NISTIR 4480

ASSEMBLY CODE TO COMPUTE SINE AND COSINE USING THE CORDIC ALGORITHM

John A. Horst

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards
and Technology
Robot Systems Division
Unmanned Systems Group
Bldg. 220 Rm B124
Gaithersburg, MD 20899

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
John W. Lyons, Director

NIST

NISTIR 4480

ASSEMBLY CODE TO COMPUTE SINE AND COSINE USING THE CORDIC ALGORITHM

John A. Horst

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Robot Systems Division
Unmanned Systems Group
Bldg. 220 Rm B124
Gaithersburg, MD 20899

December 1990

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

Assembly Code to Compute Sine and Cosine Using the CORDIC Algorithm

John Albert Horst

National Institute of Standards and Technology (NIST)

Bldg 220, Rm B-124, Gaithersburg, MD 20899, USA

Abstract

The CORDIC algorithm is commonly used to approximate certain elementary functions. Many microprocessor and microcontroller chips without the availability of math coprocessor chips could benefit from the efficient implementation of this algorithm. The focus of this work is to report on a specific implementation in assembly code (for an 8051 microcontroller) that computes the sine and cosine to eleven bits of accuracy.

1. Introduction

From the early 1970's and into the 1980's, the CORDIC (COordinate Rotation DIgital Computer) algorithm (first used by Volder [4]) has been selected for use in many hand-held calculators offering the multiply, divide, square root, sine, cosine, tangent, arctangent, sinh, cosh, tanh, arctanh, ln, and exp functions [1]. The CORDIC algorithm's usefulness for these calculators can be seen in that all of these functions can be approximated using the same set of iterative equations (in binary form) [2]

$$\begin{aligned} x_{k+1} &= x_k - m\delta_k y_k 2^{-k} \\ y_{k+1} &= y_k + \delta_k x_k 2^{-k} \\ z_{k+1} &= z_k - \delta_k \varepsilon_k \\ \delta_k &= \pm 1, \quad \text{for } k = 0,1,\ldots,n, \end{aligned} \tag{1}$$

where $m = 1$, 0, or -1, is a mode indicator and ε_k are constants stored prior to the execution of the algorithm and depend on m. Appropriate selection of initial values, x_0, y_0, z_0, and the sign of each δ_k will generate approximations of each of the elementary functions mentioned.

Many modern microprocessors and microcontrollers do not have high speed hardware multipliers on-chip making function approximation by polynomial methods relatively slow. This explains the utility and popularity of math coprocessor chips in many computers. If, in addition, there is some reason that a math coprocessor chip is not feasible, one might consider using the CORDIC equations in software to compute elementary functions on the microprocessor or microcontroller. It would make sense to write this code in assembly language to maximize the speed of execution.

The two-fold task of this report is to include as much of the theory behind the CORDIC iterations (1) as is necessary and to give an example of the CORDIC algorithm in assembly code written for the Intel Corp. 8051 microcontroller. The 8051 does have an on-chip multiplier. However, since the 8051 has only an eight bit multiplier (requiring multiple precision multiplication), the use of polynomial approximation algorithms to approximate the elementary functions may not be faster than the CORDIC iterations.

Since we merely intend to demonstrate the effectiveness of the CORDIC algorithm, only sine and cosine functions will be considered.

This work was sponsored by the US Bureau of Mines (BOM) in support of their efforts in computer-assisted underground coal mining.

2. Instructions for Use of the 8051 Code to Compute Sine and Cosine

The theory behind the CORDIC algorithm is elegantly presented in [2] and will not be repeated except to mention that on page 322 line 3, x_0 should equal K not $1/K$.

Equations 2 specify fourteen iterations of the CORDIC algorithm with constants and initial values defined for the computation of sine and cosine only. After completion of the fourteenth iteration, x_{14} and y_{14} will give the approximations to cosine and sine, respectively. This will give the sine and cosine of any angle, θ, between 0 and $\pi/2$. This result will be accurate to approximately $\pm 2^{-11} \approx \pm 0.000488$. Angles between $\pi/2$ and 2π can be handled by appropriate domain reduction.

$$
\begin{gathered}
x_{k+1} = x_k - m\delta_k y_k 2^{-k} \\
y_{k+1} = y_k + \delta_k x_k 2^{-k} \\
z_{k+1} = z_k - \delta_k \varepsilon_k \\
\varepsilon_k = \tan^{-1} 2^{-k} \\
\delta_k = \begin{cases} -1, & \text{if } z_k < 0 \\ 1, & \text{if } z_k \geq 0 \end{cases} \\
K = \prod_{k=0}^{13} \cos \varepsilon_k \\
x_0 = K,\ y_0 = 0, \text{ and } z_0 = \theta
\end{gathered}
\tag{2}
$$

A negative aspect of the CORDIC algorithm is that even if the user wants only the sine and not the cosine (or vice versa), the algorithm must compute the undesired quantity as well as the desired one. Note as well that, if one wanted to make the result more accurate (or less accurate), a simple increase (or decrease) in the number of iterations is not sufficient. One must also change the value of K as well as the number of ε_k's stored in memory.

The assembly language program (called CORDIC and listed in the Appendix) declares the following three variables as two-byte (one-word) public variables: ?Angle_16?byte, ?Sine_16?byte, and ?Cosine_16?byte.

Here is the typical way CORDIC can be used: The calling program desires to compute the Sine or Cosine of a 16-bit (one-word) quantity in radians called θ. The calling program stores θ in the two bytes of ?Angle_16?byte, storing the least significant byte at ?Angle_16?byte and the most significant byte at ?Angle_16?byte+1. The CORDIC program requires θ to be a positive number in radians between 0 and 2π. Since the largest possible value of θ, 2π, has three bits to the left of the decimal point, the calling program must send θ with the decimal point assumed to be between bit location 13 and bit location 12 for the 16-bit θ (with numbering of locations from 0 to 15). In other words, the input, θ, has a fixed decimal point location assumed by CORDIC.

3. Two Examples of How θ, the Input to CORDIC, Must Be Represented

Example 1: $\theta = 2\pi$

2π in binary form is 110.0100100010000_2. So, if one wanted the sine of θ when $\theta = 2\pi$, the calling program would put 00010000 at ?Angle_16?byte and 11001001 at ?Angle_16?byte+1. Then CORDIC would be executed after which the sine and cosine would be found as 16-bit public variables in locations ?Sine_16?byte, and ?Cosine_16?byte.

Example 2: $\theta = 0.2984$ radians

Since $0.2984_{10} = 0.0100110001100 1_2$, the calling program would put 10001100 ($8C_{16}$) at ?Angle_16?byte and 00001001 (09_{16}) at ?Angle_16?byte+1.

4. An Example of a Comparison of the Approximation for Sine and Cosine Using CORDIC to the "True" Values

As a simple example of the operation of the CORDIC algorithm, assume that $\theta = 0.2984$ radians as in section 3, example 2. Computing the sine and cosine using the CORDIC algorithm we get that $x_{14} = 0.11110100101100010101_2$ and $y_{14} = 0.01001011001111100100 1_2$. These are approximations for the "exact" values, $\cos 0.2984 = 0.11110100101011111101_2$ and $\sin 0.2984 = 0.0100101101000011_2$. A comparison of the above two sets of binary numbers shows that the CORDIC algorithm is accurate only to about the eleventh significant binary digit as claimed in section 2. This is because we iterated only fourteen times. One can chose to iterate any number of times up to and including sixteen for varying degrees of accuracy (as long as the appropriate changes in the constants of equations 2 are made). NIST chose a level of accuracy for the algorithm to be that which seems as sufficient for calculations involving the positioning of underground coal mining machines. If it is too accurate or too slow in execution, one can always sacrifice accuracy for speed.

5. Conclusion

The general operation of the CORDIC algorithm has been given with the focus on a specific implementation in 8051 assembly code to compute the sine and cosine to eleven bits of accuracy.

This work can assuredly be expanded. It would be interesting to use a form of the CORDIC algorithm that allows for multiplication [3], making use of the 8051's on chip multiplier. Also useful would be to compare the performance of CORDIC with that of polynomial methods of approximating elementary functions.

The source code listed in the appendix is in the public domain and will be made available to all who request it from the author.

6. Acknowledgements

The author gratefully recognizes the technical help received from Dan Lozier. Much thanks to Ann Hargett and Shirley Jack for help preparing the manuscript.

7. References

[1] G. Haviland. and A. Tuszynski, "A CORDIC arithmetic processor chip", IEEE Trans. Computers, Vol. C-29, No. 2, Feb. 1980, pp 68-79.

[2] Charles W. Schelin, "Calculator function approximation", American Mathematical Monthly, vol. 90, No. 5, May 1983, pp 317-325.

[3] D. Timmermann, H. Hahn, and B. Hosticka, "Modified CORDIC algorithm with reduced iterations," Electronics Letters, July 20, 1989, Vol. 25, No. 15, pp 950-951.

[4] J. Volder, "The CORDIC computing technique," IRE Trans. Computers, vol. EC-8, Sept, 1959, pp 330-334.

8. Appendix

```
NAME        CORDIC
PUBLIC      ?Angle_16?byte,?Cosine_16?byte,?Sine_16?byte
CORDIC_CODE     SEGMENT CODE
CORDIC_DATA     SEGMENT DATA
RSEG        CORDIC_DATA
?Angle_16?byte: DS    2
?Cosine_16?byte:      DS    2
?Sine_16?byte:  DS    2
K:    DS    1
XTMP_0:   DS    1
XTMP_1:   DS    1
YTMP_0:   DS    1
YTMP_1:   DS    1
X_0:  DS    1
X_1:  DS    1
Y_0:  DS    1
```

```
Y_1:  DS    1
Z_0:  DS    1
Z_1:  DS    1
E_0:  DS    1
E_1:  DS    1
RSEG        CORDIC_CODE
E_00:       DB    22H,19H,0D6H,0EH,0D7H,07H,0FBH,03H
   DB       0FFH,01H,00H,01H,80H,00H,40H,00H
   DB       20H,00H,10H,00H,08H,00H,04H,00H
   DB       02H,00H,01H,00H
Angle_16:
   MOV      X_0,#6FH                 ;INITIALIZE X[0]
   MOV      X_1,#13H
   MOV      Y_0,#00H                 ;INITIALIZE Y[0]
   MOV      Y_1,#00H
   MOV      R2,#0                    ;INITIALIZE SIGN INDICATOR
                                     ;REGISTER AS POSITIVE FOR
                                     ;BOTH SINE AND COSINE.
   CLR      C                        ;CLEAR THE BORROW (CARRY)
   BIT.
   MOV      A,?Angle_16?byte         ;PLACE LOWER BYTE OF ANGLE IN
   A
   SUBB     A,#44H                   ;SUBTRACT LOWER BYTE BY PI/2
   MOV      Z_0,A                    ;PLACE RESULT IN LOWER BYTE
                                      OF Z[0]
   MOV      A,?Angle_16?byte+1       ;PLACE UPPER BYTE OF ANGLE IN
                                      ACCUM
   SUBB     A,#32H                   ;SUBT (WITH BORROW) UPPER
                                      BYTE OF PI.
   MOV      Z_1,A                    ;PLACE RESULT IN UPPER BYTE
                                      OF Z[0]
   JC       Add_PiDiv2               ;IF BORROW SET, THE ANGLE
                                     ;WAS [0,PI/2).
 ;NOW CHECK IF THE ANGLE IS IN [PI/2,PI), IF NOT CONTINUE
   MOV      R2,#2                    ;INITIALIZE SIGN INDICATOR
                                     ;REGISTER POSITIVE FOR SINE
                                     ;NEGATIVE FOR COSINE.
   MOV      A,Z_0                    ;PLACE LOWER BYTE OF ANGLE IN
                                     ;ACCUMULATOR
```

```
        SUBB    A,#44H              ;SUBTRACT LOWER BYTE BY PI/2
        MOV     Z_0,A               ;PLACE RESULT IN LOWER BYTE
                                    ;OF Z[0]
        MOV     A,Z_1               ;PLACE UPPER BYTE OF ANGLE IN
                                    ;ACCUM.
        SUBB    A,#32H              ;SUBT WITH BORROW UPPER
                                    ;BYTE BY PI/2
        MOV     Z_1,A               ;PLACE RESULT IN UPPER BYTE
                                    ;OF Z[0]
        JC      Twos                ;IF BORROW SET, ANGLE WAS IN
                                    ;[PI/2,PI)
;NOW CHECK IF THE ANGLE IS BETWEEN PI AND 3PI/2, IF NOT
 CONTINUE
        MOV     R2,#3               ;INITIALIZE SIGN INDICATOR
                                    ;REGISTER NEGATIVE FOR BOTH
                                    ;SINE AND COSINE
        MOV     A,Z_0               ;PLACE LOWER BYTE OF ANGLE IN
                                    ;ACCUM.
        SUBB    A,#44H              ;SUBTRACT LOWER BYTE BY PI/2
        MOV     Z_0,A               ;PLACE RESULT IN LOWER BYTE
                                    ;OF Z[0]
        MOV     A,Z_1               ;PLACE UPPER BYTE OF ANGLE IN
                                    ;ACCUM
        SUBB    A,#32H              ;SUBT (WITH BORROW) UPPER
                                    ;BYTE OF PI
        MOV     Z_1,A               ;PLACE RESULT IN UPPER BYTE
                                    ;OF Z[0]
        JC      Add_PiDiv2          ;IF BORROW SET, ANGLE WAS IN
                                    ;[PI,3PI/2)
;IF WE GET THIS FAR, THE ANGLE IS BETWEEN 3PI/2 AND 2PI.
        MOV     R2,#1               ;INITIALIZE SIGN INDICATOR
                                    ;REGISTER POSITIVE FOR
                                    ;COSINE AND NEGATIVE FOR
                                    ;SINE
        MOV     A,Z_0               ;PLACE LOWER BYTE OF ANGLE IN
                                    ;ACCUM
        SUBB    A,#44H              ;SUBTRACT LOWER BYTE BY PI/2
        MOV     Z_0,A               ;PLACE RESULT IN LOWER BYTE
```

```
                                        ;OF Z[0]
    MOV     A,Z_1                       ;PLACE UPPER BYTE OF ANGLE IN
                                        ;ACCUM.
    SUBB    A,#32H                      ;SUBT WITH BORROW UPPER
                                        ;BYTE BY PI/2
    MOV     Z_1,A                       ;PLACE RESULT IN UPPER BYTE
                                        ;OF Z[0]
Twos:
    MOV     A,Z_0                       ;FORM THE TWOS COMPLEMENT
                                        ;OF Z[0]
    CPL     A
    ADD     A,#1
    MOV     Z_0,A
    MOV     A,Z_1
    CPL     A
    ADDC A,#0
    MOV     Z_1,A
    AJMP    Cordic_Algo
Add_PiDiv2:
    MOV     A,Z_0
    ADD     A,#44H                      ;ADD BACK PI/2
    MOV     Z_0,A
    MOV     A,Z_1
    ADDC    A,#32H
    MOV     Z_1,A
  ;IT IS AT THIS POINT THAT THE Cordic Algorithm BEGINS
Cordic_Algo:
    MOV     DPTR,#E_00                  ;INIT DATA POINTER AT CORDIC
    CONSTANTS
    MOV     R1,#0                       ;INIT THE LOOP COUNTERS
    MOV     K,#0
 ;BELOW IS THE CORDIC LOOP
Cordic_Loop:
    MOV     R0,K                        ;Temporarily store K for Shift_XY
    MOV     XTMP_0,X_0                  ;Temporarily Store X[K]
    MOV     XTMP_1,X_1
    MOV     YTMP_0,Y_0                  ;Temporarily Store Y[K]
    MOV     YTMP_1,Y_1
```

```
    MOV     A,#0                    ;Temporarily Store E[K]
    MOVC    A,@A+DPTR
    MOV     E_0,A
    MOV     A,#1
    MOVC    A,@A+DPTR
    MOV     E_1,A
    INC     DPTR
    INC     DPTR
;SET UP THE CONTROL REGISTER, R3, THAT WILL CONTAIN INFO
;ON THE NEGATIVITY
;OF X[K], Y[K], AND Z[K]
    MOV     R3,#0
    MOV     A,X_1
    ANL     A,#80H
    RL      A
    ORL     A,R3
    MOV     R3,A
    MOV     A,Y_1
    ANL     A,#80H
    RL      A
    RL      A
    ORL     A,R3
    MOV     R3,A
    MOV     A,Z_1
    ANL     A,#80H
    RL      A
    RL      A
    RL      A
    ORL     A,R3
    MOV     R3,A
    INC     R3                      ;THIS STEP REQUIRED FOR
                                    ;LATER DJNZ INSTRUCTIONS
 ;COMPUTE Z[K+1]
    MOV     A,#80H
    ANL     A,Z_1                   ;TEST FOR Z NEGATIVE
    JNZ     Add_Z
    MOV     A,E_0                   ;FORM TWOS COMPLEMENT OF
                                    ;E[K] IF Z[K] IS POSITIVE,
```

```
                                    ;SINCE THEN A SUBTRACTION
                                    ;IS REQUIRED
    CPL     A
    ADD     A,#1
    MOV     E_0,A
    MOV     A,E_1
    CPL     A
    ADDC A,#0
    MOV     E_1,A
    Add_Z:
    MOV     A,E_0
    ADD     A,Z_0
    MOV     Z_0,A
    MOV     A,E_1
    ADDC    A,Z_1
    MOV     Z_1,A
  ;COMPUTE X[K+1] AND Y[K+1]
CASE1:
    DJNZ    R3,CASE2
    ACALL   Shift_XY
    ACALL   Twos_Y_Shfted
    AJMP    Add_XY
CASE2:
    DJNZ    R3,CASE3
    ACALL   Abs_X
    ACALL   Shift_XY
    ACALL   Twos_X_Shfted
    ACALL   Twos_Y_Shfted
    AJMP    Add_XY
CASE3:
    DJNZ    R3,CASE4
    ACALL   Abs_Y
    ACALL   Shift_XY
    AJMP    Add_XY
CASE4:
    DJNZ    R3,CASE5
    ACALL   Abs_X
    ACALL   Abs_Y
```

```
    ACALL   Shift_XY
    ACALL   Twos_X_Shfted
    AJMP    Add_XY
CASE5:
    DJNZ    R3,CASE6
    ACALL   Shift_XY
    ACALL   Twos_X_Shfted
    AJMP    Add_XY
CASE6:
    DJNZ    R3,CASE7
    ACALL   Abs_X
    ACALL   Shift_XY
    AJMP    Add_XY
CASE7:
    DJNZ    R3,CASE8
    ACALL   Abs_Y
    ACALL   Shift_XY
    ACALL   Twos_X_Shfted
    ACALL   Twos_Y_Shfted
    AJMP    Add_XY
CASE8:
    ACALL   Abs_X
    ACALL   Abs_Y
    ACALL   Shift_XY
    ACALL   Twos_Y_Shfted
Add_XY:
  ;FORM X[K+1]
    MOV     A,YTMP_0
    ADD     A,X_0
    MOV     X_0,A
    MOV     A,YTMP_1
    ADDC    A,X_1
    MOV     X_1,A
  ;FORM Y[K+1]
    MOV     A,XTMP_0
    ADD     A,Y_0
    MOV     Y_0,A
    MOV     A,XTMP_1
```

```
    ADDC    A,Y_1
    MOV     Y_1,A
  ;INCREMENT K AND TEST IF WE`VE LOOPED 14 TIMES YET
    INC     K
    INC     R1
    CJNE    R1,#0EH,Long_Jump
    AJMP    Cordic_End
Long_Jump:
    LJMP    Cordic_Loop
Cordic_End:
   ;IF THE COMPUTED ANSWER IS THE NEGATIVE OF THE TRUE
   ;ANSWER,
   ;TEST IF ANSWERS ARE NEGATIVE OR POSITIVE AND CHANGE
   ;SIGN.
    MOV     A,#3                        ;LEAVE SIGN OF
                                        ;ANSWERS POSITIVE IF
                                        ;THE ANGLE IS [0,PI/2) OR R2 = 0
    ANL     A,R2
    JZ      The_End
;
    MOV     A,#2                        ;SKIP NEGATION OF COSINE
                                        ;IF ANGLE IS IN
                                        ;[3PI/2,2PI] OR R2 = 1
    ANL     A,R2
    JZ      Twos_Y
Twos_X:
    MOV     A,X_0                       ;FORM THE TWOS COMPLEMENT
                                        ;OF THE COSINE
    CPL     A                           ;FOR ANGLES IN [PI/2,3PI/2)
    ADD     A,#1                        ;OR R2 = 2 OR 3.
    MOV     X_0,A
    MOV     A,X_1
    CPL     A
    ADDC A,#0
    MOV     X_1,A
Twos_Y:
    MOV     A,#1                        ;SKIP NEGATION OF SINE IF THE
                                        ;ANGLE IS IN [PI/2,PI)
```

```
    ANL     A,R2
    JZ      The_End
    MOV     A,Y_0               ;FORM THE TWOS COMPLEMENT
                                ;OF THE SINE
    CPL     A                   ;FOR ANGLES IN [PI,2PI) OR
    ADD     A,#1                ;EQUIVALENTLY, WHEN R2 = 1
                                ;OR 3.
    MOV     Y_0,A
    MOV     A,Y_1
    CPL     A
    ADDC A,#0
    MOV     Y_1,A
The_End:
    AJMP    The_Real_End
Abs_X:
    CLR     C
    MOV     A,XTMP_0
    SUBB    A,#1
    MOV     XTMP_0,A
    MOV     A,XTMP_1
    SUBB    A,#0
    MOV     XTMP_1,A
    RET
Abs_Y:
    CLR     C
    MOV     A,YTMP_0
    SUBB    A,#1
    MOV     YTMP_0,A
    MOV     A,YTMP_1
    SUBB    A,#0
    MOV     YTMP_1,A
    RET
Shift_XY:
    MOV     A,R0
    JZ      End_Shift_XY
    DEC     R0
    CLR     C
    MOV     A,XTMP_1
```

```
    RRC     A
    MOV     XTMP_1,A
    MOV     A,XTMP_0
    RRC     A
    MOV     XTMP_0,A
    CLR     C
    MOV     A,YTMP_1
    RRC     A
    MOV     YTMP_1,A
    MOV     A,YTMP_0
    RRC     A
    MOV     YTMP_0,A
    AJMP    Shift_XY
End_Shift_XY:
    RET
Twos_X_Shfted:
    MOV     A,XTMP_0
    CPL     A
    ADD     A,#1
    MOV     XTMP_0,A
    MOV     A,XTMP_1
    CPL     A
    ADDC    A,#0
    MOV     XTMP_1,A
    RET
Twos_Y_Shfted:
    MOV     A,YTMP_0
    CPL     A
    ADD     A,#1
    MOV     YTMP_0,A
    MOV     A,YTMP_1
    CPL     A
    ADDC    A,#0
    MOV     YTMP_1,A
    RET
The_Real_End:
    MOV     ?Cosine_16?byte,X_0
    MOV     ?Cosine_16?byte,X_1
```

```
   MOV     ?Sine_16?byte,Y_0
   MOV     ?Sine_16?byte,Y_1
END
```

NIST-114A
(REV. 3-89)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NISTIR 4480

2. PERFORMING ORGANIZATION REPORT NUMBER

3. PUBLICATION DATE
DECEMBER 1990

4. TITLE AND SUBTITLE

Assembly Code to Compute Sine and Cosine Using the CORDIC Algorithm

5. AUTHOR(S)

John A. Horst

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

10. SUPPLEMENTARY NOTES

[ ] DOCUMENT DESCRIBES A COMPUTER PROGRAM; SF-185, FIPS SOFTWARE SUMMARY, IS ATTACHED.

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

The CORDIC algorithm is commonly used to approximate certain elementary functions. Many microprocessor and microcontroller chips without the availability of math coprocessor chips could benefit from the efficient implementation of this algorithm. The focus of this work is to report on a specific implementation in assembly code (for an 8051 microcontroller) that computes the sine and cosine to eleven bits of accuracy.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

8051 assembly code, 8051 microcontroller, calculator, CORDIC, cosine, elementary function approximation, microprocessor, sine

13. AVAILABILITY

[X] UNLIMITED
[ ] FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).
[ ] ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.
[X] ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
19

15. PRICE
A02

ELECTRONIC FORM