Digital Filtering with NS32GX320

National Semiconductor
Application Note 695
Zohar Peleg
July 1990

INTRODUCTION

Digital computation of filter transfer functions is a key operation in Digital Signal Processing. The NS32GX320 may be used for digital filtering as well as for other DSP operations, due to its DSP support, consisting of its hardware multiplier and a set of dedicated instructions. This application note describes the realization of FIR and IIR filters, using the NS32GX320. It contains some theoretical overview, practical considerations and NS32GX320 assembly code implementation.

DIGITAL FILTERS

Consider a rational system given by the following transfer function:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{k=1}^{N} a_k z^{-k}} \quad (1)$$

where H(z) is the Z transform of the system's impulse response h(n), X(z) and Y(z) are Z transforms of the input sequence x(n) and output sequence y(n) respectively and a_k and b_k are two sets of coefficients that define the system's behavior. Both the coefficients and the data sequences may be complex.

The input and output of this system satisfy the difference equation

$$y(n) = \sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M} b_k x(n-k) \quad (2)$$

NS32GX320 DSP SUPPORT

A software implementation of (2) is used for digital filtering. The implementation requires a number of multiplications and accumulations for each sample. Two major problems may arise when trying to implement it for a real time application, using a general purpose CPU:

1. Heavy code is required for implementation of each step. That implies a long execution time.
2. Multiply is slow.

If the overall time required for calculating one output point can not meet the input sampling rate of a given application, the filter cannot be used for the application.

The NS32GX320 offers a solution for the implementation of such expressions. The solution is based on the CMACD instruction (Complex Multiply Accumulate).

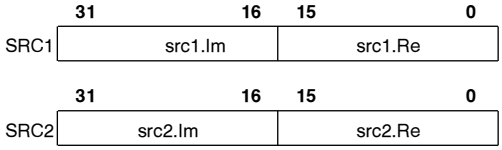


FIGURE 1. Complex Memory Operands

Consider the instruction:

CMACD src1, src2

Where src1 and src2 are 32-bit operands as shown in *Figure 1*. The result of this instruction is

$$\begin{aligned} R0 &\leftarrow R0 + src\,1.Re \times src\,2.Re - src\,1.Im \times src\,2.Im \\ R1 &\leftarrow R1 + src\,1.Re \times src\,2.Im + src\,1.Im \times src\,2.Re \end{aligned} \quad (3)$$

Where R0 and R1 are general purpose registers in the NS32GX320. As shown in (3) the single instruction CMACD performs four 16 × 16 bit multiplications and four additions. The multiplications are performed by a fast hardware multiplier. Note that (3) is the fundamental step of (2). The difference equation of (2) is composed of (M + N) CMACD operations. Due to the advantages of the hardware multiplier and the CMACD instruction, there is reduced amount of code and increased execution speed, that may allow the use of the NS32GX320 for a large variety of real time digital filtering applications.

DATA REPRESENTATION

Both the data sequences and the coefficients are represented by 32-bit complex numbers—16 bits real part in the lower word and 16 bits imaginary part in the upper word. The 16-bit number is represented as a signed fixed point normalized number in the range of −1 : +1. The integer I represents the real number I/32K (I/32768.) The result of multiplying two such numbers is a 31-bit signed fixed point number in the accumulator. The 32-bit integer I in the accumulator represents the real number I/1G (I/1,073,741,824). *Figure 2* shows example of the memory and accumulator representation of some numbers. Translation from accumulator to memory is done by shifting 15 bits to the right. Translation from memory to accumulator is done by sign extension from word to doubleword and then a shift of 15 bits to the left.

Real Number	16-Bit In Memory	32-Bit In Accumulator
−1.0	8000	C0000000
−0.5	C000	E0000000
−0.25	E000	F0000000
0.0	0000	00000000
0.25	2000	10000000
0.5	4000	20000000
~1.0	7FFF	3FFFFFFF

FIGURE 2. Data Representation

Sometimes there are coefficients larger than 1. In such a case all the coefficients' vector is scaled down by 2^l where *l* is the smallest integer that guarantees that all the coefficients become smaller than 1. In some of these cases the data sequence must be scaled down by the same factor, and hence it has to be shifted only 15 − *l* bits when loading the data to the accumulator, or when storing the accumulator's result in the memory.

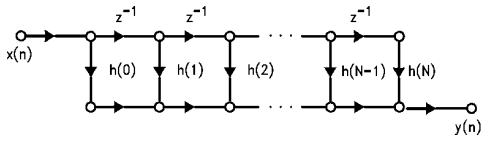
RRD-B30M75/Printed in U. S. A.

FIR FILTER

In FIR (Finite Impulse Response) all the a_k are zero and (2) may be written as:

$$y(n) = \sum_{k=0}^{k=N} h_k x(n-k) \quad (4)$$

The implementation of such a filter is described in *Figure 3*. It is known as direct form FIR.



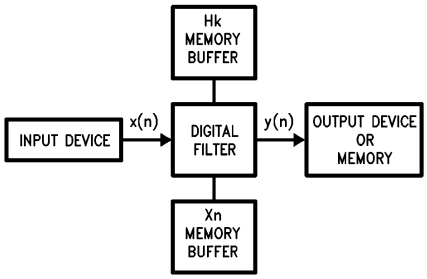
TL/EE/10864–1

FIGURE 3. Flowgraph of Direct-Form FIR

PRACTICAL CONSIDERATIONS

System Configuration

A computation model for software realization of (4) is described in *Figure 4*. When there is a new sample ready in the input device (memory mapped I/O), the digital filter is called either by Jump-Subroutine instruction or by interrupt. The filter reads the new sample, stores it in the data buffer, calculates the new output, and sends it to the distination output device or memory buffer.



TL/EE/10864–2

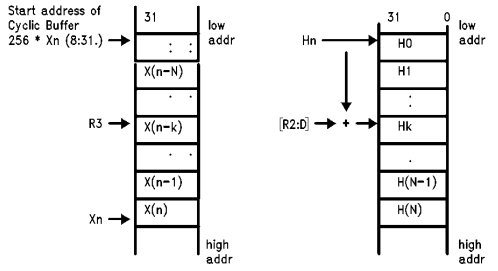
FIGURE 4. System Configuration

DATA ORGANIZATION

In order to be able to calculate the n'th output, in a length-N direct form FIR, there must be a memory buffer that holds the N coefficients, and another memory buffer that holds the recent N samples (x(n−N+1) − x(n)). They will be referred to as Hn and Xn respectively.

The Xn buffer is a problematic one. Since the FIR is built to filter an infinite stream of input samples, the Xn buffer may overflow and run out of the memory space. The pointers to Xn buffer must be handled in a special manner to form a cyclic buffer that can accommodate at least N samples.

A 256 byte cyclic buffer can be easily achieved by advancing a 32-bit pointer, using byte operations. In such a way the 24 MSB will point to the beginning of the buffer and will not be affected by the advancing of the 8 LSB that will point to the desired memory location within the buffer. This solution is applicable for filters up to 64 points. This is sufficient for most practical needs. For larger filters though, the pointer adjustments must be composed of masking the operation with the desired number of bits and then adding it to the buffer's base address. *Figure 5* shows the memory organization of the FIR Implementation. In this specific implementation Xn and Hn are memory locations used as pointers to the current data point and to the coefficients table start address respectively. R3 is the pointer to the point n-k. R2 is the counter k.



TL/EE/10864–3

FIGURE 5. Data Organization for FIR

ASSEMBLY LANGUAGE IMPLEMENTATION

Figure 6 shows an example of FIR routine in NS32GX320 assembly language. It is operated under the following conditions. Xn, Hk, and fir__len are predefined memory locations that are initialized as following:

1. Xn holds the data buffer address of the recent point. (For the first point it is initialized to the beginning of the data buffer.)
2. Hk is initialized to the beginning address of the coefficients table.
3. fir__len is initialized to the desired filter length.
4. in__dev__Re, in__dev__Im are the memory locations of two 16-bit input devices, that hold the value of the next data sample.
5. out__dev__Re and out__dev__Im are the memory locations of two output devices which are the destination of the processed data.

```
FIR:
addb    $4,           Xn              # adjust pointer to next point
movd    Xn,           r3              # in the cyclic buffer
movw    in_dev_Re,    0(r3)           # store new point
movw    in_dev_Im,    2(r3)           # in buffer.
movqd   $0,           r0              # Zero accumulator.
movqd   $0,           rl
movqd   $0,           r2              # k ← 0
cnvl:
cmacd   0 (r3),       Hk [r2:d]       # acc ← acc + x (n-k)*h(k)
cmpd    fir_len,      r2              # is k=N ?
beq     out
addqd   $l,           r2              # increment k
subb    $4,           r3              # adjust pointer to x(n-k)
br      cnvl
out:
ashd    $-l5,         r0              # Normalize result
ashd    $-l5,         rl
movw    r0,           out_dev_R       # Send result to
movw    rl,           out_dev_I       # output device
reti
```

FIGURE 6. NS32GX320 Code for Direct Form FIR

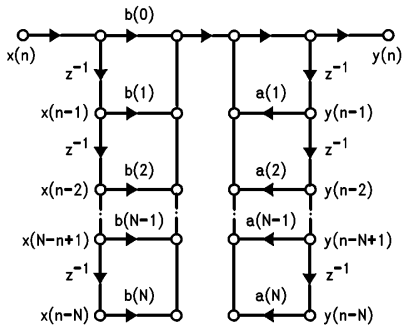
Note: If the FIR length is a small predefined number N, the loop ''cnvl:'' may be replaced by N consecutive CMACD instructions. That will save the loop overhead and improve performance.

IIR FILTER

In IIR filters, at least 1 of the a(k) in (2) is nonzero. Assuming N = M (may be achieved by adding |N − M| zero coefficients to the short expression), (2) becomes:

$$y(n) = \sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{N} b_k x(n-k) \qquad (5)$$

The direct form realization of (5) is described in *Figure 7*.

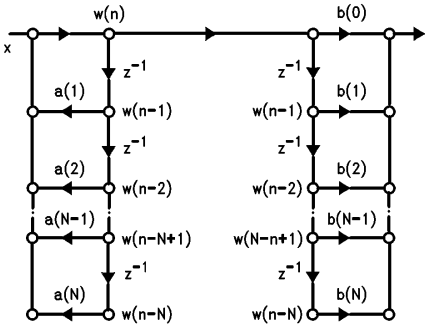


TL/EE/10864–4

FIGURE 7. Direct Form I IIR Filter

The flowgraph of *Figure 7* may be implemented using the same concept as in the FIR implementation. A straight forward implementation would require to maintain buffers of both the recent N inputs and recent N outputs.

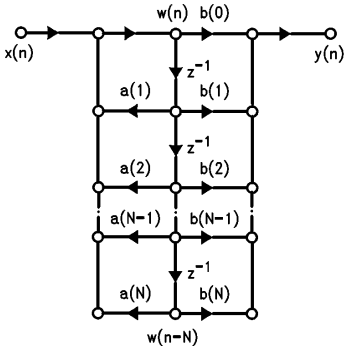
Figure 7 may be viewed as a cascade of two networks. The first one with the b_k corresponds to the numerator of (1), and the second one, with the a_k corresponds to the denominator of (1). In linear shift-invariant systems the order of cascading subsystems may be reversed without changing the input-output relation. The result of reversing the cascading order is shown in *Figure 7*.



TL/EE/10864–5

FIGURE 8. Reversed Cascading Order of *Figure 7*

Figure 8 may be redrawn as shown in *Figure 9*, by combining the two identical delay strings. The resulting network is known as Direct Form II.



TL/EE/10864–6

FIGURE 9. Direct Form II IIR

The difference equation for this network is

$$w(n) = x(n) + \sum_{k=1}^{N} a_k w(n-k)$$
$$y(n) = \sum_{k=0}^{N} b_k w(n-k) \quad (6)$$

w (n) is a state signal inside the system. This is the only delayed signal in that system. Therefore it is preferred to implement the IIR using Direct Form II, since it requires only one delay buffer—Wn rather than two buffers Xn and Yn as required in Direct Form I.

SYSTEM CONFIGURATION

Consider a system similar to the one described for the FIR and shown in *Figure 4*. A Wn buffer is organized the same way as the Xn buffer in *Figure 5*. Ak and Bk buffers are organized like the Hk buffer in *Figure 5*.

IMPLEMENTATION

A routine similar to the FIR is executed on the recent N−1 Wn to calculate the new Wn and stores it in the Wn buffer. Another FIR is then executed on the N recent Wn to calculate the new y(n). If the Ak buffer was scaled down by 2^{scale} then x(n) must be scaled down by the same factor when loaded to the accumulator. This scale down is compensated when storing the accumulator in Wn buffer. In such a case the scale constant assignment must be changed accordingly.

```
.set    scale,            0               # The number of shifts that
                                          # were used to scale down Ak.
IIR:
addb    $4,               Wn              # adjust pointer to next point
movd    Wn,               r3              # load pointer to Wn buf.
movw    in_dev_Re,        r0              # acc.Re  ←  x(n) .Re
movw    in_dev_Im,        rl              # acc.Im  ←  x(n) .Im
movxwd  r0,               r0              # extend sign bit and
movxwd  rl,               rl              # shift right 15 bits
ashd    $15-scale,        r0              # to translate data to
ashd    $15-scale,        rl              # accumulator representation
movqd   $1,               r2              # k  ←  1
A_loop:
cmacd   0(r3),            Ak[r2:d]        # acc  ←  acc + w (n-k) * a(k)
cmpd    irr_len,          r2              # is k = N ?
beq     out_A
addqd   $1,               r2              # increment k
subb    $4,               r3              # adjust pointer to w(n-k)
br      A_loop
out_A:
ashd    $-15+scale,       r0              # normalize the new w(n)
ashd    $-15+scale,       rl
movd    Wn,               r3              # reload pointer to w(n)
movw    r0,               0(r3)           # Store Real part of Wn
movw    rl,               2(r3)           # Store Im part of Wn
#
# The rest is FIR of Bk
# coefficients, on the W buffer.
#
movqd   $0,               r0              # Zero accumulator.
movqd   $0,               rl
movqd   $0,               r2              # R2  ←  0
B_loop:
cmacd   0(r3),            Bk[r2:d]        # acc  ←  acc + b(k) * x(n-k)
cmpd    iir_len,          r2
beq     out_B
addqd   $1,               r2              # k  ←  k + 1
subb    $4,               r3              # adjust pointer to w(n-k)
br      B_loop
out_B:
ashd    $-15,             r0              # normalize result
ashd    $-15,             rl
movw    r0,               out_dev_Re      # send result to
movw    rl,               out_dev_Im      # its destination
reti
```

FIGURE 10. NS32GX320 Code of Direct Form II IIR

SECOND-ORDER DIRECT FORM IIR

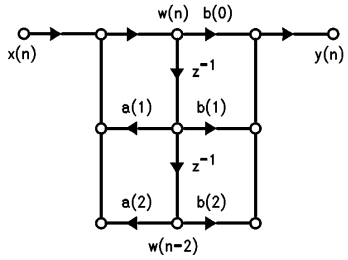
Every direct form IIR of order N may be implemented as a cascade of N/2 order-2 direct form II sections. Furthermore, in many applications the order-2 IIR is sufficient. There are some practical advantages in realizing low order IIR with pre-defined length:

1. The loops of CMACD iterations may be unrolled to save the loop overhead.
2. A small number of delayed elements may be shifted along the data buffer to save the overhead of managing a cyclic buffer.

For 2nd order IIR, (6) may be rewritten as

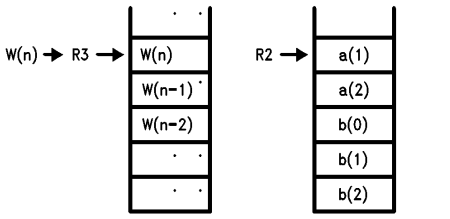
$$w(n) = x(n) + a(1)w(n-1) + a(2)w(n-2)$$
$$y(n) = b(0)w(n) + b(1)w(n-1) + b(2)w(n-2) \quad (7)$$

The realization of (7) is illustrated in *Figure 11*. *Figure 12* shows the data organization, and *Figure 13* shows the NS32GX320 assembly code implementation for the order-2 direct form II IIR.



TL/EE/10864–7

FIGURE 11. Order 2 Direct Form II IIR



TL/EE/10864–8

FIGURE 12. Data Organization for Order-2 IIR

REFERENCES

1. A.V. Oppenheim and R.W. Schafer, *Digital Signal Processing*, Prentice-Hall, 1975.
2. L.R. Rabiner and B. Gold, *Theory and Applications of Digital Signal Processing*, Prentice-Hall, 1975.

```
.set    scale          0              # must be changed to the
                                      # actual number of shifts
                                      # that were used to scale down
                                      # the Ak coefficients.
ORDER_2_IIR:
movd    Wn,            r3             #r3 ← Wn pointer.
movd    AkBk,          r2             #r3 ← coefficients table pointer
movw    in_dev_Re,     r0             # store new point in Xn
movw    in_dev_Im,     rl             # store new point in Xn
movxwd  r0,            r0             # n'th sample is de-normalized
movxwd  rl,            rl             # to the accumulator's
ashd    $15-scale,     r0             # representation -
ashd    $15-scale,     rl             # acc = Xn
amacd   4(r3),         0(r2)          # acc ← acc + W(n-1)*A(1)
cmacd   8(r3),         4(r2)          # acc ← acc + W(n-2)*A(2)
nop
nop                                   # wait till acc is ready
ashd    $-15+scale,    r0             # normalize acc
ashd    $-15+scale,    rl
movw    r0,            0(r3)          # W(n) .Re ← acc.Re
movw    rl,            2(r3)          # W(n) .Im ← acc.Im
movd    $0,            r0             # acc.Re ← 0
movd    $0,            rl             # acc.Im ← 0
cmacd   0(r3),         8(r2)          # acc ← acc + W(n)*B(0)
cmacd   4(r3),         12(r2)         # acc ← acc + W(n-1)*B(1)
cmacd   8(r3),         16(r2)         # acc ← acc + W(n-2)*B(2)
nop
nop                                   # wait till acc is ready
ashd    $-15,          r0             # normalize Y(n) .Re
ashd    $-15,          rl             # normalize Y(n) .Im
movw    r0,            out_dev_Re     # send Y(n) to its destination.
movw    rl,            out_dev_Im
movd    4(r3),         8(r3)          # W (n-2) ← W(n-1)
movd    0(r3),         4(r3)          # W (n-1) ← W(n)
reti
```

FIGURE 13. Order-2 Direct Form II IIR

LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

National Semiconductor Corporation
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-8090
Tel: 1(800) 272-9959
TWX: (910) 339-9240

National Semiconductor GmbH
Livry-Gargan-Str. 10
D-82256 Fürstenfeldbruck
Germany
Tel: (81-41) 35-0
Telex: 527649
Fax: (81-41) 35-1

National Semiconductor Japan Ltd.
Sumitomo Chemical
Engineering Center
Bldg. 7F
1-7-1, Nakase, Mihama-Ku
Chiba-City,
Ciba Prefecture 261
Tel: (043) 299-2300
Fax: (043) 299-2500

National Semiconductor Hong Kong Ltd.
13th Floor, Straight Block,
Ocean Centre, 5 Canton Rd.
Tsimshatsui, Kowloon
Hong Kong
Tel: (852) 2737-1600
Fax: (852) 2736-9960

National Semiconductores Do Brazil Ltda.
Rue Deputado Lacorda Franco
120-3A
Sao Paulo-SP
Brazil 05418-000
Tel: (55-11) 212-5066
Telex: 391-1131931 NSBR BR
Fax: (55-11) 212-1181

National Semiconductor (Australia) Pty, Ltd.
Building 16
Business Park Drive
Monash Business Park
Nottinghill, Melbourne
Victoria 3168 Australia
Tel: (3) 558-9999
Fax: (3) 558-9998