TEXAS INSTRUMENTS

Modified Goertzel Algorithm in DTMF Detection Using the TMS320C80

Application Report

1996 Digital Signal Processing Solutions

Printed in U.S.A., June 1996 SPRA066

Modified Goertzel Algorithm in DTMF Detection Using the TMS320C80

Application Report

Chiouguey J. Chen
Digital Signal Processing Solutions—Semiconductor Group

SPRA066
June 1996

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Contents

Appendix

INTRODUCTION

Dual-tone multi-frequency (DTMF) signaling is a standard in telecommunication systems.[1, 3] It has been gaining popularity for some years now because of its numerous advantages over the traditional telephone signaling scheme. In the DTMF scheme, a telephone is equipped with a keypad as shown in Figure 1. The A, B, C, and D keys are usually not present on a regular telephone keypad. Each key represents the sum of a pair of tones. One tone is from the high-frequency group between 1 kHz and 2 kHz, and the other tone is from the low-frequency group below 1 kHz. These frequencies are selected carefully so that the DTMF signal, which is the sum of the two tones, can be distinguished clearly as the signaling tone even in the presence of speech waveforms that might occur on the line.

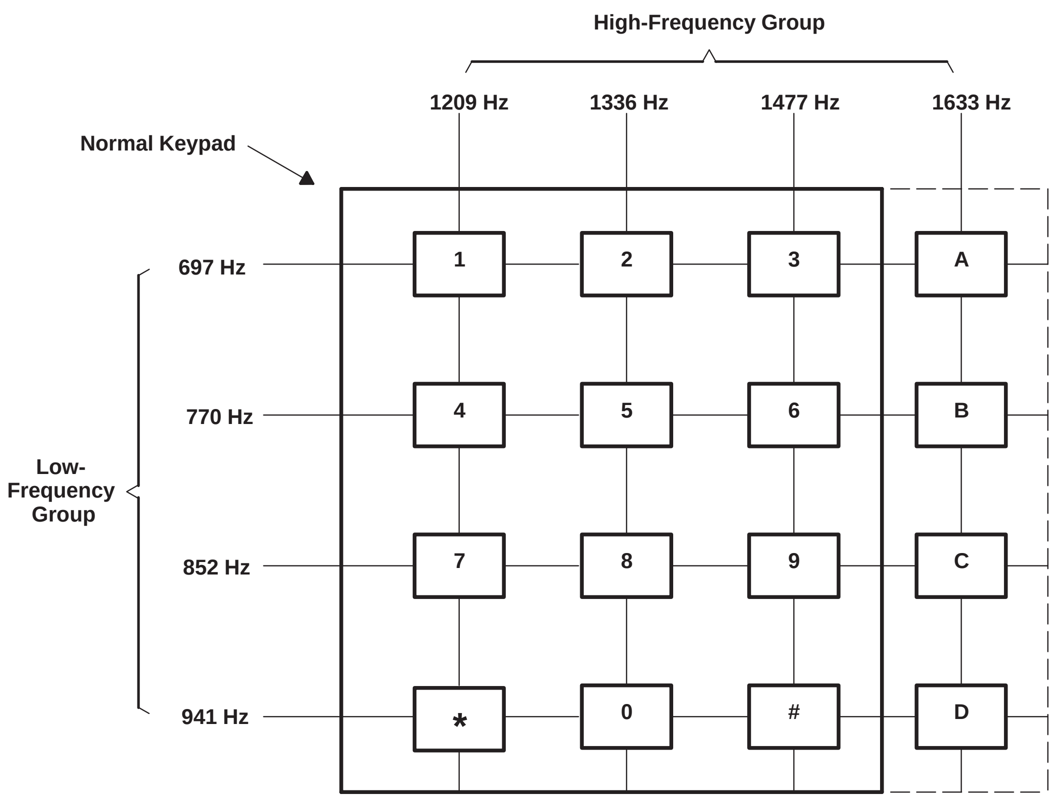

Figure 1. DTMF Keypad

DTMF detection is used to detect DTMF signals in the presence of speech and dialing tone pulses. Besides being used to set up regular calls on a telephone line, DTMF detection is suitable for computer applications such as voice mail and electronic mail, and telephone control features such as conference calling and call forwarding.[3]

The intent of this application report is to demonstrate the C-callable Goertzel DTMF-detection algorithm implementation on one of the TMS320C80's parallel processors (PP)—an advanced 32-bit digital signal processor (DSP) with a 64-bit instruction word. A PP is capable of performing a number of operations in a single clock cycle because of its wide instruction word, three-operand arithmetic logic unit, and single-cycle multiplier. Furthermore, the PP code is allocated with a PP register allocator and assembled with a PP assembler.

This report describes the implementation with the assumption that the received DTMF signal has passed a tone validation and correct timing intervals check, and has been filtered back to its original two tones.

The Goertzel algorithm implementation examines the energy of one of the two tones from an incoming signal at eight different DTMF frequencies to determine which DTMF frequency is present. To do this evaluation, the input signal is transformed to the DTMF frequencies, which are computed by the modified Goertzel algorithm. The matched filter concept is used for each DTMF frequency to determine the frequency at which the incoming signal has maximum energy. Since maximum energy corresponds to DTMF frequency, this procedure enables us to detect the DTMF frequency.

MODIFIED GOERTZEL ALGORITHM

It is important to choose the right algorithm for detection to save memory and computation time. The Goertzel algorithm is the optimal choice for this application because it does not use many constants, which saves a great deal of memory space. Also, only eight DTMF frequencies need to be calculated for this application, and the Goertzel algorithm can calculate selected frequencies. This saves computation time.

The DTMF frequency is transformed to a discrete fourier transform (DFT) coefficient. The relationship between the DTMF frequency (fi) and the DFT coefficient (k) is given in equation (1): [4]

$$k = N \times \frac{fi}{fs} \tag{1}$$

where

fs = Sampling frequency

N = Filter length

Note that k is the nearest integer to equation (1). For each k, the state variable, $v_k(n)$, is obtained by using the recursive difference equation shown in equation (2): [2]

$$v_k(n) = 2 \times \cos\left(\frac{2 \times \pi \times k}{N}\right) \times v_k(n-1) - v_k(n-2) + x(n) \tag{2}$$

where

n = 0, 1, ..., N

Within the same k, equation (2) is iterated until the last state variable, $v_k(N)$, is obtained. Thereafter, the output, $y_k(N)$, is given in equation (3):

$$y_k(N) = v_k(N) - W_N^k \times v_k(N-1) \tag{3}$$

where

$$W_N^k = \exp(-2 \times \pi \times k/N)$$

This is the desired DFT value, that is, $X(k) = y_k(N)$. Equations (2) and (3) are described in the direct-form realization shown in Figure 2. This figure gives an overview of the entire Goertzel algorithm, so that equation (3) is computed once after equation (2) has been calculated N+1 times. Also, k is constant when equations (2) and (3) are evaluated.

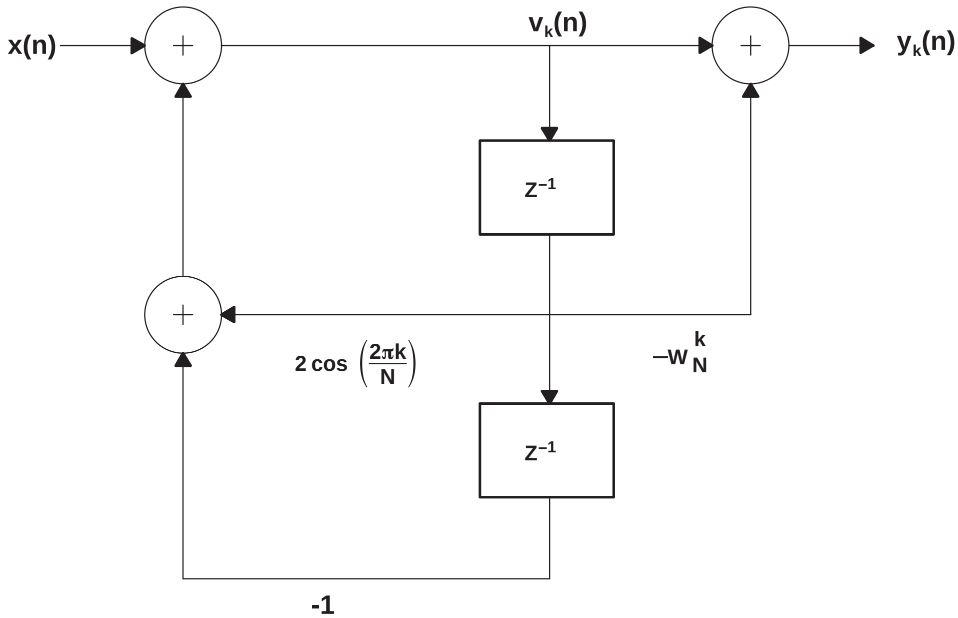

Figure 2. Direct-Form Realization of the Goertzel Algorithm

The Goertzel algorithm is modified further based on the matched filter concept to achieve DTMF detection. The energy of the incoming signal is calculated at the eight DTMF frequencies. The DTMF frequency at which the incoming signal has maximum energy is the *detected frequency.* This energy calculation is given in equation (4):

$$\text{mag\_square} = |X(k)|^2 \tag{4}$$

$$\text{max} = \text{maximum } (\text{max}, \ \text{mag\_square}) \tag{5}$$

In equation (5), max is the maximum energy that initially was set to a zero value and stored in memory. The energy from equation (4) is used for comparison with the stored maximum energy. As soon as the new energy is greater than the stored maximum energy from the comparison, this new energy is stored as the maximum energy for the next comparison. Also, the index that was initialized to a zero value is changed to a number that represents the frequency of this new energy. The comparison is performed for a total of eight times. After the final comparison, the index, a number between 0 and 7 from the result of the comparisons, is returned to the calling program. This number represents the detected DTMF frequency.

IMPLEMENTATION

Since the telephone industry has preset the sampling frequency to 8 kHz and the DTMF frequencies to 697, 770, 852, 941, 1209, 1336, 1477, and 1633 Hz, the filter length must be large enough to find the desired k value that corresponds to the DTMF frequencies. Therefore, there is a trade off to be considered between the computation burden and better resolution. For this application report, the filter length, N, was chosen as 105, which is the smallest value that can fulfill DTMF detection.(3) Table 1 shows the calculated k values for N = 105.

Table 1. Calculated k Values for N = 105

FREQUENCY (Hz)	k
697	9
770	10
852	11
941	12
1209	16
1336	18
1477	19
1633	21

The C-callable modified Goertzel algorithm uses block processing. The algorithm arguments include a filter order, a pointer to the input signal, and a pointer to a structure of filter coefficients. Before the detection algorithm starts, an input signal (a Q8 fixed-point number) is generated for a total of 105 samples and stored in a memory location containing 105 2-byte memory spaces. The values of the coefficients, $\cos(2\pi k/N)$ and $\sin(2\pi k/N)$, are pre-stored in a memory location consisting of 16 2-byte memory spaces. Also, there are two additional load operations in the recursive part of the filter function. Two memory accesses are needed to perform these loads. The algorithm requires 264 bytes of program memory and $2N+64$ bytes of on-chip data RAM. The array index range and fixed-point format of internal processing variables used in the algorithm are shown in Table 2.

Table 2. Internal Processing Variables

NAME	ARRAY INDEX RANGE	FIXED-POINT FORMAT	DESCRIPTION
N	1	Q0	Filter order
M	1	Q0	Number of DTMF frequencies
x	0 to N-1	Q8	Input signal
coefficients	0 to M-1	Q8	Filter coefficients
v1	1	Q8	State variable v(n-1)
v2	1	Q8	State variable v(n-2)
xfi	1	Q8	Detected frequency (imaginary part)
xfr	1	Q8	Detected frequency (real part)
mag_square	1	Q16	Energy of the complex frequency
max	1	Q16	Maximum energy
index_flag	1	Q0	Detected index of the matched frequency

This program takes full advantage of the 64-bit-long instruction word in the TMS320C80's PP to do many parallel operations in one cycle. Therefore, the number of cycles has been reduced to a minimum number of instructions.

To avoid an overflow problem, the input signal and the filter coefficients are limited to a fixed-point Q8 format. The size of the input signal is 16 bits long. The size of the coefficients is 32 bits long wherein the first half is the cosine element and the second half is the sine element. The input signal is loaded during the recursive operation.

One set of coefficients, cosine and sine, is used to compute one DTMF frequency. The software does the memory shift while calculating the new v1 state variable for the memory location. The initial state variables, v1 and v2, are set to zero. When each new v1 is computed, the old v1 is shifted to the v2 memory location. The old v2 is then discarded. Since this is a recursive operation, each state variable is dependent on the previous value. In other words, the number of instructions cannot be reduced. However, this doesn't require many memory spaces.

The recursive operation is repeated N times. The last recursive operation is combined with the non-recursive part. The DFT value, X(k), is calculated separately as its real and imaginary parts. Its energy is obtained by summing the squares of the real and imaginary parts (that is, $|X(k)|^2$). This energy uses a Q16 format to save one instruction of PP code by not shifting back to Q8 format. The energy of the first DTMF frequency is compared to the maximum energy, which was set initially to a zero value and stored in the stack. The greater energy is stored back to the stack along with the index that indicates which frequency has the greater energy.

Next, the state variables are cleared, the pointer is reset to the beginning of the input signal, and the second cycle begins. The cycle is repeated until the last energy is computed. When the comparison process is over, the greatest energy and its corresponding index reside in register d5, which returns an integer to the calling function. The number of cycles for the C-callable Goertzel algorithm is 19+(8+2N)8+4. This number excludes ICACHE misses, and assumes that the filter coefficients buffer and the input memory buffer are stored in different on-chip data RAMs. Figure 3 shows a flow chart of this algorithm.

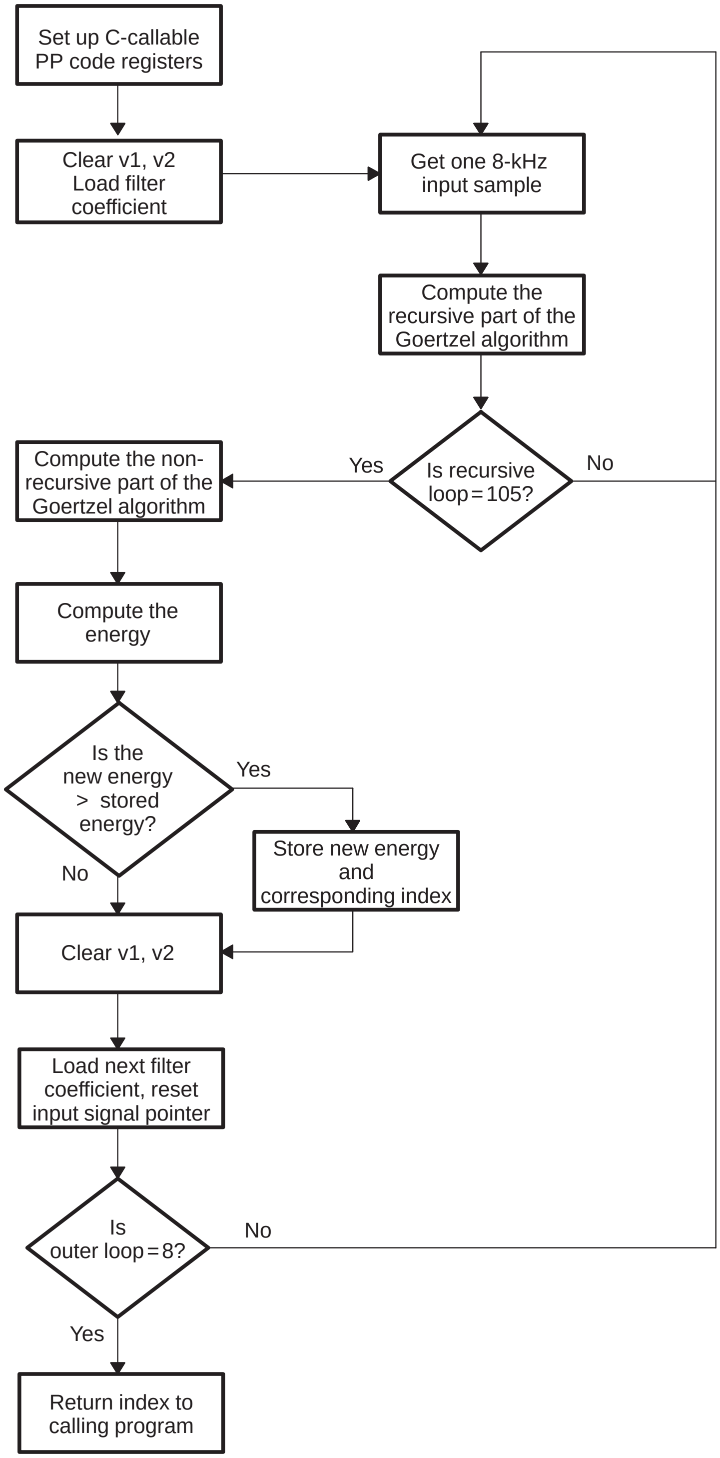

Figure 3. Flow Chart of the Modified Goertzel Algorithm

PERFORMANCE

For this application report, five sets of test cases were generated that were based on DTMF frequencies with different percentage errors (that is, 0%, ±1.5%, and ±3.0%). The test results showed that the modified Goertzel algorithm can detect all the frequencies within an offset range of ±1.5%; however, it does not detect the frequency that has an offset range of ±3.0%.

SUMMARY

The modified Goertzel algorithm can detect the incoming frequency within a ±1.5% offset range. This algorithm does not check for overflow problems, nor is it a complete detection algorithm. To ensure complete detection, further evaluation of the detected tone in the form of many tests is required. These tests could include twist tests, dynamic tests, guard time tests, signal-to-noise ratio tests, and talk off tests.

REFERENCES

1. C. Marven, “General-Purpose Tone Decoding and DTMF Detection,” in *Theory, Algorithms, and Implementations, Digital Signal Processing Applications with the TMS320 Family*, Vol. 2, literature number SPRA016, Texas Instruments (1990).
2. J. G. Proakis and D. G. Manolakis, *Digital Signal Processing: Principles*, *Algorithms*, *and Applications*, 2nd ed., Macmillan, New York, NY (1992).
3. G. L. Smith, *Dual-Tone Multifrequency Receiver Using the WE DSP16 Digital Signal Processor*, AT&T Application Note.
4. P. Mock, “Add DTMF Generation and Decoding to DSP-μP Designs,” in *Theory, Algorithms, and Implementations, Digital Signal Processing Applications with the TMS320 Family*, Vol. 1, literature number SPRA012A, Texas Instruments (1989).
5. Y. C. Huang, “DSP Techniques for Telecommunication Systems,” Master’s thesis, Northern Illinois University (August 1995).

APPENDIX/MODIFIED GOERTZEL ALGORITHM SOURCE CODE

```
*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
*   Copyright (C) 1995 Texas Instruments Incorporated. All Rights Reserved
*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
*
* dftgl_mod.s –– This C-callable modified Goertzel algorithm is implemented
*                on one of the C80’s Parallel Processors (PP). It computes 8
*                frequencies and returns the index of the frequency which has
*                maximum energy. The Goertzel algorithm does not require much
*                memory and is optimal when used to compute a small number of
*                frequencies.
*
* Environment:
*   –– execute on a PP (TMS320C80 devices).
*   –– Allocate with versions 1.00 and above of TI’s PP register allocator
*   –– Assemble with versions 1.10 and above of TI’s PP assembler.
*
****************************************************************************
;                     type                description
;

arg1          .set    d1; argument        number of taps of filter
arg2          .set    d2; argument        pointer to input passed by calling
arg3          .set    d3; argument        pointer to coefficients passed by
                        ;                 calling

index         .set    d5; output          flag register

c             .reg    d ; input           cosine register
count         .reg    d ; intermediate    current count register
count1        .reg    d ; intermediate    next count register
mag_square    .reg    d ; intermediate    energy register
max           .reg    d ; intermediate    maximum energy register
s             .reg    d ; input           sine register
sum           .reg    d ; intermediate    sum register
two_c         .reg    d ; input           cosine ×2 register
one           .reg    d ; intermediate    one register
two           .reg    d ; intermediate    two register
```

```
v1            .reg    d ; intermediate    recursive output –1 register
v2            .reg    d ; intermediate    recursive output –2 register
x             .reg    d ; input           input signal register
xfi           .reg    d ; intermediate    imag frequency register
xfr           .reg    d ; intermediate    real frequency register
xfi_2         .reg    d ; intermediate    squared imag frequency register
xfr_2         .reg    d ; intermediate    squared real frequency register
zero_val      .reg    d ; intermediate    zero register

Gx_MAX        .reg    gx; constant
Lx_INDEX      .reg    lx; constant
Lx_COUNT      .reg    lx; constant

Ga_struct_ptr .reg    ga; input           pointer to structure
Ga_x_start    .reg    ga; input           pointer to start of input buffer
Ga_cos        .reg    ga; input           pointer to cosine
La_sin        .reg    la; input           pointer to sine
La_x          .reg    la; input           pointer to input buffer

dum_val       .dummy

      .system $dftgl_mod
      .system _dftgl_mod

      .ptext

      .entry arg1,arg2,arg3,a12,d6,d7,La_x

      .lock a4

$dftgl_mod:
_dftgl_mod:

    *––sp = d6                           ; Push the save–on–entry register onto
                                         ; stack
    *––sp = d7
    *––sp = a12
```

```
loop_setup:
    Ga_x_start = arg2                   ; Point to start of input signal
    Ga_struct_ptr = arg3                ; Point to start of coefficients mem

    le0 = OUT_LOOP_END                  ; Initialize loop registers, and
    lrs0 = 7                            ; do outer loop 8 times
    ls0 = OUT_LOOP_START
    le1 = RECURSIVE_LOOP_END            ; Initialize loop registers, and
    lrs1 = arg1–1                       ; do recursive loop N times
    ls1 = RECURSIVE_LOOP_START          ; N is the number of taps of the
                                        ; filter

    Lx_INDEX = 4                        ; Set up constants for stack storage
    Lx_COUNT = 5
    Gx_MAX = 6
    d0 = SADD                           ; Define EALU operation

    v1 = 0                              ; Initialize state variable to zero
        || La_x = Ga_x_start            ; Set input signal pointer to first
                                        ; element

    two = &*(2)                         ; Set two register = 2
        || *(sp+[Lx_COUNT]) = v1        ; Save zero count on the stack

    v2 = v1 – v1                        ; Initialize state variable
        || c =h *Ga_struct_ptr          ; Load one cosine element
        || x =h *La_x++                 ; Load first input element (16 bit)

    two_c = c * two                     ; two_c = 2 × c
        || *(sp+[Gx_MAX]) = v1          ; Save zero value for MAX on the stack
        || *(sp+[Lx_INDEX]) = v1        ; Save zero index on the stack

OUT_LOOP_START:
RECURSIVE_LOOP_START:
    v1 = v1 * two_c                     ; v1 = v1 ×2 ×c, the result is a Q16
                                        ; number
        || sum = x – v2                 ; sum = x – v2
        || v2 = v1                      ; Update the state variable
```

```
     || x =h *La_x++                ; Load one input element, and then
                                    ; increment the input pointer

RECURSIVE_LOOP_END:
    v1 =ealu(SADD: v1>>8 + sum)     ; Shift v1 back to a Q8 number, and
                                    ; then add the sum
                                    ; i.e. v1 = 2×c×v1 – v2 + x
       || c =h *Ga_struct_ptr       ; Reload current cosine element
       || zero_val = &*(0)          ; Initialize zero_val register

   .cjump RECURSIVE_LOOP_START

    xfr = v1 * c                     ; xfr = v1×c, the result is a Q16
                                     ; number
       || v2 = zero_val – v2         ; Set v2 = – v2
       || one = &*(1)                ; one = 1
       || s =h *++Ga_struct_ptr      ; Increment the struct pointer, and
                                     ; then load one sine element

    xfi = v1 * s                     ; Imag frequency xfi = v1×s
       || xfr =ealu(SADD: xfr>>8 + v2)  ; Shift xfr back to a Q8 number, and
                                     ; then add v2 i.e. xfr = v1×c – v2
       || La_x = Ga_x_start          ; Set input signal pointer to first
                                     ; element
       || count = *(sp+[Lx_COUNT])   ; Load count from stack

    xfr_2 = xfr * xfr                ; Square real frequency, keep it as
                                     ; Q16 number
       || xfi =ealu(SADD: xfi>>8 + zero_val)
                                     ; Shift xfr back to a Q8 number, and
                                     ; then add a zero value in order to
                                     ; use the same d0
       || two = &*(2)                ; Set two register = 2

    xfi_2 = xfi * xfi                ; Square imag frequency
       || count1 = count + one       ; Increment next count register
       || c =h *++Ga_struct_ptr      ; Increment the struct pointer, and
                                     ; then load one cosine element
       || x =h *La_x++               ; Reload first input element (16 bit)
```

```
    mag_square = xfr_2 + xfi_2          ; Energy of the frequency
       || max = *(sp+[Gx_MAX])          ; Load max energy from stack
       || *(sp+[Lx_COUNT]) = count1     ; Store the incremented count back to
                                        ; stack

    two_c = c * two                     ; two_c = 2 × c
       || dum_val = max – mag_square    ; Compare the max energy to the
                                        ; current energy
       || index = *(sp+[Lx_INDEX])      ; Load index value which corresponds
                                        ; to the max energy
       || v2 = &*(0)                    ; Reinitialize the state variable

    index = [lt] count                  ; If the max energy is less than the
       || max = [lt] mag_square         ; current energy, then replace index
                                        ; to the current count and change max
                                        ; to the current energy

OUT_LOOP_END:
    v1 = v2 – v2                        ; Reinitialize the state variable
       || *(sp+[Gx_MAX]) = max          ; Store the max energy and the index
       || *(sp+[Lx_INDEX]) = index      ; back to the stack

    .cjump OUT_LOOP_START

return:
    a12 = *sp++                         ; Pop the save–on–entry register from
                                        ; the stack
    br = iprs
    d7 = *sp++
    d6 = *sp++

    .uexit
```