s ASSUMED
; NDP: NUMBER OF 8087 AVAILABLE IN THE SYSTEM
;------------------------------------------

PUBLIC FFT
PROC FAR
JMP START

ARRAYA DD 128 DUP(?) ;SET UP AN ARRAY
ARRAYB DD 128 DUP(?) ;SET UP AN ARRAY
COS DD 128 DUP(?) ;SET UP COS TABLE
SIN DD 128 DUP(?) ;SET UP SIN TABLE
ARRAYI DW 128 DUP(?) ;SET INDEX I ARRAY
ARRAYL DW 128 DUP(?) ;SET INDEX L ARRAY
INDEX DW 128 DUP(?)
ANGLE DW 128 DUP(?) ;ARRAY FOR SAVING ANGLES
NDP DW ? ;NUMBER OF 8087 IN THE SYSTEM
N DW ? ;NUMBER OF THE COMPLEX NO.
N1 DW ? ;ONE OF THE FACTOR OF N
N2 DW ? ;THE SECOND FACTOR OF N
N3 DW ? ;COUNTER LOCATION
M DW ? ;POWER OF TWO
TWO DB 2
TEMP DD 128 DUP(?)
E DW ? ;KEEP A LOCATION FOR 6.28/N1
A DW ?
L DW ?
C DD ? ;KEEP A LOC. FOR CONSTANT
S DD ? ;KEEP A LOC. FOR CONSTANT
J DW ? ;LOC. OF INDEX OF DO 20
STATUS WORD DW ?
SIGN_STORE
MINUS2
REALLY_COS
N11
NONE
P
FOUR
PITWO
N4
PI2
RN
K
INDEXI
INDEXL
START:

DB ?
DW -2
DB ?
DD ?
DD ?
DD ?
DD ?
; P = 2*PI/N
DW 4
; KEEP CONSTANT 4 HERE
DD 6.28319
; 2*PI HERE
DW ?
; SAVE N/4 HERE
DD ?
; FOR REAL N
DW ?
; FOR ANGEL CALCULATION
DW ?

PUSH BP
MOV BP,SP
PUSH AX
PUSH BX
PUSH CX
PUSH DX
PUSH SI
PUSH DI

MOV BX,[BP]+14
; GET ADDR(NDP)
MOV AX,[BX]
; SAVE NDP IN AX
MOV NDP,AX
; SAVE NUMBER OF NDP
MOV BX,[BP]+6
; GET ADDR(N) AND SAVE:
MOV AX,[BX]
; GET COUNTER IN AX
CMP AX,2
; CHECK IF N=2

JNE NORMAL
; NO, GO TO NORMAL PROCESS
XOR DI,DI
; SET DI TO ZERO
MOV SI,DI
; SET SI TO ZERO
FLDZ
; SET TOS TO ZERO
FST DWORD PTR SIN[SI]
; SET SIN(0) TO ZERO
ADD SI,4
; ADDRESS SIN(1)
FSTP DWORD PTR SIN[SI]
; SET SIN(1) TO 0
FLD1
; LOAD 1 ON TOS
FST DWORD PTR COS[DI]
; SEND IT TO COS
ADD DI,4
; MAKE A CORRECT OFFSET
FCHS ;GET -1  
FSTP DWORD PTR COS [DI] ;SEND IT TO COS TABLE  
JMP FINISH ;SKIP SIN AND COS CACULATION  

NORMAL:  
PUSH AX ;SAVE COUNTER ONTO STACK  
MOV N,AX ;SAVE COUNTER INTO NN  
MOV K,0 ;SET K TO 0  
FLD PI TWO ;GET 2 * PI  
FILD N ;GET INTEGER N  
FSTP RN ;CHANGE IT INTO REAL  
FDIV RN ;GET P ON TOS  
FSTP P ;SAVE P INTO P LOCATION  
MOV AX,N ;GET COUNTER  
MOV BX,NDP ;GET THE NUMBER OF 3987  
MOV CX,BX ;SET CX REGISTER  
IDIV BX ;FIND N/NDP, SAVE IT  
INC AX ;FIND N/NDP + 1  
MOV N3,AX ;SAVE COUNTER  
PUSH AX ;PUSH COUNTER  

;PRECALCULATE ANGLES  
S:  
MOV BX,NDP  
MOV CX,BX  
JMP S1  

S1:  
DW ODFFFH ;ENTER SERIAL MODE  

S2:  
FLD P  
FILD K  
INC K  
DEC CX  
JNZ S2  
JMP S3  

S3:  
DW ODFFFH ;ENTER PARALLEL  
FMULP ST(1),ST ;ANGLE = K*P  
MOV CX,NDP ;RESET COUNTER  
JMP S4  

S4:  
DW ODFFFH ;ENTER SERIAL MODE  

S5:  
FSTP ANGLE [SI] ;SEND ANGLE TO ANGLE ARR.
ADD SI, 4 ; GET CORRECT OFFSET
DEC CX ; CX-1 => CX
JNZ S5 ; CX NOT ZERO, GO TO S6
JMP S6 ; CX IS ZERO, GO TO NEXT STEP

S6:

DW ODFFEH ; ENTER SCALAR MODE
POP CX ; CHECK IF DONE
DEC CX
JE LOOP1 ; YES, GO TO NEXT SEGMENT
PUSH CX ; NO, SAVE NEW COUNTER
JMP S ; BACK TO BEGINING

LOOP1:

FLD ANGLE [SI] ; GET ANGLE ON TOS
MOV SIGN_STORE, 0 ; ASSUME POSITIVE
FTST ; TEST STACK TOP
FSTSW STATUS_WORD ; GET STATUS WORD
FWAIT
MOV AH, BYTE PTR STATUS_WORD+1 ; GET 1ST BYTE
SAHF ; SET STATUS BIT
JNC NON_NEGATIVE ; JMP IF CF = 0
MOV SIGN_STORE, -1 ; ITS NEGATIVE
FABS ; CHANGE TO POSITIVE, ABSOLUTE VALUE

NON_NEGATIVE:

MOV REALLY_COS, 0 ; SIN, NOT COS
FILD MINUS2 ; LOAD MINUS INTEGER
FLDPI ; LOAD PI
FScale
FSTP ST(1) ; DUMP -2
FXCH

RANGE:

FPREM ; FIND PARTIAL REMAINDER
FSTSW STATUS_WORD
FWAIT
MOV AH, BYTE PTR STATUS_WORD+1
SAHF
JP RANGE ; THIS TESTS BIT C2
CMP REALLY_COS, 0
JE ITS_SINE
XOR AH, 0100000B
TEST AH, 01000000B
JNZ NOCARRY
XOR AH, 1B

NOCARRY:

ITS_SINE:

FTST
FSTSW STATUS_WORD
FWAIT
MOV BX, 0
AND BYTE PTR STATUS_WORD+1, 01000000B
CMP BYTE PTR STATUS_WORD+1, 01000000B
JNE NOT_ZERO

NOT_ZERO:

TEST AH, 10B ; IS C1 ON
JZ C1ISOFF ; JUMP IF OFF
CMP BX, 0 ; ST EXACTLY ZERO
JNE STOANDC1 ; JUMP IF YES
FŚUBP ST(1), ST ; NOW PI/4-ST
FPTAN
JMP SINDONE

STOANDC1:

FSTP ST ; POP ST
FSTP ST ; AND PI/4
FLD1 ; LOAD RATIO 1 TO 1
FLD1
JMP SINDONE

C1ISOFF:

FSTP ST(1) ; GET RID OF PI/4
CMP BX, 0 ; ST EXACTLY ZERO ?
JNE STOANDNOC1 ; JUMP IF YES
FPTAN
JMP SINDONE

STOANDNOC1:

FSTP ST ; DUMP ST
FLDZ ; READ RATIO 0 TO 1
FLD1

SINDONE:

MOV BX, 0 ; ASSUME C3 OFF
TEST AH, 01000000B
JZ NOC3 ; JUMP IF OFF
MOV BX, 1 ; NOTE C3 ON
NOC3:
TEST AH, 10B
JZ NOC1; JUMP IF OFF
XOR BX, 1
JMP DOSINE

NOC1:
XOR BX, 0
JMP DOSINE

DOSINE:
XOR BX, 0
CMP BX, 1
JNE SINFUNC
FXCH

SINFUNC:
FMUL ST(0), ST(0) ; Y*Y
FLD ST(1) ; ST(0) = X
FMUL ST(0), ST(0) ; ST(0) = X*X
FADDP ST(1), ST(0) ; ST(0) = X*X + Y*Y
FSQRT
FDIVP ST(1), ST(0)
TEST AH, 1B
JZ COOFF
NOT SIGN_STORE

COOFF:
CMP SIGN_STORE, 0
JE LEAVE_POS
FCHS

LEAVE_POS:
FSTP DWORD PTR SIN [SI]
ADD SI, 4 ; ADDRESS NEXT ELE.
INC K ; K = K + 1
POP CX
DEC CX ; CX - 1 TO CX
JZ L2 ; ZERO, GO TO SET UP
PUSH CX ; PUSH NEW COUNTER
FNINIT ; RESET 8087 REGISTERS
FWAIT
MOV BX, N3 ; SET COUNTER FOR COS
JMP LOOP1
MOV K, 0
XOR DI, DI ; SET DI TO 0
MOV SI, DI ; SET SI TO 0

BEGIN:
MOV CX, NDP
MOV DX, CX ; SAVE THE COUNTER
JMP L ; CLEAR CPU QUEUE
DW ODFFFH ;ENTER SERIAL MODE
L1:
FILD K ;LOAD K ONTO TOE
FILD N4 ;LOAD N/4
INC K ;ADD ONE TO K
DEC CX ;CX - 1 TO CX
JNZ L1 ;IF NOT ZERO, GO TO L1
JMP L2

L2:
DW ODFFDH ;ENTER PARALLEL MODE
FISUB ST(0),ST(1);PERFORM N/4 - K IN PAR.
FIMUL FOUR ;MAKE A CORRECT INDEX VALUE
JMP L3

L3:
DW ODFFEH ;ENTER SCALAR MODE
MOV CX,DX ;SET COUNTER
JMP L4

L4:
DW ODFFFH ;ENTER SERIAL MODE
L5:
FST WORD PTR INDEX [DI];SAVE THE INDEX
ADD DI,4
DEC CX ;CHECK CX =0 ?
JNZ L5
JMP L6

L6:
DW ODFFEH ;RETURN TO SCALAR MODE
MOV CX,BX
DEC CX ;DONE?
MOV BX,CX
JNZ L ;NO, GO TO BEGINING
MOV CX,DX ;MOVE NDP INTO CX
XOR DI,DI ;SET DI TO ZERO
MOV BX,N3
JMP L7

L7:
DW ODFFFH ;ENTER SERIAL MODE
L8:
FILD DWORD PTR INDEX [DI]
DEC CX ;DONE?
JNZ L8 ;NO, DO AGAIN
L9:
JMP L9
DW ODDFH ;ENTER PARALLEL MODE
FILD ST(0) ;DUPLICATE TOS
FCHS ;CHANGE THE SIGN OF TOS
FILD ST(1) ;GET ORIGINAL INDEX NUMBER
FABS ;FIND ITS ABSOLUTE VALUE
FXCH ;EXCHANGE THE POSITION
FIDIV ST(1) ;FIND -X/ABS(X)
JMP L10

L10:
DW ODFFEH ;RETURN TO SCALAR MODE
MOV CX,DX
JMP L11

L11:
DW ODFFFH ;ENTER SERIAL MODE

L12:
FLD DWORD PTR SIN INDEX [DI]
DEC CX
JNZ L12
JMP L13

L13:
DW ODDFDH ;ENTER PARALLEL MODE
FIMUL ST(1) ;MULTIPLY THE SIGN
JMP L14

L14:
DW ODFFEH ;SCALAR MODE
MOV CX,DX
JMP L15

L15:
DW ODFFFH ;ENTER SERIAL MODE

L16:
FSTP DWORD PTR COS [SI]
ADD DI,4
DEC CX
JNZ L16
JMP L17

L17:
DW ODFFEH ;ENTER SCALAR MODE
MOV CX,BX
DEC CX
MOV BX,CX
JNZ L7

FINISH:
MOV BX,[BP]+8
MOV AX,[BX]
MOV M,AX
MOV BX,[BP]+6 ;GET COUNTER
MOV AX,[BX] ;CX HAS THE COUNTER
MOV N,AX
MOV DX,N
MOV CX,N
MOV BX,[BP]+12 ;GET ADDR (ARRAYA)
MOV SI,4
FNINIT

RC1:
FLD DWORD PTR [BX]
FSTP DWORD PTR ARRAYA [SI]
DEC CX
JNZ RR1
JMP RB

RR1:
ADD SI,4
ADD BX,4
JMP RC1

RB:
MOV BX,[BP]+10 ;GET ADDR (ARRAYA)
MOV CX,DX
MOV SI,4

RR2:
FLD DWORD PTR [BX]
FSTP DWORD PTR ARRAYB [SI]
DEC CX
JNZ TT
JMP DON

TT:
ADD SI,4
ADD BX,4
JMP RR2

DON:
MOV CX,M
PUSH CX
DO10:

MOV AX,N ;SAVE N INTO AX
MOV N2,AX ;FINISH N TO N2
XOR DX,DX ;INITIALIZE THE DX REGISTER
MOV AX,N2 ;SAVE N2 TO AX
MOV N1,AX ;FINISH SAVING N2 INTO N1
IDIV TWO ;FIND N2/2. Q IN AX, R IN DX
MOV N2,AX ;FINISH N2/2 TO N2
MOV AX,N
IDIV N1
MOV E,AX
MOV A,0
MOV J,1 ;MOVE 1 TO J
MOV AX,N2
PUSH AX

R2:

MOV AX,A
IMUL FOUR
MOV BX,AX
FLD DWORD PTR SIN [BX]
FSTP S
FLD DWORD PTR COS [BX]
FSTP C
MOV BX,A
MOV AX,E
ADD AX,BX
MOV A,AX
XOR DX,DX
MOV AX,N
SUB AX,J
IDIV N1
XOR DX,DX
INC AX
PUSH AX ;PUSH THE COUNTER INTO STACK

:CALCULATE INDEX I AND INDEX L AND SAVE THEM INTO ARRAY I AND ARRAY L
MOV SI,4 ;SAVE STARING OFFSET INTO SI
POP CX ;POP COUNTER INTO CX
PUSH CX ;PUSH IT BACK TO STACK
MOV AX,J
REP:
IMUL FOUR ;GET CORRECT ADDRESS
MOV WORD PTR ARRAY[SI], AX ;SAVE I
MOV BX, N2 ;SAVE INTEGER N2 INTO BX
ADD AX, BX ;J+N2 => AX
IMUL FOUR ;MULTIPLY BY FOUR
MOV WORD ARRAYL[SI], AX ;SAVE L
ADD AX, N1 ;J+N1 => N1
ADD SI, 4 ;ADDRESS NEXT ELE.
DEC CX ;CX-1=>CX
JNZ REP ;CX=0? NO, GO TO REP

;INDECE ARE IN TWO ARRAYS INDIVIDUALLY
POP AX
IDIV NDP ;FIND N/NDP
INC AX
PUSH AX ;SAVE IT ON STACK
MOV INDEXI, 0
MOV INDEXL, 0

REPEAT1:
MOV AX, NDP ;GET THE NUMBER OF TIMES
PUSH AX ;GET COUNTER

REPEAT:
LEA BX, ARRAYA ;GET ADDRESS OF ARRAYA
MOV ADDRESS, BX ;SAVE THE ADDRESS
JMP SKIP1

SKIP1:
DW ODFD FH ;ENTER PARALLEL MODE
FLD ST(0) ;DUPLICATE THE TOS
JMP SKIP2

SKIP2:
DW ODFEEH ;RETURN TO SCALAR MODE
MOV SI, ARRAYI ;SAVE I INTO SI
MOV DI, INDEXI ;MOVE I TO DI REGISTER
POP CX
PUSH CX
JMP SKIP3

SKIP3:
DW ODFFFFH ;ENTER SERIAL MODE
R2:
FLD DWORD PTR ADDRESS [SI] ;GET N
FWAIT
ADD DI, 2 ;INCREMENT NY 2
MOV SI, ARRAY[DI] ; LOAD INDEX INTO SI
DEC CX ; CX-1 => CX
JNZ R2 ; DONE? NO, GO TO READ NEXT
JMP SKIP4 ; YES, GO TO SKIP4

SKIP4:
DW ODFFEH ; RETURN TO SCALAR MODE
POP CX
PUSH CX
MOV SI, ARRAYL
MOV DI, INDEXL
JMP SKIP5

SKIP5:
DW ODFFFFH ; ENTER SERIAL MODE

R3:
FLD ADDRESS [SI]
FWAIT
ADD DI, 2
MOV SI, ARRAYL[DI]
DEC CX
JNZ R3
JMP SKIP6

SKIP6:
DW ODFFEH ; ENTER SCALAR MODE
JMP SKIP7

SKIP7:
DW ODFFDH ; ENTER PARALLEL MODE

R4:
FST ST(2), ST ; TRANSFER ST TO ST(2)
FSUBR ST, ST(1) ; X(I) - X(L) => TOS
FINCSTP ; INCREMENT SP BY ONE
FADD ST, ST(1) ; X(I) + X(L) => X(I)
FDECSTP ; SP-1 => SP
FWAIT ; MAKE SURE 8087 HAS DONE
JMP SKIP16

SKIP16:
DW ODFFEH ; BACK TO SCALAR MODE

; NEXT PART IS TO FIND YT AND Y(I)
MOV CX, FLAG1 ; GET FLAG IN CX
DEC CX
JNZ AGAIN ; DO ARRAYB
AGAIN:
JMP NEXT
LEA BX, ARRAYB ; POINT ARRAY B
MOV ADDRESS, BX ; SAVE THE ADDR.
MOV FLAG1, 1 ; CHANGE FLAG
JMP SKIP1

NEXT:
POP CX
PUSH CX
JMP SKIP17

SKIP17:
DW ODFFFH ; ENTER SERIAL MODE
R5:
FLD C ; LOAD C ON TOS
FLD S ; LOAD S ON TOS
FWAIT
DEC CX ; CX-1 =&gt; CX
JNZ R5 ; NO DONE, GO TO R5
POP CX
PUSH CX
JMP SKIP21

SKIP21:
DW ODFFDH ; ENTER PARALLEL MODE
FMUL ST, ST(2)
FINCSTP
FMUL ST, ST(4)
FDECSTP
FWAIT
JMP SKIP28

SKIP28:
DW ODFFEH ; RETURN TO SCALAR MODE
MOV AX, FLAG2
CMP AX, 1
JZ CHANGE ; GO TO CHANGE SIGN
JMP SKIP29

CHANGE:
JMP T3

T3:
DW ODFFDH ; INTO PARALLEL MODE
FCHS ; CHANGE SIGN
FWAIT
JMP T4
T4:

```
DW ODFFFEH ; SCALAR MODE
JMP SKIP29
```

SKIP29:

```
DW ODFFFDH ; PARALLEL MODE
FADD ST, ST(1)
FXCH ST(7)
FIN CSTP
FIN CSTP
FXCH ST(3)
FIN CSTP
FXCH ST(3)
FIN CSTP
FXCH ST(3)
FIN CSTP
FXCH ST(3)
FDEC CSTP
FDEC CSTP
FDEC CSTP
FDEC CSTP
FWAIT
JMP SKIP46
```

SKIP46:

```
DW ODFFFEH ; RETURN TO SCALAR
MOV CX, FLAG2
DEC CX
JNZ NEXT2
JMP NEXT3
```

NEXT2:

```
MOV FLAG2, 1
MOV CX, 2
JMP T
```

T:

```
DW ODFFFDH ; PARALLEL MODE
FIN CSTP
FIN CSTP
FIN CSTP
FIN CSTP
FIN CSTP
FIN CSTP
FIN CSTP
FWAIT
JMP SKIP48
```
SKIP48:

DW ODFFEH ; SCALAR MODE
POP CX
PUSH CX
MOV DI,4
MOV SI,ARRAYI
JMP SKIP49

SKIP49:

DW ODFFFFH ; ENTER SERIAL

R10:

FST ARRAYB[SI]
FWAIT
MOV SI,ARRAYI[DI]
ADD DI,4
DEC CX
JNZ R10
JMP SKIP51

SKIP51:

DW ODFFDH ; PARALLEL MODE
FINCSTP ; POINT TO X(L)
FWAIT
JMP SKIP52

SKIP52:

DW ODFFEH ; SCALAR MODE
POP CX
PUSH CX
MOV DI,4
MOV SI,ARRAYL
JMP SKIP53

SKIP53:

DW ODFFDH ; SERIAL MODE
R11:

FST ARRAYB[SI] ; RETURN X(L)
MOV SI,ARRAYL[DI]
ADD DI,4
DEC CX
JNZ R11
JMP SKIP54

SKIP54:

DW ODFFEH ; SCALAR MODE
SKIP55:
JMP SKIP55

DW ODFFFDH ; PARALLEL MODE
FINCSTP
FINCSTP
FWAIT
JMP SKIP56

SKIP56:
DW ODFFEH ; SCALAR MODE
MOV SI, ARRAYI
MOV DI, 4
POP CX
PUSH CX
JMP SKIP57

SKIP57:
DW ODFFFFFFFH ; SERIAL MODE
R13:
FST ARRAYA[SI]
FWAIT
MOV SI, ARRAYI[DI]
INC DI
DEC CX
JNZ R13
JMP SKIP58

SKIP58:
DW ODFFEH ; SCALAR MODE
JMP SKIP59

SKIP59:
DW ODFFFDH ; PARALLEL MODE
FINCSTP ; POINT TO YI L
FWAIT
JMP SKIP60

SKIP60:
DW ODFFEH ; SCALAR MODE
POP CX
PUSH CX
MOV SI, ARRAYL
MOV DI, 1
JMP SKIP61
SKIP61:

DW ODFFFH ;SERIAL MODE
R14:
FST ARRAYA[SI]
FWAIT
MOV SI,ARRAYL[DI]
ADD DI,4
DEC CX
JNZ R14
JMP SKIP62

SKIP62:

DW ODFFEH ;SCALAR MODE
POP DX ;DUMP NDP
POP CX ;GET REAL DO30 LOOP COUNTER
DEC CX ;CX-1=>CX
JE EXIT1 ;IF ZERO, GET RID OF DO:
PUSH CX ;OTHERWISE, PUSH NEW COUNTER
MOV AX,DX ;GET NDP
IMUL FOUR ;4*NDP
ADD AX,COUNTER ;FIND NEXT STARTING OFFSET
MOV INDEXI,AX ;SAVE THEM INTO INDEXI
MOV INDEXL,AX ;SAVE THEM INTO INDEXL
JNZ ALLDONE ;ANY NUMBER LEFT?
JMP REPEAT1
JMP EXIT1

EXIT1:

POP AX ;GIVE UP DO30 COUNTER NOW
POP CX
DEC CX ;CX=0,N2=0
JNZ REC1
JMP EXIT2
Rec1:
MOV AX,J.
ADD AX,1
MOV J,AX
PUSH CX
JMP DO20

EXIT2:

POP AX
POP CX
DEC CX
JNZ REC2
REC2:
    JMP EXIT3
    INC AX
    MOV K, AX
    PUSH CX
    JMP D010
EXIT3:
    MOV SI, 1
    MOV CX, N0
    POP CX
SKIP6:
    MOV SI, 1
    MOV AX, N; SAVE N INTO AX
    SUB AX, 1
    MOV N1, AX
    PUSH AX; SAVE N1
D0104:
    MOV DI, 1
R104:
    CMP DI, SI
    JGE R101
    FNINIT
    MOV AX, SI
    IMUL FOUR
    MOV SI, AX
    MOV AX, DI
    IMUL FOUR
    MOV DI, AX
    FLD DWORD PTR ARRAYA [SI]
    FLD DWORD PTR ARRAYA [DI]
    FXCH ST(1)
    FSTP DWORD PTR ARRAYA [DI]
    FSTP DWORD PTR ARRAYA [SI]
    FLD DWORD PTR ARRAYB [SI]
    FLD DWORD PTR ARRAYB [DI]
    FXCH ST(1)
    FSTP DWORD PTR ARRAYB [DI]
    FSTP DWORD PTR ARRAYB [SI]
    XOR DX, DX
    MOV AX, SI
    IDIV FOUR
MOV SI, AX
XOR DX, DX
MOV AX, DI
IDIV FOUR
MOV DI, AX

R101:
XOR DX, DX
MOV AX, NO
IDIV TWO

R102:
CMP AX, SI
JGE R103
SUB SI, AX
XOR DX, DX
IDIV TWO
JMP R102

R103:
ADD SI, AX
POP CX
DEC CX
JNZ R
JMP OUT1

R:
PUSH CX
INC DI
JMP R104

OUT1:
MOV CX, NO
MOV SI, 4
MOV BX, [BP] + 16

RE1:
FLD DWORD PTR ARRAYA [SI]
FSTOP DWORD PTR [BX]
ADD BX, 4
ADD SI, 4
DEC CX
JNZ RE1
MOV CX, N
MOV SI, 4
MOV BX, [BP] + 18

RE2:
FLD DWORD PTR ARRAYB [SI]
FSTOP DWORD PTR [BX]
ADD BX, 4
ADD SI, 4
DEC CX
JNZ RE2
JMP ALLDONE

ALLDONE:

POP DI
POP SI
POP DX
POP CX
POP BX
POP AX
POP BP
RET 14

ENDP

END
Fig. 4.4
BASIC Inverse FFT Calling Program

75 DIM A(128), B(128), X(128), Y(128)
80 INPUT N
82 INPUT M
85 FOR I = 1 TO N
90 INPUT A(I), B(I)
100 NEXT I
110 CALL FFT(X(1), Y(1), A(1), B(1), M, N)
150 FOR I = 1 TO N
160 Y(I) = -Y(I)
170 NEXT I
180 REM USE THE OUTPUT OF FFT AS THE INPUT OF IFFT
200 CALL FFT(A(1), B(1), M, N)
210 REM RESULTS ARE THE SAME AS THE ORIGINAL INPUT OF FFT
220 PRINT "", A(I)/N, -B(I)/N
230 NEXT I
390 END
References


