AN1442
APPLICATION NOTE

Signal Processing with ST10-DSP

By André ROGER and Charles AUBENAS

By providing ST10 micro-controllers with an enhanced 16-bit DSP co-processor STMicroelectronics proposes a powerful solution to a wide range of applications. The ST10 micro-controllers including a DSP unit are identified with a "2" in the second digit of the variant code like ST10F269.

Most of the algorithms encountered in signal processing such as data acquisition (average, MAX/MIN), signal processing control (PID, PD) and filtering (FIR, IIR) can be easily run with the ST10-DSP micro-controllers. So the ST10-DSP, combined to the well know real-time performances of the ST10-MCU core, efficiently supports demanding applications like electronic steering, suspension, engine control, airbag among a varied list.

The goal of this application note is to precise ST10-DSP features and tools. It describes how to use and program the ST10-DSP co-processor for common signal processing algorithms with a set of dedicated examples.

Additional documentation :

– ST10F2xx product data sheets

– ST10 family programming manual

– ST10F269 user's manual

– Various application notes

Figure 1 : ST10-DSP Architecture

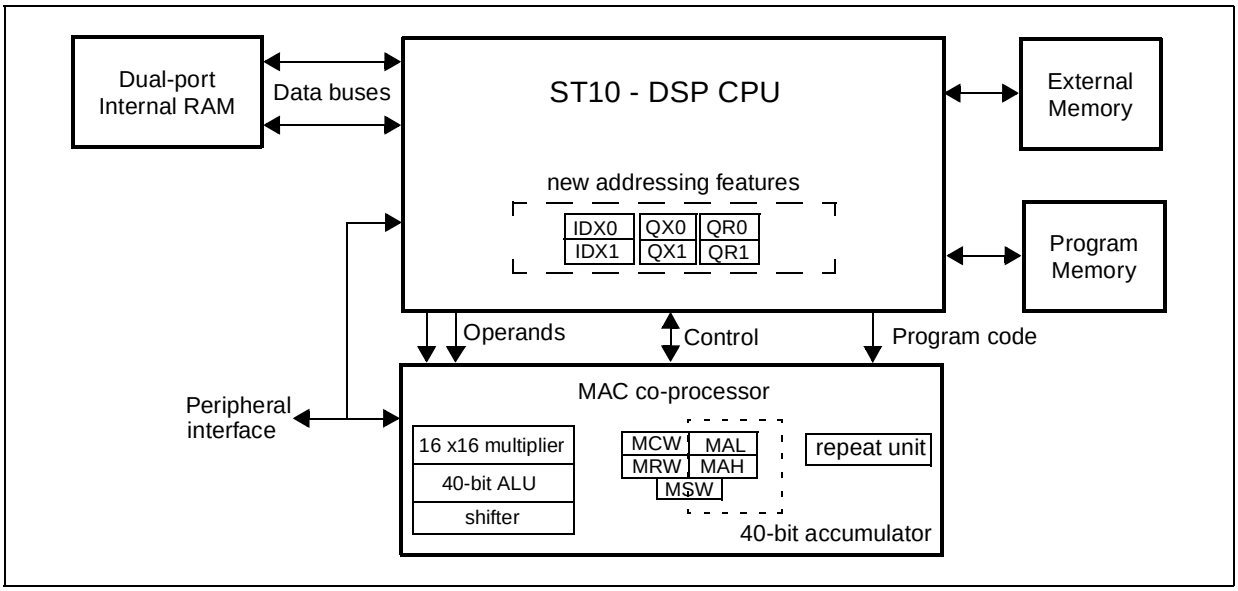


TABLE OF CONTENTS PAGE

1 - INTRODUCTION

To better take advantage of the ST10F2XX DSP capabilities, information and software examples are gathered in this application note.

The second chapter reviews the features and the key points of the different parts of the ST10-DSP.

The third chapter details the ways to develop DSP functions using the ST10 programming tools. This is completed with some hints related to the tools and to the ST10-DSP.

Chapter four highlights the ST10F2XX DSP advantages and strengths.

To complete the application note, various programming examples like Matrix multiply, FIR & LMS routines and operations on tables are provided in the chapter five.

2 - ST10F2XX DSP FEATURES

ST10F2xx is a combined CPU and DSP :

– As a CPU, it is a powerfull real time oriented 16-bit CPU,

– As a DSP, it is a single MAC 16 by 16-bit multiplier with a 40-bit accumulator.

The key features of the ST10-DSP are explained in the following sections. For details on the ST10 instruction set and related information refer to the "ST10 FAMILY PROGRAMMING MANUAL", especially in the chapter 3 - "MAC INSTRUCTION SET".

2.1 - Harvard Architecture

ST10-DSP is an Harvard architecture implementation. On every CPU-cycle it allows :

– 1 opcode fecth,

– 2 operand reads,

– 1 optional operand write.

On the other hand the ST10 core is based on unified memory organization, code and data share the same linear addressing space.

2.2 - Multiply and Accumulate

ST10-DSP supports different multiply and accumulate instructions with several addressing modes.

With the CoMAC [IDXy+], [Rz+], in a single cycle, ST10 is :

– Fetching 2 operands addressed by IDXy and Rz,

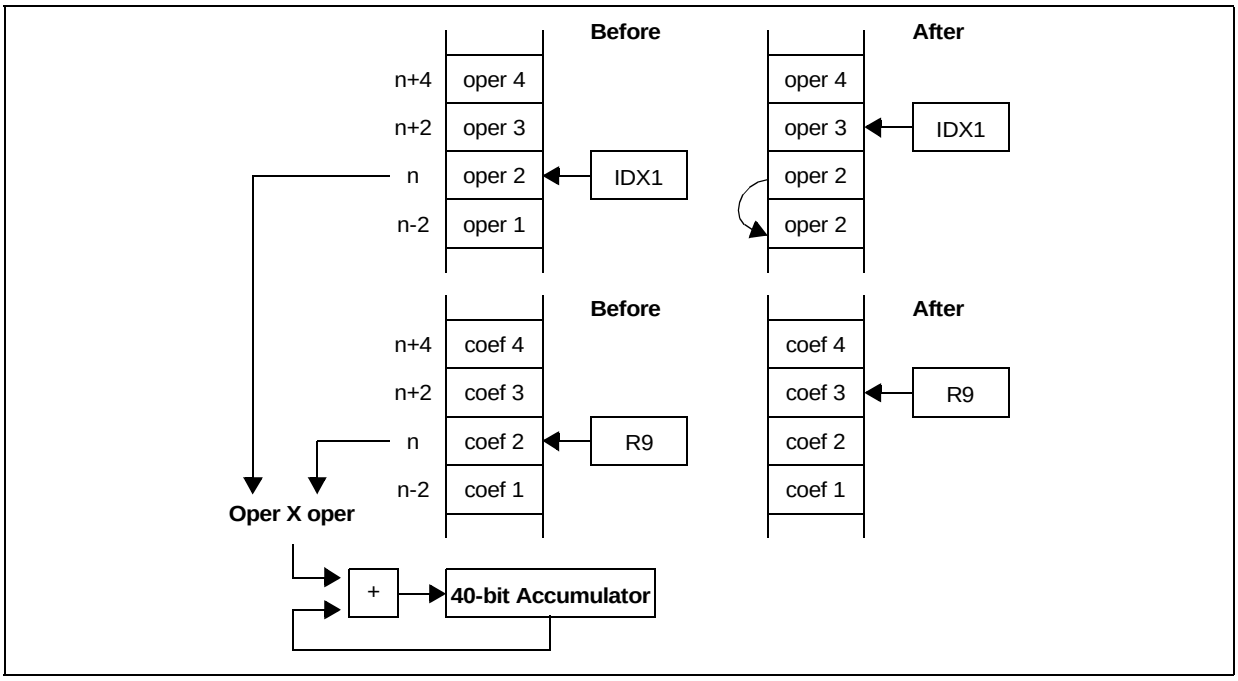
– Updating 2 pointers (increment, decrement, add an index).

A Multiply and Substract instruction ("CoMAC-") is also available with the same addressing modes as the Multiply and Accumulate.

CoMacM is similar to CoMac except that ST10 also move 1 operand to the bottom of the table (usefull for **circular buffers** in data acquisition routines) :

The following figure illustrates the behavior of the CoMACM [IDX1+], [R9+] instruction.

Figure 2 : Operand parallel move with CoMACM instruction



2.3 - Minimum and Maximum

ST10-DSP has 2 instructions for detection of minimum and maximum (CoMin and CoMax). These instructions are generally used for saturating arithmetic.

Hint : Combined with auto-incrementation, these instructions allow the ST10 to scan a table of samples and detect either the minimum or the maximum in one CPU-cycle per sample.

2.4 - Instruction Repeat Unit

Each instruction can be repeated either an fixed number of times (immediate value) or a variable number of times.

Register MRW is used to repeat a variable number of times.

Repeat sequences can be interrupted.

This allows ST10-DSP to compute a FIR (16 by 16; real) at a rate of 1 TAP per CPU-cycle.

2.5 - Barrel Shifter

ST10-DSP has a barrel shifter connected to the accumulator : any result or value loaded into the accumulator can be shifted. The shift value must be between 0 and 8 (included).

2.6 - DSP Addressing Modes

New addressing modes and their associated registers have been added : 2 index pointers and 4 offset registers.

2.6.1 - Index Registers

The new index registers (IDX0 and IDX1) are giving more flexibilities in operand addressing : they allow double indirect addressing.

When ST10-DSP is addressing 2 operands, 1 data-pointer shall be from the register bank (R0 through R15) and the other pointer must be from the index registers.

All pointer post modifications available with GPR addressing are available with Index-addressing (see below).

Restriction : The use of IDX index registers is limited to internal DPRAM addressing except for the CoMOV instruction.

2.6.2 - Offset Registers

4 Offset registers (QX0, QX1, QR0, QR1) have been added to enhance ST10 capability in handling matrix. They are usefull when doing computation on matrix lines and/or columns.

As a result, 5 different pointer modifications are available with ST10-DSP :

– No pointer modification : ex : [IDX0], [R1]

– Pointer post incrementation : ex : [IDX1+], [R2+]

– Pointer post decrementation : ex : [IDX0-], [R3-]

– Pointer post incrementation by offset : ex : [IDX0 + QX0], [R1 + QR0]

– Pointer post decrementation by offset : ex : [IDX1 - QX0], [IDX1 - QX1], [R2 - QR1]

As explained above, pointer post-modification are done in parallel to the execution of the instruction.

2.7 - Real time Aspects

The ST10-DSP is both a real time CPU and a DSP. Any DSP code developped for ST10-DSP can be interrupted at any time (including during repeat sequences) and execution resumed after the interrupt routine. During the interrupt, bit MR remains set to indicate that a repeated instruction has been interrupted.

– **Latency :** there is no added latency on interrupts when DSP functions are used.

Interrupt routine requirements : The MAC registers must be saved at the entry point and restored at the exit of an interrupt routine using the DSP function and interrupting a DSP function. This is the single constraint in using ST10-DSP with interrupt. This control can be automatically done by Tasking tool chain by using **"#pragma savemac"** on each task using DSP functions (for details, refer to Tasking user's manual).

2.8 - ST10 intrinsic Benchmarks

The following table is showing ST10 instrinsic benchmarks in DSP algorithms.

Table 1 : ST10 Intrinsic Benchmarks in DSP Algorithms

		Instruction Cycles	Program Words
Mathematics	32 by 32 signed multiplication	12	24
	Nth Order Power Series	9N/2+3	22
	[NxN][Nx1] Matrix Multiply	N^2+5N+5	24
	N-Real Multiply (Windowing)	2N+3N/URF[1)]	5+4.URF
DSP Routines [2)]	32x16 L-tap FIR	2L+3	18
	DF1 [3)] Nth Order IIR filter	2N	10
	DF2 [4)] Nth Order IIR filter	2N+1	12
	DF2 N-cascaded Biquads	11N-1	19
	TF [5)] N-cascaded Biquads	14N-15	24
	16x16 L-tap LMS	4L+2(L-2)/URF +1	51+2(URF-1)
	32x16 L-tap LMS	6L+2(L-2)/URF +20	71+2(URF-1)
Operations on Tables	Table Move (L items)	L+3	8
	Find the Index of a Maximum Value in a table (L items)	3L/2+10 [6)]	21
	"Compare For Search" [7)] (L items)	L/2+7 [8)]	13

Notes: 1. "URF" stands for "UnRolling Factor".
2. Representative part of the routine only.
3. Direct Form 1.
4. Direct Form 2.
5. Transpose Form.
6. On average.
7. First data in a table that matches a specified condition.
8. On average.

3 - DEVELOPPING DSP FUNCTIONS

This chapter is explaining how to develop DSP functions with ST10F2xx. ST10-DSP programming aspects and also tool chain aspects are covered by this chapter.

3.1 - Developping Your Own DSP Functions

With Tasking tool chain, there are 3 different ways of developping an DSP function.

3.1.1 - DSP Functions in Full Assembly

New functions can be developped in full assembly. They need to be compatible with the EABI defined by Tasking (see Tasking documentation). The ST10-DSP library has been developped in this way. Then, it can be decided to make a library of the new DSP functions or to include the assembly sources in the Tasking project.

3.1.2 - DSP Functions in Mixed "C" and Assembly

Tasking pseudo registers can be used to develop user's dedicated DSP functions.

This is the recommended way of developping new DSP functions for ST10.

This way of programming is :

– Ideal for short functions (typically a loop of 1 or 2 instructions).

– More portable than full assembly functions : register allocation is done by the compiler depending on available registers, the code is independant of the memory model.

Such a function can also be advantageously inlined by using the keyword "_inline".

3.1.2.1 - Example of Mixed "C" and Assembly for a Data Acquisition Routine

The following example detect the maximum value within a collection of samples (16-bit unsigned format) within a table.

```
_inline int max_table( *int a, int nb_samp)
{
   int retval;
#pragma asm( @w1=*a, @w2=nb_samp  )
   EXTERN XVAL:WORD, BVAL:BYTE, YVAL:WORD
   MOV @1, @w1   ; initialise register to point to the beginning of the table
   MOV @2, @w2   ; temporary register
   MOV MAL,#0    ; initialise accumulator to the minimum possible value
                 ; (table of unsigned int assumed)
   MOV MAH,#0
   MOV MAE ,#0
   MOV MRW, @2   ; load repeat counter with nb_samples
   MOV @2, #0
   LOOP1 :
   Repeat MRW times CoMAX  @2, [@1+]  ; perform comparison MRW times.
   MOV @w1, MAH
#pragma endasm( retval=@w1 )
return retval;
}
```

3.1.3 - User Defined Intrinsics

Tasking tool chain allows to create user's defined intrinsics. For details please refer to Tasking documentation (m_c166.pdf).

3.1.4 - Tasking Support of ST10-DSP

Tasking is developping new intrinsics made specifically to take advantage of ST10-DSP. These intrinsics are implemented starting from tool chain V7.0 r5.

3.2 - ST10-DSP hints

3.2.1 - Instruction Scheduling

All ST10-DSP instructions are executed in 1 Instruction cycle (2 CPU clock cycles) without any latency.

There is no automatic interlock between the DSP units and the CPU part. As a consequence, 1 dummy instruction shall be inserted between :

– 1 instruction doing pointer initialisation (IDX0/1, QX0/1 and QR0/1) and a DSP instruction using it,

– 1 instruction in the DSP (CoXXX) and a CPU instruction that reads from the DSP : ex : between CoMAC and a compare instruction on MCW.

These are the only constraints in ST10-DSP programming.

3.2.2 - DSP Loops

ST10 repeat unit is limited to 1 instruction. Bigger loops are done using loop counter mapped in an ST10 GPR and using the instruction "CMPD1 reg, #0 together with JMPR cc_NE, Loop-label".

Hint : usually, it is better to unroll loops to get better performance as long only 1 accumulator is needed.

3.2.3 - Memory Mapping

ST10-DSP can read 2 operands from the memory provided they are properly mapped.

The "X and Y" memories of ST10 is the DualPort RAM (DPRAM) : ST10 internal DPRAM has 2 independant read/write ports which allow parallel read and write operation without delay.

=> all operands and coefficients shall be mapped in ST10 's DPRAM.

Hint : when ST10-DSP functionality is used, it is recommended to limit DPRAM usage to Stack, register banks for context switches, variables in bit format (bit addressable area is in the DPRAM), and variables for the DSP unit.

3.2.4 - Enhanced 32-bit Arithmetic with ST10-DSP Unit

The concatenation unit enables the MAC unit to perform 32-bit operation in 1 CPU-cycle. It concatenates two 16-bit operands to a 32-bit operand before the 32-bit operation is executed in the 40-bit adder/ subtracter. The second operand is always the current accumulator content.

This feature can be used for specific 32-bit arithmetic :

– 32-bit arithmetic on tables (sum, comparison)

– custom arithmetic : multiplication followed immediatly by few extented precision corrections (ex : addition, subtraction, shift) before storing the result.

3.2.5 - Development Tools

Code Debugging

The Algorithms developped for ST10-DSP can be debugged :

– without any hardware : Tasking Crossview functional simulator,

– with an evaluation board : ex : FORTH ST10F269 evaluation board,

– with an emulator.

Code Optimisation

The best tool for code optimisation is the emulator, it allows to :

– trace each instruction, the addresses and the values of the operands,

– time stamp each instruction, to verify the performance of the routine.

4 - ARCHITECTURAL ADVANTAGES OF ST10F2XX AS A DSP/MCU

The efficiency of a DSP micro-controller depends several key features, all fully supported by the ST10-DSPs :

– A hardware multiply and accumulate unit with a 40-bit accumulator (8 guard bits).
– A DSP's Harvard architecture allowing simultaneous program instruction, 2 operand fetches and 1 operand write per cycle.
– An address generation unit providing with zero-overhead address update for the 2 indirect pointers to be used for the next processor cycle. This unit supports double indirect addressing with 2 different offset registers for efficient matrix addressing.
– On-chip memory architecture of ST10F2xx provide zero penalty accesses to program and data, resulting in high throughput even in the automotive temperature range (see relevant ST10F2xx product datasheets).
– A choice of proven automotive software components like CAN-drivers, OSEK operating systems.
– On-chip DMA (8 channels) to handle automated and sophisticated acquisitions (time based, position based) without software overhead.
– A powerfull interrupt controller handling up to 64 interrupt sources with 16 different interrupt levels.
– A proven debugging environment based on real time emulation, featuring real time trace of DSP/MCU registers.

4.1 - Summary

ST10F2xx provides a solid platform for motor control and other control applications in real time environment. This good balancing of DSP and MCU features allows ST10F2xx to have similar performance in real applications (interrupts, scaling of results, more variables to handle like calibration variables) than in benchmarks.

ST10-DSP preferred algorithms are data acquisition (average, MAX/Min), signal processing oriented control (PID, PD) and filtering (FIR, IIR).

For automotive applications, ST10F2xx has superior advantages compared to other DSP/MCUs : availability in the entire automotive temperature range, wide choice of OSEK kernels + CAN certified drivers and communication stack (TP, NM) are ST10-DSP differentiating factors.

5 - ST10-DSP PROGRAMMING EXAMPLES

5.1 - Initialization

This routine shows an example of initialisation of DSP registers :

```
; Control Registers Initialization.
MOV     MCW,    #mcw        ; (MCW) ← mcw.
MOV     MRW,    #mrw        ; (MRW) ← mrw.
;
; Accumulator Initialization.
;
MOV     MAH,    #data16     ; (MAH) ← #data16,
                            ; (MAE) ← 8 times (MAH.15 ),
                            ; (MAL) ← 0000h .
;
; Core SFRs Initialization.
;
MOV     IDX0,   #idx0       ; (IDX0) ← idx0.
MOV     IDX1,   #idx1       ; (IDX1) ← idx1.
EXTR    #4                  ; Next 4 instructions will utilize the ESFR space.
MOV     QX0,    #qx0        ; (QX0) ← qx0.
MOV     QX1,    #qx1        ; (QX1) ← qx1.
MOV     QR0,    #qr0        ; (QR0) ← qr0.
MOV     QR1,    #qr1        ; (QR1) ← qr1.
```

	Instruction Cycles	Program Words
Total	10	19

ST10-DSP registers are handled like SFRs : all addressing modes available for SFRs can be used to initialise ST10-DSP : immediate (see above) , register, memory.

Reminder : after initialisation of DSP registers, a dummy instruction shall be insterted to take into account pipeline effect.

5.2 - Mathematics

5.2.1 - Double Precision Multiplication

This routine assumes that:

– X_L (LSW) and X_H (MSW) are stored in R0 and R1, respectively.

– Y_L (LSW) and Y_H (MSW) are stored in R2 and R3, respectively.

– MP and MS are cleared.

– t performs P=X*Y.

After computation, the 64-bit product P is stored in R4-R7, where R7 contains the most significant word and R4 the least significant word.

```
;
; XL *YL multiplication (unsigned)
;
CoMULu          R0,         R2          ; (ACC) ← XL *YL .
CoSTORE         R4,         MAL         ; (R4) ← (ACC)L .
;
; XL *YH multiplication (unsigned/signed) and X H *Y L multiplication (signed/unsigned).
;
CoSHR           #8                      ; (ACC) ← (ACC) >> 8.
CoSHR           #8                      ; (ACC) ← (ACC) >> 8.
CoMACus         R0,         R3          ; (ACC) ← (ACC) + XL *YH .
CoMACsu         R1,         R2          ; (ACC) ← (ACC) + XH *YL .
CoSTORE         R5,         MAL         ; (R5) ← (ACC)L .
;
; XHL *YH multiplication (signed/signed)
;
CoASHR          #8                      ; (ACC) ← (ACC) >>a 8.
CoASHR          #8                      ; (ACC) ← (ACC) >>a 8.
CoMAC           R1,         R3          ; (ACC) ← (ACC) + XH *YH .
CoSTORE         R6,         MAL         ; (R6) ← (ACC)L .
CoSTORE         R7,         MAH         ; (R7) ← (ACC)H .
```

	Instruction Cycles	Program Words
Total	12	24

5.2.2 - Nth Order Power Series

The formula is:

$$y = \sum_{i=0}^{n} a(i) \times x^{i} = [[[[a(n) \times x + a(n-1)] \times x + a(n-2)] \times x + a(n-3)] + \ldots]$$

The associated pseudo code is:

```
; x = input.
; y = output.
; a(i) for i=0,1,...,n, are the coefficients.
y = a(n);
for (i=1 to n)    {
                  y = y*x+a(n-i);
                  }
```

Assuming that:

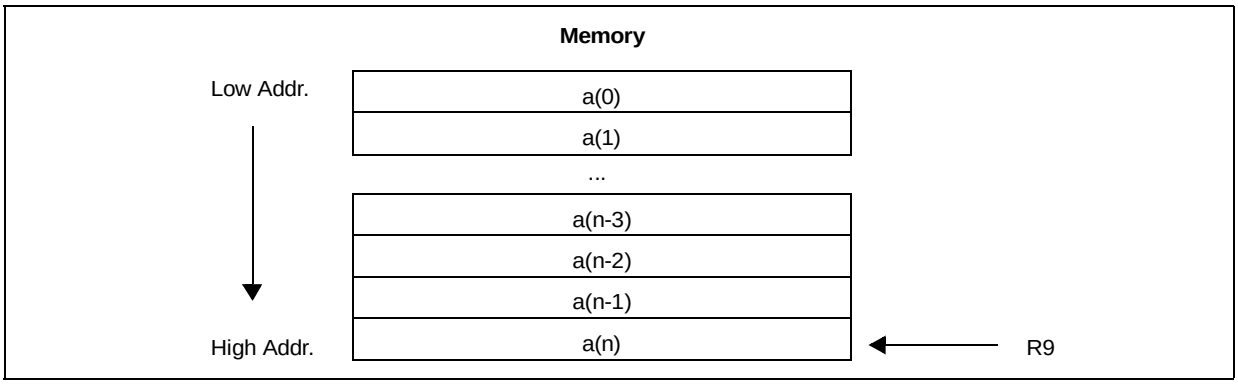
– x is a fractional and is located in R1.

– a(i) are fractional.

– y can be represented by a 16-bit data.

The final result is contained in the accumulator (ACC).To minimize the loop overhead, the program uses “loop unrolling” and assumes that n is even.

Figure 3 : Memory Map of a N^{th} Order Power Serie Processing

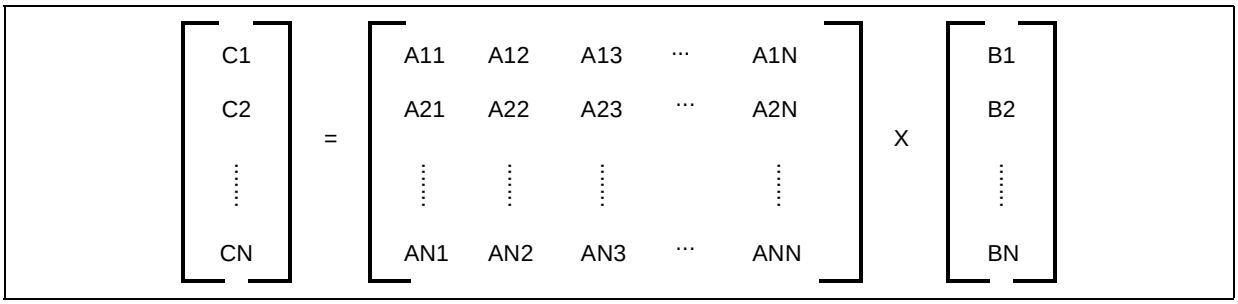



```
                ; Initialization.
                ;
                MOV         MCW,        #mcw            ; (MCW) ← mcw, MS and MP are cleared.
                MOV         R0,         #0              ; (R0) ← 0
                MOV         R9,         a(n)_address    ; (R9) ← address of a(n).
                ;
                ; Initialize the Loop Count
                ;
                MOV         R3          #n/2             ; (R3) ← n/2.
                ;
                ; Unrolled Loop
                ;
SERIE_LOOP:     CoMUL       R1,         [R9-]           ; (ACC) ← a(n)*x;
                                                        ; (R9) ← (R9)-2
                CoADD       R0,         [R9-]           ; (ACC) ← (ACC) + a(i-1);
                                                        ; (R9) ← (R9)-2.
                CoSTORE     R2,         MAS             ; (R2) ← limited (ACC)
                CoMUL       R1,         R2              ; (ACC) ← (R2)*x
                                                        ; (R9) ← (R9)-2
                CoADD       R0,         [R9-]           ; (ACC) ← (ACC)+ a(i-2);
                                                        ; (R9) ← (R9)-2.
                CoSTORE     R2,         MAS             ; (R2) ← limited (ACC)
                ;
                ; End_of_loop Checking.
                ;
                CMPD1       R3          #0h             ; (R3) ← (R3)-1.
                JMPR        cc_NZ       SERIE_LOOP      ; End-of-Loop test & branch.
                ;
```

	Instruction Cycles	Program Words
Total	9N/2+3	22

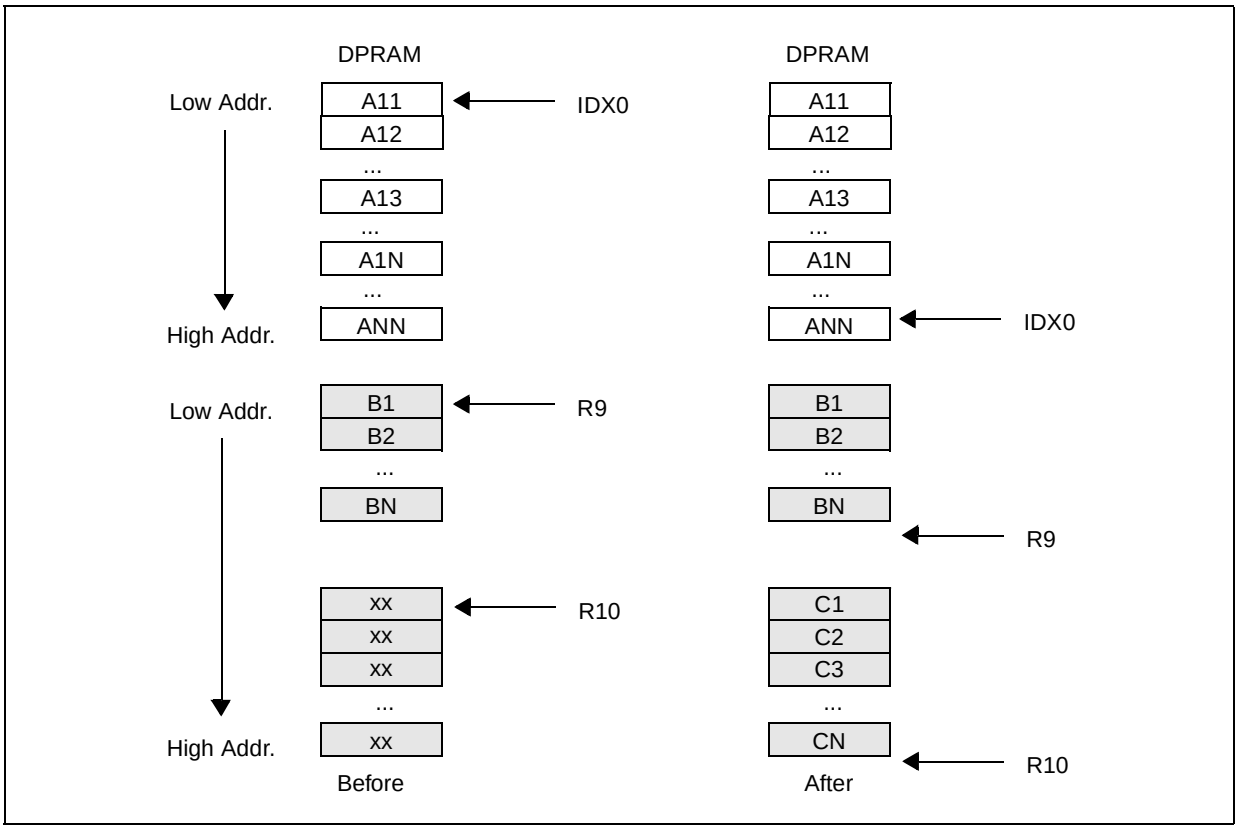
5.2.3 - [NxN][Nx1] Matrix Multiply

Figure 4 : [NxN][Nx1] Matrix Multiply



The [NxN][Nx1] matrix multiply memory map is shown below:

Figure 5 : Memory Map of the Matrix Multiply



N is assumed to be less than 31.

```
                ;
                ; MAC dedicated registers' initialization:
                ;
                MOV         IDX0,       @A11        ; (IDX0) ← A11_addr.
                EXTR        #1                      ; next instruction  uses  ESFR space
                MOV         QR0,        #2*(N-1)    ; (QR0) ← 2(N-1): N-1 words.
                ;
                ; GPRs initialization:
                ;           - R7 is used as loop counter.
                ;           - R9 contains B1 Address.
                ;           - R10 contains C1 Address.
                ;
                MOV         R7,         #N          ; (R7) ← N
                MOV         R9,         @B1         ; (R9) ← B1_addr
                MOV         R10,        @C1         ; (R10)) ← C1_addr

                ;
                ; Product prolog
                ;
MATRIX_LOOP     CoMUL       [IDX0+],    [R9+]       ; (ACC) ← Ai1.B1
                                                    ; (IDX0) ← (IDX0)+2
                                                    ; (R9) ← (R9)+2.
                ;
                ; Product loop.
                ;
REPEAT N-2 TIMES CoMAC      [IDX0+],    [R9+]       ; (ACC) ← (ACC) + Aij*Bj
                                                    ; (IDX0) ← (IDX0)+2
                                                    ; (R9) ← (R9)+2.
                ;
                ; Product epilog (provide Ci in an appropriate format).
                ;
                CoMAC       [IDX0+],    [R9-QR0]    ; (ACC) ← (ACC) + AiN*Bn
                                                    ; (IDX0) ← (IDX0)+2
                                                    ; (R9) ← (R9)-(N-1).
                ;
                ; Shift & Rounding: shift to put the result in the right fractionnal format if needed
                ;
                CoASHR      #data3,     rnd         ; (ACC)=(ACC)>>#data3+rnd
                ;
```

```
; Write Ci into memory.
;
CoSTORE     [R10+]      MAS          ; ((R10)) ← Ci.
                                     ; (R10) ← (R10)+2.
;
; End_of_loop Checking.
;
CMPD1       R7          #0h          ; (R7) ← (R7) -1.
JMPR        cc_NZ       MATRIX_LOOP  ; End-of-Loop test & branch
```

	Instruction Cycles	Program Words
Total	N^2+5N+5	24

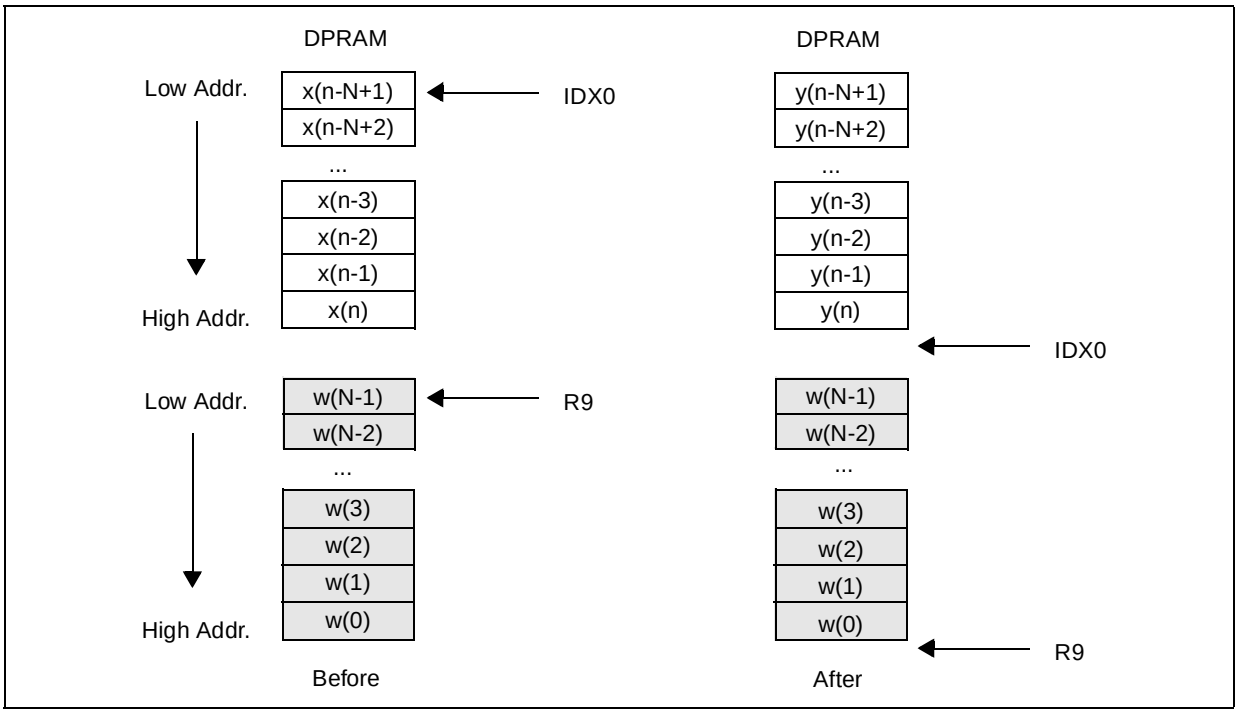
5.2.4 - N-real Multiply (windowing)

The formula is: `y(i) = x(i)Þw(i)` for i =0,1,...,N-1

The memory mapping is shown in Figure 6. To minimize the loop overhead, this program uses “loop unrolling”. The associated pseudo code is:

```
; x(n) = input signal at time n.
; w(n) = window coefficient at time n.
;
for (i=0 to N-1) {
                y(i)= x(i)*w(i);
                }
```

Figure 6 : Memory Map of the N-Real Multiply



This routines assumes that the following general purpose and co-processor registers (SFRs) have been initialized once for ever and L is a multiple of 4:

– R9 contains the w(N-1) address.

– IDX0 contains the x(n-N+1) address.

– QX0 and QR0 with N-1.

```
                ;
                ; Initialize the Loop Count
                ;
                MOV             R3          #N/4                ; (R3) ← N/4.
                ;
                ; Unrolled Loop
                ;
WINDOW_LOOP     CoMUL           [IDX0],     [R9-]       rnd     ; (ACC) ← w(i)*x(i) + rnd;
                                                                ; (R9) ← (R9)+2
                CoSTORE         [IDX0+],    MAH                 ; (IDX0) ← (ACC)
                                                                ; (IDX0) ← (IDX0)+2
                CoMUL           [IDX0],     [R9-]       rnd     ; (ACC) ← w(i+1)*x(i+1)+rnd;
                                                                ; (R9) ← (R9)+2
                CoSTORE         [IDX0+],    MAH                 ; (IDX0) ← (ACC)
                                                                ; (IDX0) ← (IDX0)+2
                CoMUL           [IDX0],     [R9-]       rnd     ; (ACC) ← w(i+2)*x(i+2)+rnd;
                                                                ; (R9) ← (R9)+2
                CoSTORE         [IDX0+],     MAH                ; (IDX0) ← (ACC)
                                                                ; (IDX0) ← (IDX0)+2
                CoMUL           [IDX0],     [R9-]       rnd     ; (ACC) ← w(i+3)*x(i+3)+rnd;
                                                                ; (R9) ← (R9)+2
                CoSTORE         [IDX0+],    MAH                 ; (IDX0) ← (ACC)
                                                                ; (IDX0) ← (IDX0)+2
                ;
                ; End_of_loop Checking.
                ;
                CMPD1           R3          #0h                 ; (R3) ← (R3)-1.
                JMPR            cc_NZ       WINDOW_LOOP         ; End-of-Loop test & branch.
```

	Instruction Cycles	Program Words
Total	2N+3N/4	5+ (2*2)*4

Note: The number of Instruction Cycles and Program Words required for this application depends on the “unrolling factor”. “2N” corresponds to the number of cycles per coefficient, “2N/4” corresponds to the branch penalty when the “unrolling factor” is 4. Similarly, “(2*2)*4-4” corresponds to the increase in program words when the “unrolling factor” is 4.
Typically, if URF defines the factor, the execution time and number of program words becomes: 2N+3N/URF instruction cycles, and 5+ 4*URF program words.

5.3 - FIR Filter-Real Correlation-Convolution

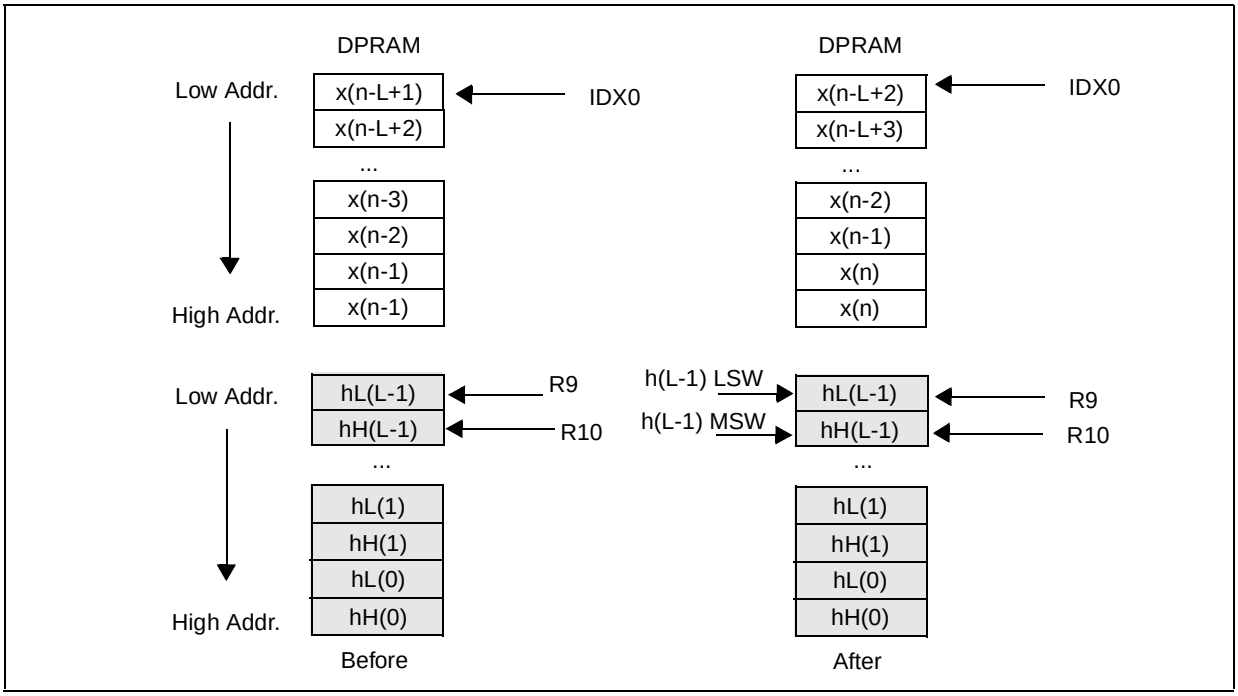
5.3.1 - Multiple Precision FIR Filter

The pseudo code is:

```
;  x(n) = input signal at time n. 16 bit value.
;  y(n) = output signal at time n. 16 bit value.
;  h(k) = k'th coefficient. 32-bit value.
;  L = Number of coefficient taps in the filter.
;
y(n)=0;
for (k=0 to L-1){
                y(n)= y(n) + h(k)*x(n-k);
                }
```

This program illustrates the use of multiply/multiply-accumulate instructions, "CoMIN & CoMAX" (performing a programmable saturation), and a shift instruction. The corresponding memory map is shown below. It is assumed that the coefficients and samples have been initialized by another routine.

Figure 7 : Memory Map for FIR



This routine assumes that the following general purpose registers and co-processor registers (SFRs) have been initialized:

– R0 with 0000h
– R1 with the 16-bit MAXimum tolerated value
– R2 contains the 16-bit MINimum tolerated value
– R9 contains the hL(L-1) address (low word of the coefficient)
– R10 contains hH(L-1) address (high word of the coefficient)
– IDX0 contains the x(n-L+1) address
– QX0 with 2*(L-1)
– QR0 with 4
– and QR1 with 4*(L-1)

```
                ;
                ; Repeat Count Initialization (repeat count > 31)
                ;
                MOV         MRW,        #L-4        ; (MRW) ← L-4.
                ;
                ; Read the new filter input from a (E)SFR and move it into the DPRAM
                ; at x(n-1) address therefore overwriting x(n-1).
                ;
                MOV         @x(n),      ADC_sfr     ; move the new input x(n)
                ;
                ; FIR prolog (LSWs of Impulse response): first multiplication
                ;
                CoMULsu     [IDX0+],    [R9+QR0]    ; (ACC) ← hL(L-1)*x(n-L+1)
                                                    ; (IDX0) ← (IDX0)+2,
                                                    ; (R9) ← (R9)+4.
                ;
                ; FIR loop (LSWs of Impulse response) Repeat the same MAC instruction L-2 times
                ;
REPEAT MRW TIMES CoMACsu    [IDX0+],    [R9+QR0]    ; (ACC) ← (ACC) + hL(i)*x(n-i)
                                                    ; (IDX0) ← (IDX0)+2,
                                                    ; (R9) ← (R9)+4.
                ;
                ; FIR epilog (LSWs of Impulse response): last MAC instruction and provide y(n)
                ; in an appropriate format
                ;
                CoMACsu     [IDX0-QX0],  [R9-QR1]   ; (ACC) ← (ACC) + hL(0)*x(n)
                                                    ; & x(n-L+1) ← x(n-L+2),
                                                    ; (IDX0) ← (IDX0)-2*(L-1),
                                                    ; (R9) ← (R9)-4*(L-1).
                ;
                ; Rounding & Shift
                ;
                MOV         MRW,        #L-4        ; (MRW)=L-4.
                CoRND                               ; (ACC)=(ACC)+rnd
                CoASHR      8,                      ; (ACC)=(ACC)>>8
                CoASHR      8,                      ; (ACC)=(ACC)>>8
                            ;
                ; FIR prolog (MSWs of Impulse response): first multiplication
                ;
```

```
                CoMAC       '[IDX0+],       [R10+QR0]     ; (ACC) ← hH(L-1)*x(n-L+1)
                                                          ; (IDX0) ← (IDX0)+2,
                                                          ; (R10) ← (R10)+4.
                ;
                ; FIR loop (MSWs of Impulse response)Repeat the same MAC instruction L-2 times
                ;
REPEAT MRW TIMES CoMACM     [IDX0+],        [R10+QR0]     ; (ACC) ← (ACC) + hH(i)*x(n-i)
                                                          ; & x(n-i-1) ← x(n-i),
                                                          ; (IDX0) ← (IDX0)+2,
                                                          ; (R10) ← (R10)+4.
                ;
                ; FIR epilog (MSWs of Impulse response): last MAC instruction and provide
                ; y(n) in an appropriate format
                ;
                CoMACM      [IDX0-QX0],     [R10-QR1]     ; (ACC) ← (ACC) + hH(0)*x(n)
                                                          ; & x(n-l+1) ← x(n-L+2),
                                                          ; (IDX0) ← (IDX0)-2*(L-1),
                                                          ; (R10) ← (R10)-4*(2L-1).
                ;
                ; Shift & Rounding
                ;
                CoASHR      #data3,         rnd           ; (ACC) ← (ACC)>>a #data3 +rnd
                ;
                ; Limiting
                ;
                CoMIN       R0,             R1            ; (ACC) ← Min((ACC), MAX).
                CoMAX       R0,             R2            ; (ACC) ← Max((ACC),MIN).
                ;
                ;Write the new filter output y(n) into a (E)SFR.
                ;
                NOP                                       ; Pipeline Effect.
                MOV         DAC_sfr,        MAH           ; move the new output y(n).
```

	Instruction Cycles	Program Words
Read Input sample	1	2
Initialization	2	4
FIR Loop	2L+3	18
Post -Processing	4	7
Write Output sample	1	2
Total	2L+11	33

5.4 - IIR Filters

5.4.1 - Nth Order IIR Filter: Direct Form 1

The rules for the implementation of FIR filters can be extended to IIR filters. The Nth-order difference equation is:

$$y(n) = \sum_{k=1}^{N} a(k) \times y(n-k) + \sum_{k=0}^{M} b(k) \times x(n-k)$$

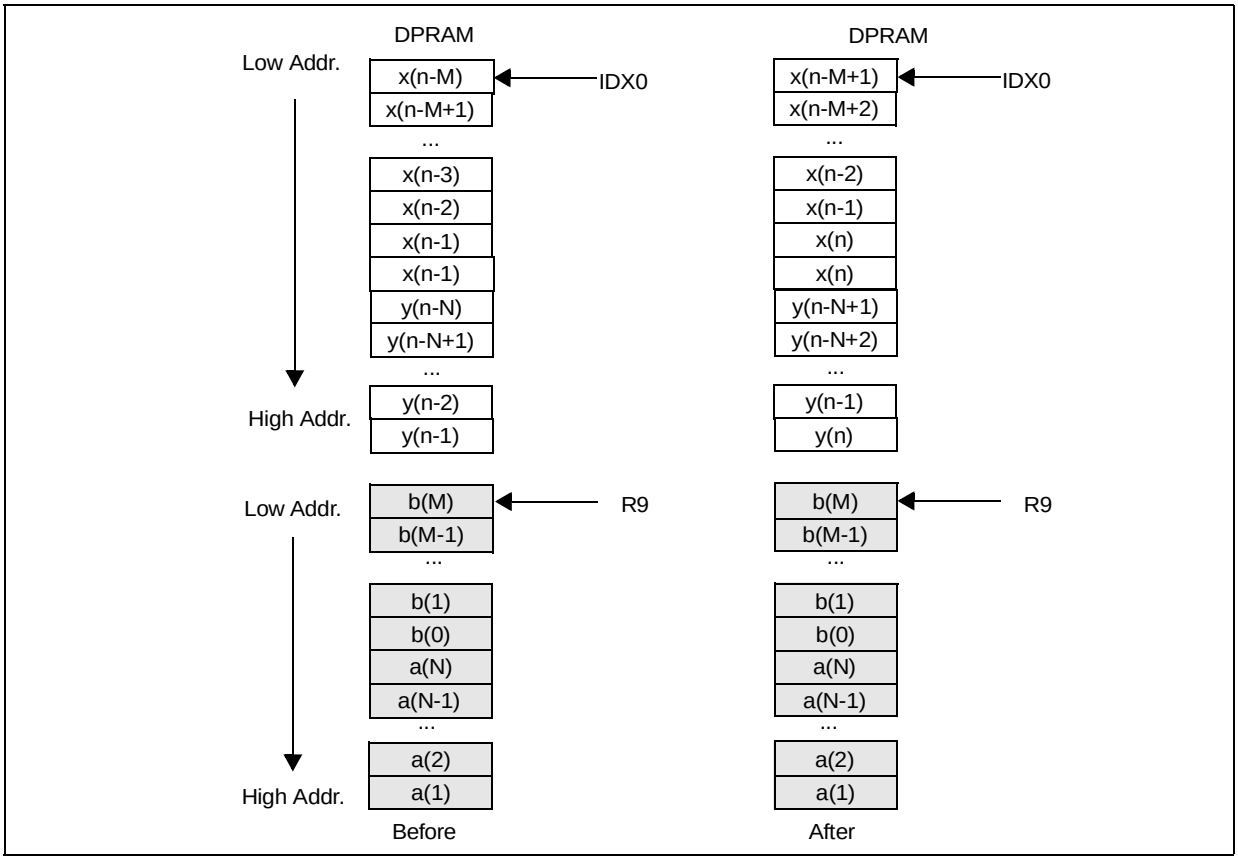
This can be called “Direct Form 1”. The associated pseudo code is:

```
;  x(n) = input signal at time n
;  y(n) = output signal at time n
;  a(k), b(k)= IIR coefficients
;  N, M refer to the above equation

y(n)=0;
for (k=0 to M)  {
                y(n)= y(n) +b(k)*x(n-k)
                }
for (k=1 to N)  {
                y(n)= y(n) +a(k)*y(n-k);
                }
```

Figure 8 shows the memory map. It has been assumed that the coefficients and samples have been initialized by another routine.

Figure 8 : Memory Map for IIR Direct Form 1



This routines assumes that the following general purpose and co-processor registers (SFRs) have been initialized:

– R0 with 0000h
– R1 with the 16-bit MAXimum tolerated value
– R2 contains the 16-bit MINimum tolerated value
– R9 contains the b(M) address
– R10 contains the y(n) address
– IDX0 contains the x(n-M) address
– QX0 with (N+M)*2
– QR0 with (N+M)*2

```
                    ;
                    ; Repeat Count Initialization (repeat count > 31) for the first IIR Loop
                    ;
                    MOV                 MRW,        #M-1        ; (MRW) ← M-1.
                    ;
                    ; Read the new filter input from a (E)SFR & move it into the DPRAM
                    ; at x(n-1) address, overwriting x(n-1).
                    ;
                    MOV                 @x(n),      ADC_sfr     ; move the new input x(n)
                    ;
                    ; Prolog of the First IIR loop.
                    ;
                    CoMUL               [IDX0+],    [R9+]       ; (ACC) ← b(M)*x(n-M)
                                                                ; (IDX0) ←(IDX0)+2,
                                                                ; (R9) ← (R9)+2.
                    ;
                    ; First IIR loop.
                    ;
REPEAT MRW TIMES    CoMACM              [IDX0+],    [R9+]       ; (ACC) ← (ACC)+b(i)*x(n-i)
                                                                ; & x(n-i-1) ← x(n-i),
                                                                ; (IDX0) ← (IDX0)+2,
                                                                ; (R9) ← (R9)+2.
                    ;
                    ; Repeat Count Initialization (repeat count > 31) for the second.
                    ;
                    MOV                 MRW,        #N-3        ; (MRW) ← N-3.
                    ;
                    ; prolog of the Second IIR loop.
                    ;
                    CoMAC               [IDX0+],    [R9+]       ; (ACC) ← a(N)*y(n-N)
                                                                ; (IDX0) ← (IDX0)+2,
                                                                ; (R9) ← (R9)+2.
```

```
                    ;
                    ;
                    ; Second IIR loop.
                    ;
REPEAT MRW TIMES    CoMACM          [IDX0+],      [R9+]      ; (ACC) ← (ACC) + a(i)*y(n-i)
                                                             ; & y(n-i-1) ← y(n-i),
                                                             ; (IDX0) ← (IDX0)+2,
                                                             ; (R9) ← (R9)+2.
                    ;
                    ; Epilog of the second IIR loop.
                    ;
                    CoMACM          [IDX0-QX0],   [R9-QR0]   ; (ACC) ← (ACC)+h(0)*x(n)
                                                             ; & y(n-2) ← y(n-1),
                                                             ; (IDX0) ← (IDX0)-2*(N+M),
                                                             ; (R9) ← (R9)-2*(N+M).
                    ;
                    ; Rounding
                    ;
                    CoRND                                    ; (ACC) ← (ACC) + rnd
                    ;
                    ; Limiting
                    ;
                    CoMIN           R0,           R1         ; (ACC) ← Min((ACC), MAX).
                    CoMAX           R0,           R2         ; (ACC) ← Max((ACC),MIN).
                    ;
                    ; Write the new filter output, y(n), into memory.
                    ;
                    CoSTORE         [R10],        MAH        ; ((R10)) ← y(n).
                    ;
```

	Instruction Cycles	Program Words
Read Input sample	1	2
Initialization	2	4
DF1 IIR Loop	N+M	10
Post -Processing	3	6
Write Output sample	1	2
Total	N+M+7	24

5.4.2 - N^{th} Order IIR Filter: Direct Form 2

The following equations equally represent the Nth Order IIR filter:

$$u(n) = x(n) + \sum_{k=1}^{N} a(k) \cdot u(n-k)$$

$$y(n) = \sum_{k=0}^{N} b(k) \cdot u(n-k)$$

These equations use the intermediate state variable vector U={u(n), u(n-1), u(n-2),..., u(n-N)}. This representation is called “Direct Form 2” and is illustrated by Figure 9. Direct Form 2 has an advantage over Direct Form 1 as it requires less data memory.

The associated pseudo code is:

```
; x(n) = input signal at time n.
; u(n) = state variable at time n.
; y(n) = output signal at time n.
; a(k), b(k)= IIR coefficients.
; It is assumed N = M.
;
u(n)=x(n);
for (k=1 to N)     {
                   u(n)= u(n) +a(k)*u(n-k);
                   }
y(n)=b(0)*u(n);
for (k=1 to N)     {
                   y(n)= y(n) +b(k)*u(n-k);
                   }
```

Figure 9 : IIR Direct Form 2

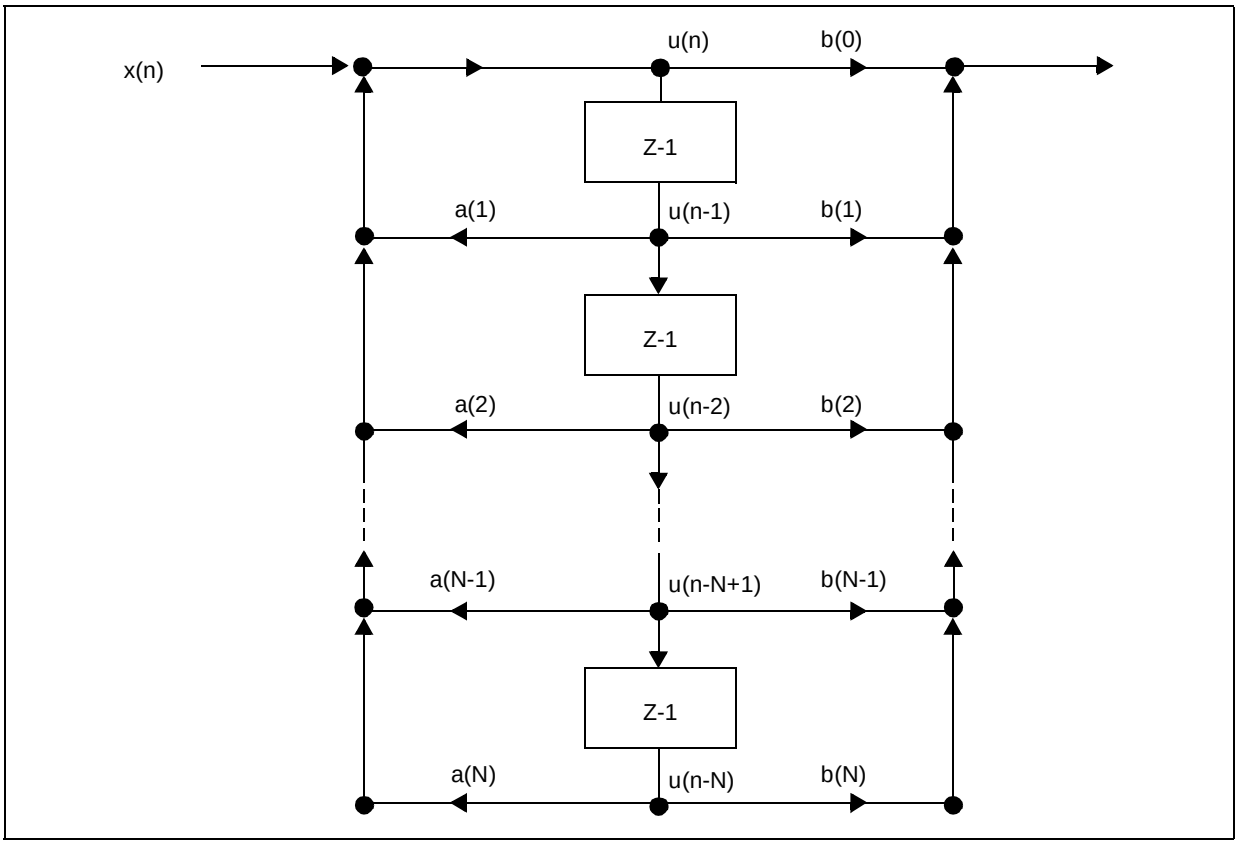
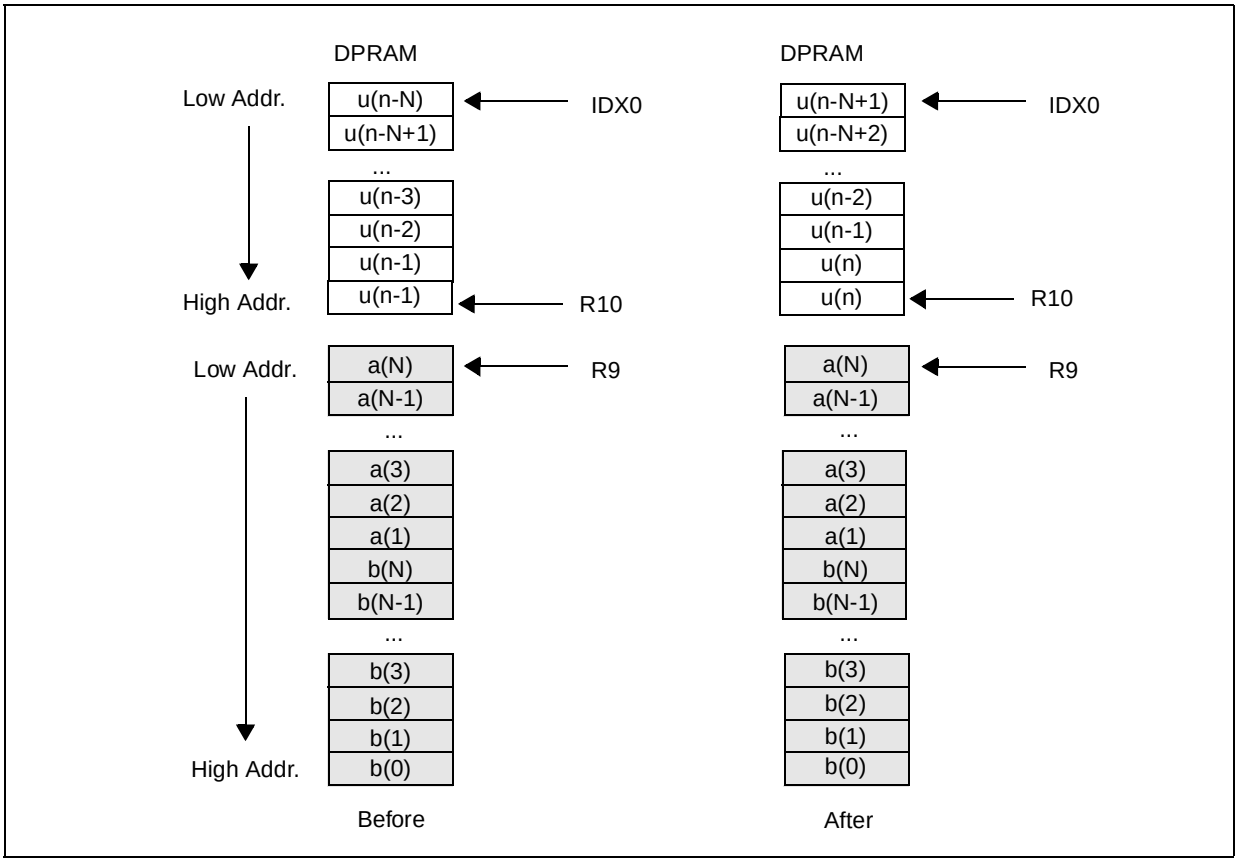


Figure 10 shows the corresponding memory map. It has been assumed that the coefficients and samples have been initialized by another routine.

Figure 10 : Memory Map for Nth Order IIR Filter Direct Form 2



This routines assumes that the following general purpose and co-processor registers (SFRs) have been initialized:

– R0 with 0000h
– R1 with the 16-bit MAXimum tolerated value
– R2 contains the 16-bit MINimum tolerated value
– R9 contains the a(N) address
– R10 contains the u(n) address
– IDX0 contains the u(n-N) address
– QX0 with 2*(N-1) and QX1 with 2N
– QR0 with 4N-2, [2N+2*(N-1)]

```
                ;
                ; Repeat Count Initialization (repeat count > 31) for the first IIR Loop
                ;
                MOV             MRW,            #N-2            ; (MRW) ← N-2.
                ;
                ; Read the new filter input from a (E)SFR and move it into the Accumulator.
                ;
                MOV             MAH,            ADC_sfr         ; (MAH) ← x(n),
                                                                ; (MAE) ← 8 times (MAH.15),
                                                                ; (MAL) ← 0000h.
                ;
                ; First IIR loop: u(n) computation.
                ;
REPEAT MRW TIMES CoMAC          [IDX0+],        [R9+]           ; (ACC) ← (ACC) + a(i)*u(n-i)
                                                                ; (IDX0) ← (IDX0) + 2,
                                                                ; (R9) ← (R9)+2.
                ;
                ; Epilog of the first IIR loop.
                ;
                CoMAC           [IDX0-QX0],     [R9+], rnd      ; (ACC) ← (ACC)+a(1)*u(n-1)
                                                                ; +rnd
                                                                ; (IDX0) ← (IDX0)-2*(N-1),
                                                                ; (R9) ← (R9)+2.
                ;
                ; Repeat Count Initialization (repeat count > 31) for the second.
                ;
                MOV             MRW,            #N-2            ; (MRW) ← N-2.
                ;
                ; Move u(n) into memory.
                ;
                CoSTORE         [R10],          MAS             ; ((R10)) ← u(n)
                ;
                ; Prolog of the Second IIR loop.
                ;
                CoMAC           [IDX0+],        [R9+]           ; (ACC) ← b(N)*u(n-N)
                                                                ; (IDX0) ← (IDX0)+2,
                                                                ; (R9) ← (R9)+2.
                ;
```

```
                    ; Second IIR loop.
                    ;
REPEAT MRW TIMES    CoMACM      [IDX0+],        [R9+]          ; (ACC) ← (ACC)+b(i)*u(n-i)
                                                               ; & u(n-i-1) ← u(n-i),
                                                               ; (IDX0) ← (IDX0)+2,
                                                               ; (R9) ← (R9)+2.
                    ;
                    ; Epilog of the Second IIR loop.
                    ;
                    CoMACM      [IDX0-QX1],     [R9-QR0], rnd  ; (ACC) ← b(0)*u(n)+rnd
                                                               ; & u(n-1) ← u(n),
                                                               ; (IDX0) ← (IDX0)-2N,
                                                               ; (R9) ← (R9)-4N-2.
                    ;
                    ; Limiting
                    ;
                    CoMIN       R0,             R1             ; (ACC) ← Min((ACC), MAX).
                    CoMAX       R0,             R2             ; (ACC) ← Max((ACC),MIN).
                    ;
                    ;Write the new filter output y(n) into a (E)SFR.
                    ;
                    NOP                                        ; Pipeline Effect.
                    MOV         DAC_sfr,        MAH            ; move the new output y(n).
```

	Instruction Cycles	Program Words
Read Input sample	1	2
Initialization	2	4
DF2 IIR Loop	2N+1	12
Post -Processing	2	4
Write Output sample	2	3
Total	2N+8	25

5.4.3 - N-Cascaded Real Biquads (Direct Form 2)

A high-order filter can be implemented, either as a single section, or as a combination of first and second order sections. The single section form is quicker and easier to implement, but generates a larger numerical error. This increased error occurs for two reasons:

– The long filter computation process accumulates errors from multiplication with quantized coefficients.

– The roots of high-order polynomials are increasingly sensitive to changes in their quantized coefficients.

Therefore, the single section form is not recommended except for a very low order controller. (see “Nth Order IIR Filter: Direct Form 1” on page 20).

To implement a high-order transfer function, first decompose it into first order and second order blocks (biquads), and then connect these blocks in a cascade. The following paragraphs illustrate this technique for an even numbers of cascaded biquads.

Unlike conventional digital signal processors, the MAC co-processor is able to repeat a single instruction at high speed but does not offer flexible and fast hardware looping. Consequently, to perform a loop containing more than one instruction the programmer must use the regular instruction set incurring a several cycle penalty for the end-of-loop detection. “Loop Unrolling” minimizes this penalty but increases the number of instructions. In the following section the loop unrolling technique will not be employed.

Equations of a Direct Form 2 N^{th} Order IIR filter applied to a second order filter (N=2) yield:

$$u^i(n) = x^i(n) - a^i(1) \times u^i(n-1) - a^i(2) \times u^i(n-2)$$

$$(y^i(n) = b^i(0) \times u^i(n) + b^i(1) \times u^i(n-1) + b^i(2) \times u^i(n-2))x$$

Where “i” specifies the biquad number. Note that $y^j(n)=x^{i+1}(n)$.

For simplicity, it has been assumed that no overflow occurs on $u^i(n)$ and $y^i(n)$.

The naming convention is:

; xi(n) = input signal at time n of biquad number i.

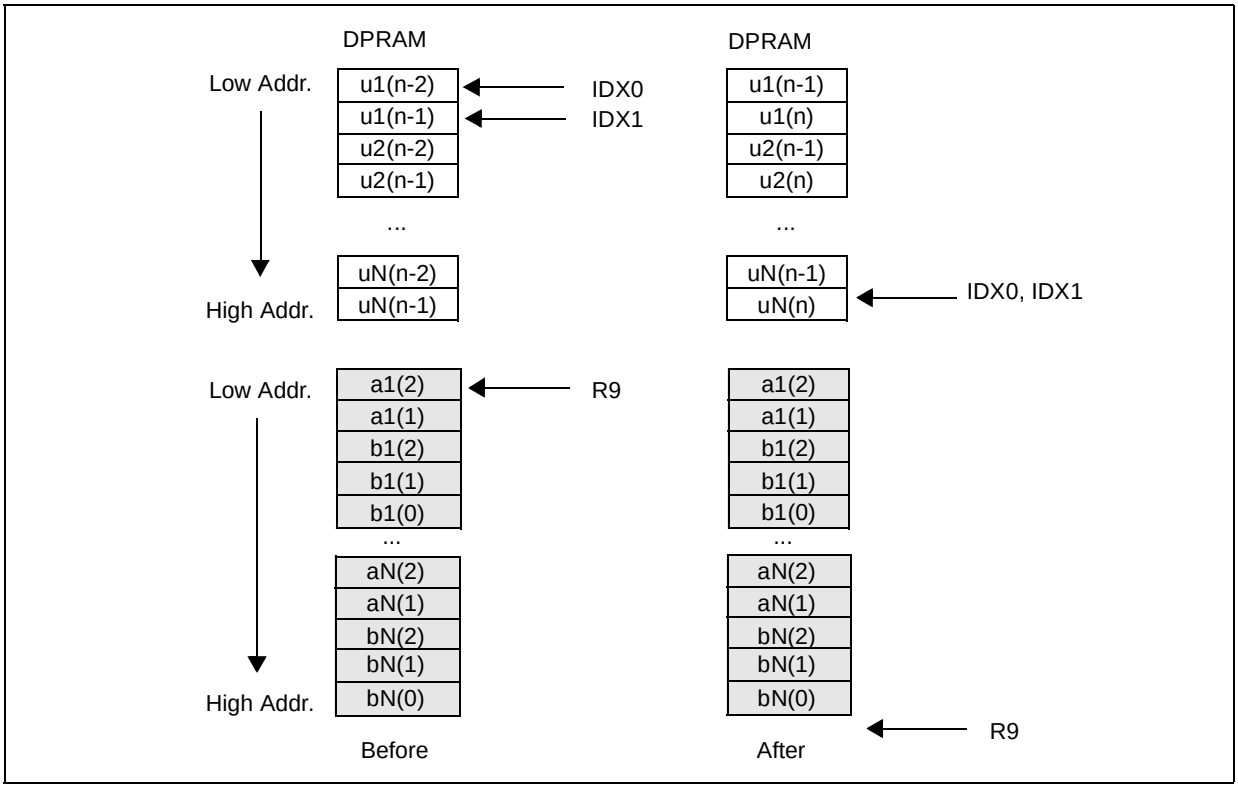
; ui(n) = state variable at time n of biquad number i.

; y(n) = output signal at time n of biquad number i.

; ai(k), bi(k)= Coefficients of biquad number i.

Figure 11 shows the corresponding memory map and assumes that both coefficients and samples have been initialized by another routine.

Figure 11 : Memory Map for N-Cascaded Real Biquad IIR



This routines assumes that the following general purpose and co-processor registers have been initialized:
– R0 with 0000h
– R1 with the 16-bit MAXimum tolerated value
– R2 contains the 16-bit MINimum tolerated value
– R9 contains the a1(2) address
– R10 contains the R5 physical address
– IDX0 contains the u1(n-2) address
– IDX1 contains the u1(n-1) address
– QX0 with 2*(N-1)
– QX1 with 4

```
                ;
                ; Initialize the Loop Count: N
                ;
                MOV             R3              #N              ; (R3) ← N.
                ;
                ; Read the new filter input from a (E)SFR and move it into the Accumulator.
                ;
                MOV             MAH,            ADC_sfr         ; (MAH) ← x(n),
                                                                ; (MAE) ← 8 times (MAH.15),
                                                                ; (MAL) ← 0000h.
DF2_BIQUAD_LOOP
                ; First Biquad iteration
                ;
                CoMAC-          [IDX0+],        [R9+]           ; (ACC) ← (ACC)-ai(2)*ui(n-2)
                                                                ; (IDX0) ← (IDX0)+2,
                                                                ; (R9) ← (R9)+2.
                CoMAC-          [IDX0-],        [R9+]           ; (ACC) ← (ACC)-ai(1)*ui(n-1)
                                                                ; (IDX0) ← (IDX0)-2,
                                                                ; (R9) ← (R9)+2.
                CoRND                                           ; (ACC) ← (ACC)+rnd
                ;
                ; Write ui(n), into a GPR (R5)
                ;
                CoSTORE         R5,             MAS             ; (R5) ← ui(n).
                ;
                ; Second Biquad iteration.
                ;
                CoMUL           [IDX0+],        [R9+]           ; (ACC) ← (ACC) + bi(2)*ui(n-2)
                                                                ; (IDX0) ← (IDX0)+2,
                                                                ; (R9) ← (R9)+2.
```

```
CoMACM      [IDX0+],        [R9+]       ; (ACC) ← (ACC)+bi(1)*ui(n-1)
                                        ; & ui(n-2) ← ui(n-1)
                                        ; (IDX0) ← (IDX0)+2,
                                        ; (R9) ← (R9)+2.
CoMAC       R5,             [R9+]       ; (ACC) ← (ACC) + bi(0)*ui(n)
                                        ; (R9) ← (R9)+2.
;
; Write ui(n) to memory.
;
CoMOV       [IDX1+QX1]      [R10]       ; ui(n-1) ← ui(n).
                                        ; (IDX1) ← (IDX1)+4,
;
; End_of_loop Checking.
;
CMPD1       R3              #0h         ; (R3)  ← (R3)-1.
JMPR        cc_NZ           DF2_BIQUAD_LOOP ; End-of-Loop test & branch.
;
; Limiting
;
CoMIN       R0,             R1          ; (ACC) ← Min((ACC), MAX).
CoMAX       R0,             R2          ; (ACC) ← Max((ACC),MIN).
;
;Write the new filter output y(n) into a (E)SFR.
;
NOP                                     ; Pipeline Effect.
MOV         DAC_sfr,        MAH         ; move the new output y(n).
```

	Instruction Cycles	Program Words
Read Input sample	1	2
Initialization	1	2
DF2 Biquad Loop	11N-1	19
Post -Processing	3	5
Write Output sample	1	2
Total	11N+5	31

5.4.4 - N-cascaded Real Biquads: Transpose Form

The equations of a Direct Form 2 N^{th} Order IIR filter applied to a second order filter (N=2) can yield:

$$y^i(n) = b^i(0) \times x^i(n) + u^i(n-1)$$

$$(u^i(n) = b^i(1) \times x^i(n-1) - a^i(1) \times y^i(n) + w^i(n-1))x$$

$$w^i(n) = b^i(2) \times x^i(n) - a^i(2) \times y^i(n)$$

Where “i” is the biquad number. Note that $y^i(n)=x^{i+1}(n)$. This form is also called the “Transpose Form”.

For simplicity, it has been assumed that no overflow occurs on $u^i(n)$ or $y^i(n)$. This form is suitable when the input-to-output delay must be minimized. The naming convention is:

; xi(n) = input signal at time n of biquad number i.

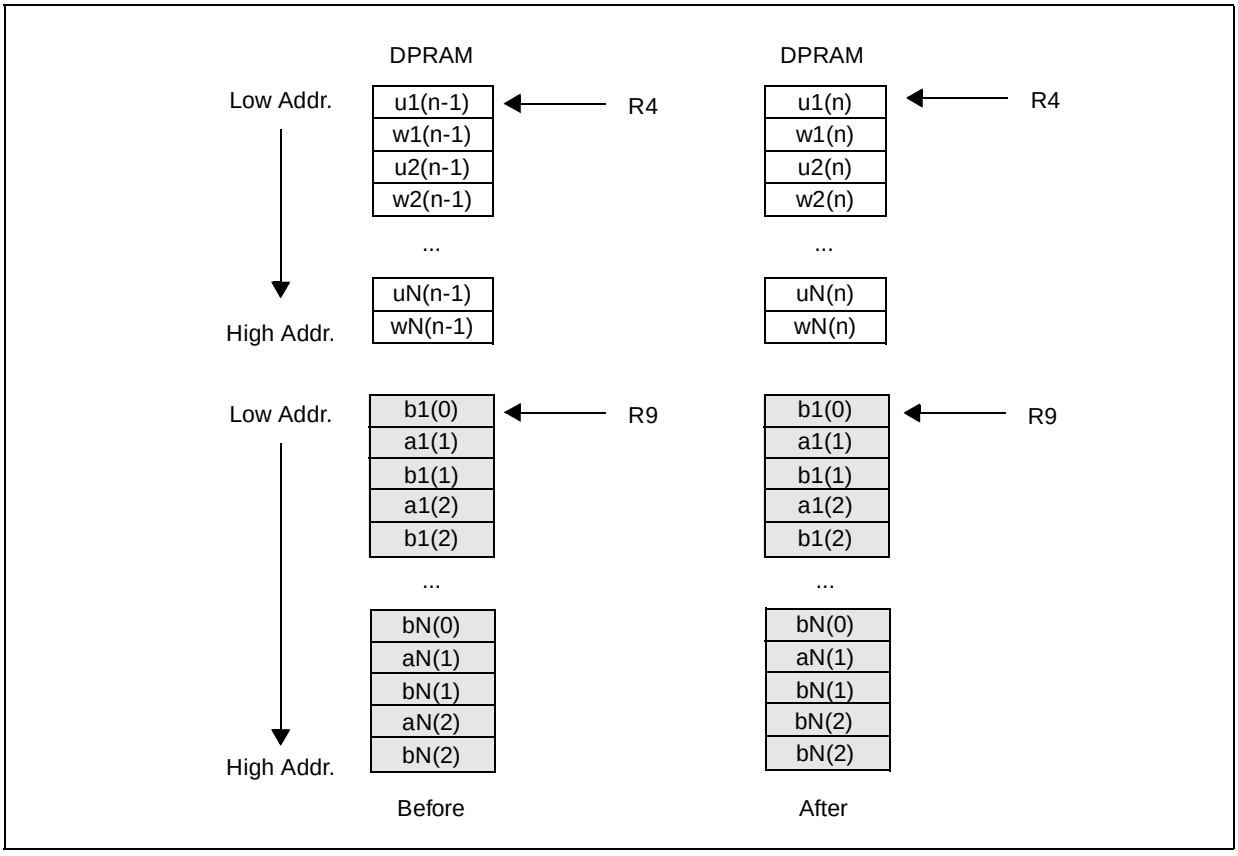
; ui(n), wi(n) = state variables at time n of biquad number i.

; yi(n) = output signal at time n of biquad number i.

; ai(k), bi(k)= Coefficients of biquad number i.

Figure 12 shows the corresponding memory map. It is assumed that both coefficients and samples have been initialized by another routine**.**

Figure 12 : Memory Map of N-Cascaded Real Biquads (Transpose Form)



This routines assumes that the following general purpose and co-processor registers (SFRs) have been initialized:

– R0 with 0000h
– R1 with the 16-bit MAXimum tolerated value
– R2 contains the 16-bit MINimum tolerated value
– R4 contains the u1(n-1) address
– R9 contains the b1(0) address
– R10 contains the R5 physical address
– QR0 with 2*(N-1)
– QR1 with 10N

```
                ;
                ; Initialize the Loop Count: N
                ;
                MOV         R3          #N-1                ; (R3) ← N-1.
                ;
                ; Read the new filter input from a (E)SFR and move it into a GPR (R5).
                ;
                MOV         R5,         ADC_sfr             ; (R5) ← x(n)
                ;
TF_BIQUAD_LOOP:
                ;
                ; Compute yi(n)
                ;
                CoLOAD      [R4+],      R0                  ; (ACC) ← ui(n-1)
                                                            ; (R4) ← (R4)+2.
                CoMAC       [R9+],      R5      rnd         ; (ACC) ← (ACC)+bi(0)*xi(n)
                                                            ; +rnd
                                                            ; (R9) ← (R9)+2.
                ;
                ; Write yi(n) into R8.
                ;
                CoSTORE     R8,         MAS                 ; (R8) ← limited(yi(n)).
                ;
                ; Compute ui(n)
                ;
                CoLOAD      [R4],       R0                  ; (ACC) ← wi(n-1)
                CoMAC-      [R9+],      R8                  ; (ACC) ← (ACC)-ai(1)*yi(n)
                                                            ; (R9) ← (R9)+2.
                CoMAC       [R9+],      R5      rnd         ; (ACC) ← (ACC)+bi(1)*xi(n)+rnd
                                                            ; (R9) ← (R9)+2.
                ;
```

```
; Write ui(n) into memory.
;
CoSTORE     [R4+],      MAS                 ; ui(n-1) ← ui(n).
                                            ; (R4) ← (R4)+2.
;
; Compute wi(n)
;
CoMUL-      [R9+],      R8                  ; (ACC) ← -ai(2)*yi(n)
                                            ; (R9) ← (R9)+2.
CoMAC       [R9+],      R5      rnd         ; (ACC) ← (ACC)+bi(2)*xi(n)+rnd
                                            ; (R9) ← (R9)+2.
;
; Write wi(n) into memory.
;
CoSTORE     [R4+],      MAS                 ; wi(n-1) ← wi(n).
                                            ; (R4) ← (R4)+2.
;
; Write yi(n) into R5.
;
MOV         R5          R8                  ; xj(n) ← yi(n).
;                                           ; j = i+1
; End_of_loop Checking.
;
CMPD1       R3          #0h                 ; (R3) ← (R3) -1.
JMPR        cc_NZ       TF_BIQUAD_LOOP      ; End-of-Loop test & branch.
;
; Compute yN(n)
;
CoLOAD      [R4+],      R0                  ; (ACC) ← uN(n-1)
                                            ; (R4) ← (R4)+2.
CoMAC       [R9+],      R5      rnd         ; (ACC) ← (ACC)+bN(0)*xN(n)
                                            ; +rnd
                                            ; (R9) ← (R9)+2.
;
; Write yN(n) into R8.
;
CoSTORE     R8,         MAS                 ; (R8) ← limited(y(n)).
;
```

```
; Limiting
;
CoMIN       R0,          R1              ; (ACC) ← Min((ACC), MAX).
CoMAX       R0,          R2              ; (ACC) ← Max((ACC),MIN).
;
;Write the new filter output y(n) into a (E)SFR.
;
NOP                                      ; Pipeline Effect
MOV         DAC_sfr,     MAH             ; move the new output y(n).
;
; Compute ui(n)
;
CoLOAD      [R4],        R0              ; (ACC) ← wN(n-1)
CoMAC-      [R9+],       R8              ; (ACC) ← (ACC)-aN(1)*yN(n)
                                         ; (R9) ← (R9)+2.
CoMAC       [R9+],       R5    rnd       ; (ACC) ← (ACC)+bN(1)*xN(n)
                                         ; +rnd
                                         ; (R9) ← (R9)+2.
;
; Write ui(n) into memory.
;
CoSTORE     [R4+],       MAS             ; uN(n-1) ← uN(n).
                                         ; (R4) ← (R4)+2.
;
; Compute wi(n)
;
CoMUL-      [R9+],       R8              ; (ACC) ← -aN(2)*yN(n)
                                         ; (R9) ← (R9)+2.
CoMAC       [R9-QR1],    R5    rnd       ; (ACC) ← (ACC)+bi(2)*xi(n)+rnd
                                         ; (R9) ← (R9)-2*(5N).
;
; Write wi(n) into memory.
;
CoSTORE     [R4-QR0],    MAS             ; wN(n-1) ← wN(n).
                                         ; (R4) ← (R4)-2*[2*(N-1)].
```

	Instruction Cycles	Program Words
Read Input sample	1	2
Initialization	1	2
TF Biquad Loop	14N-15	24
Output Post -Processing & Write	7	13
Filter Post-Processing & Update	7	14
Total	14N+1	55

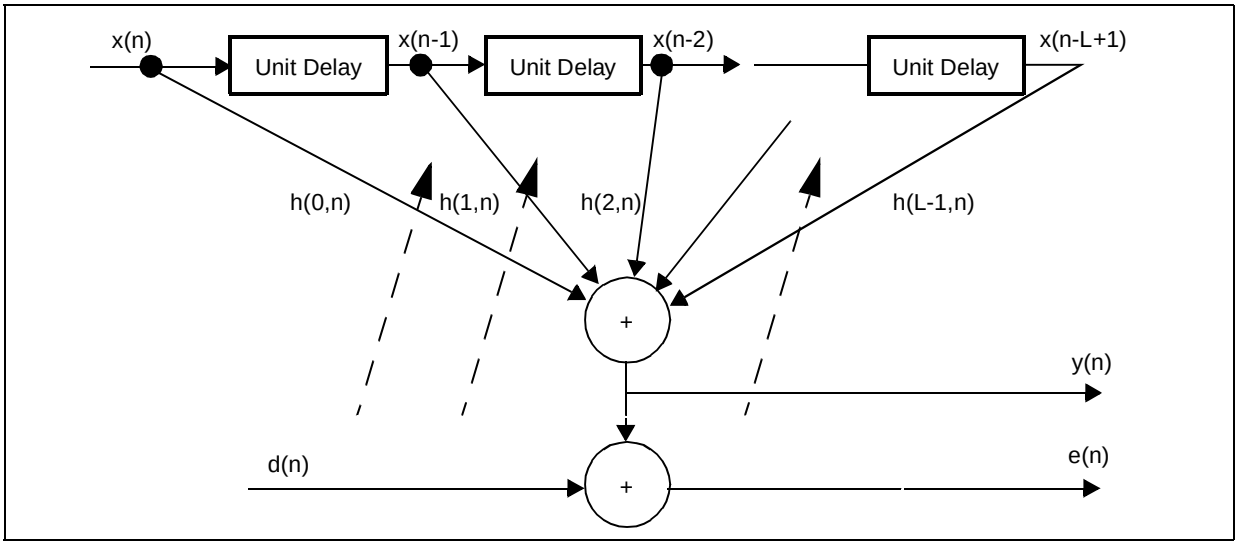
5.5 - LMS Adaptive Filter

5.5.1 - Single-Precision LMS Adaptive Filter

An adaptive filter contains coefficients that are updated by an adaptive algorithm to optimize the filter's response to a desired performance criterion. Generally, adaptive filters have two distinct parts: a filter with a structure designed to perform a processing function, and an adaptive algorithm for adjusting the filter coefficients to improve its performance. The incoming signal x(n) is weighted in a digital filter to produce an output y(n). The adaptive algorithm adjusts the filter weights to minimize the error e(n) between the filter output y(n) and the desired response of the filter d(n).

The Single-Precision LMS Adaptive Filter is a FIR filter whose coefficients are updated at each iteration according to an error signal e(n) equal to d(n)-y(n), where d(n) is the desired signal at time n and y(n) is the FIR output. Figure 13 illustrates this filter.

Figure 13 : LMS Adaptive Filter



The corresponding pseudo code is:

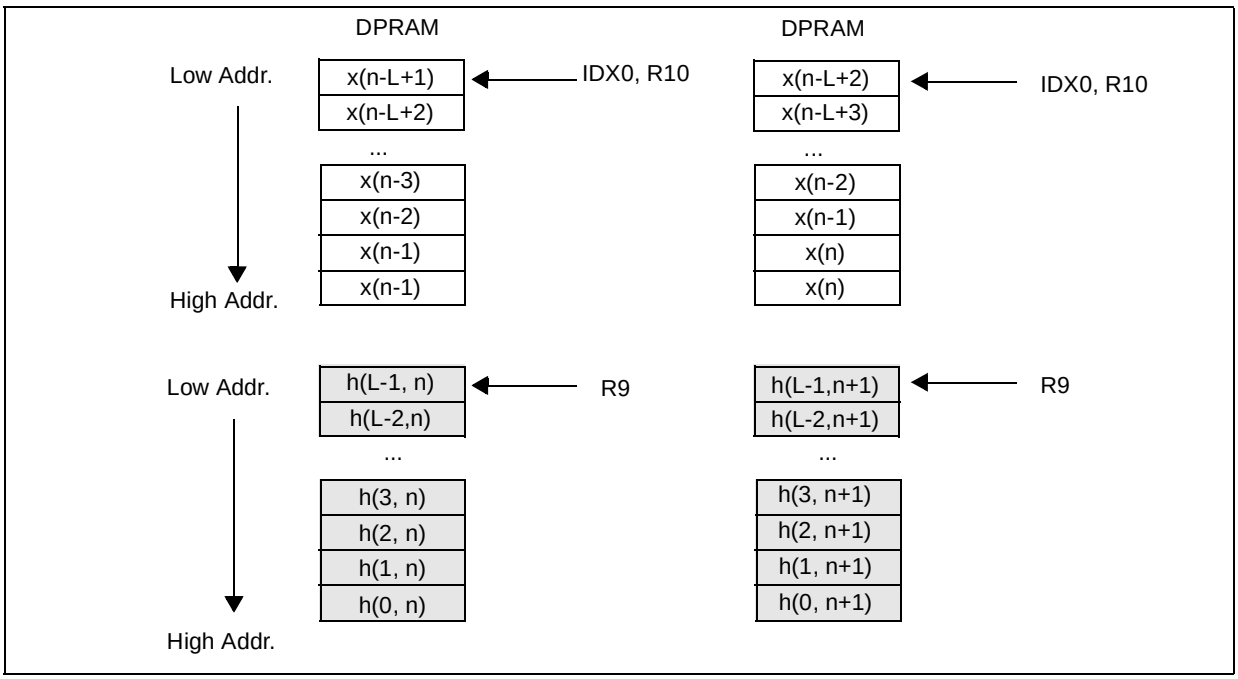
```
; x(n) = input signal at time n.
; d(n) = desired signal at time n.
; y(n) = output signal at time n.
; h(k, n) = k'th coefficient at time n.
; Mu= adaptive gain.
; L = Number of coefficient taps in the filter.
;
y(n)=0;
for (k=0 to L-1) {
                    y(n)= y(n) + h(k,n)*x(n-k);
                    }

e(n)=d(n)-y(n);
for (k=0 to L-1) {
                    h(k,n+1)= h(k,n) - Mu*x(n-k)*e(n);
                    }
```

Figure 14 shows the corresponding memory map. It has been assumed that both the coefficients and samples have been initialized by another routine.

Unlike pure DSP filters, this filter is implemented in two steps, FIR output computation is followed by an update of the coefficients.

Figure 14 : Memory Map for LMS Adaptative Filter



This routines assumes that the following general purpose and co-processor registers (SFRs) have been initialized once for ever and that L is less than 31:

– R0 with 0000h
– R1 with the 16-bit MAXimum tolerated value
– R2 contains the 16-bit MINimum tolerated value
– R8 contains the adaptive gain (Mu) value
– R9 contains the h(L-1,n) address
– R10 contains the x(n-L+1) address
– IDX0 contains the x(n-L+1) address
– QX0 and QR0 with 2*(L-1)

```
;
; Read the new filter input from a (E)SFR and move it into the DPRAM
; at x(n-1) address overwriting therefore x(n-1).
;
MOV         @x(n),      ADC_sfr     ; move the new input x(n)
;
; FIR prolog: first multiplication
;
CoMUL       [IDX0+],    [R9+]       ; (ACC) ← h(L-1,n)*x(n-L+1)
                                    ; (IDX0) ← (IDX0)+2,
```

```
                                                        ; (R9) ← (R9)+2.
                ;
                ; FIR loop: Repeat L-2 times the same MAC instruction.
                ;
REPEAT L-3 TIMES  CoMAC     [IDX0+],      [R9+]         ; (ACC) ← (ACC)+h(i)*x(n-i)
                                                        ; (IDX0) ← (IDX0)+2,
                                                        ; (R9) ← (R9)+2.
                ;
                ; FIR epilog: last MAC instruction and provide y(n) in an appropriate format
                ;
                CoMAC       [IDX0-QX0],   [R9-QR0]      ; (ACC) ← (ACC)+h(0)*x(n)
                                                        ; (IDX0) ← (IDX0)-2*(L-1),
                                                        ; (R9) ← (R9)-2*(L-1).
                ;
                ; Shift & Rounding
                ;
                CoASHR      #data3,             rnd     ; (ACC) ← (ACC)>>a #data3
                                                        ;+rnd
                ;
                ; Limiting
                ;
                CoMIN       R0,           R1            ; (ACC) ← Min((ACC), MAX).
                CoMAX       R0,           R2            ; (ACC) ← Max((ACC),MIN).
                ;
                ;Write the new filter output y(n) into an (E)SFR.
                ;
                CoSTORE     R6            MAH           ; (R6) ← y(n).
                MOV         DAC_sfr,      R6            ; move the new output y(n).
                ;
                ;Read d(n) and move it into a GPR.
                ;
                MOV         R5,           @d(n)         ; (R5) ← d(n)
                ;
                ; Error, e(n), Calculation
                ;
                SUB         R5,           R6            ; (R5) ← d(n)-y(n)=e(n)
                MOV         @e(n),        R5            ; e(n-1) ← e(n)
                CoMUL       R5,           R8            ; (ACC) ← Mu*e(n)
                CoNEG                             rnd   ; (ACC) ← -(ACC)+rnd
                CoSTORE     R11,          MAS           ; (R11) ← -Mu*e(n).
```

```
                ;
                ; Coefficients' Updating
                ;
                MOV         R3,         #L-2            ; (R3) ← L
                ;
                ; Coefficient Update Prolog.
                ;
                CoLOAD      [R9],       R0              ; (ACC) ← h(L-1,n)
                CoMAC       R11         [R10+],  rnd    ; (ACC) ← h(L-1,n) -
                                                        ; Mu.e(n)*x(n-L+1)+rnd
                                                        ; (R10) ← (R10)+2.
                CoSTORE     [R9+],      MAS             ; h(L-1,n) ← h(L-1,n+1).
                                                        ; (R9) ← (R9)+2.
                ;
                ; Coefficient Update Loop.
                ;
LMS_LOOP:
                ;
                CoLOAD      [R9],       R0              ; (ACC) ← h(k,n)
                CoMACM      R11         [R10+],  rnd    ; (ACC) ← h(k,n) -Mu.e(n)*x(n-k)
                                                        ;+rnd
                                                        ; x(n-k-1) ← x(n-k).
                                                        ; (R10) ← (R10)+2.
                CoSTORE     [R9+],      MAS             ; h(k,n) ← h(k,n+1).
                                                        ; (R9) ← (R9)+2.
                ;
                ; End_of_loop Checking.
                ;
                CMPD1       R3          #0h             ; (R3) ← (R3) -1.
                JMPR        cc_NZ       LMS_LOOP        ; End-of-Loop test & branch.
                ;
                ; Coefficient Update epilog.
                ;
                CoLOAD      [R9],       R0              ; (ACC) ← h(0,n)
                CoMACM      [R10-QR0],  R11     rnd     ; (ACC) ← h(0,n) - Mu.e(n)*x(n)+
                                                        ; rnd
                                                        ; x(n-1) ← x(n).
                                                        ; (R10) ← (R10)-2*(L-1).
                CoSTORE     [R9-QR0],   MAS             ; h(0,n) ← h(0,n+1).
                                                        ; (R9) ← (R9)-2*(L-1).
```

	Instruction Cycles	Program Words
Read Input samples	2	4
Initialization	1	2
LMS Loop	4L+2(L-2)+1	25
Post/Pre -Processing	9	16
Write Output sample	2	4
Total	4L+2(L-2)+15	51

Note: The branch penalty in the LMS loop is roughly one third of the execution time of the LMS loop. Nevertheless, as shown in Section 5.2.4 - N-real Multiply (windowing), it is possible to minimize the branch penalty by “unrolling” instructions. Therefore, if URF is the UnRolling Factor, the execution times and program words count become respectively:
4L+2(L-2)/URF+1 instruction cycles, and 51 +(URF-1)*2 Program words.

5.5.2 - Extended-Precision LMS Adaptive Filter

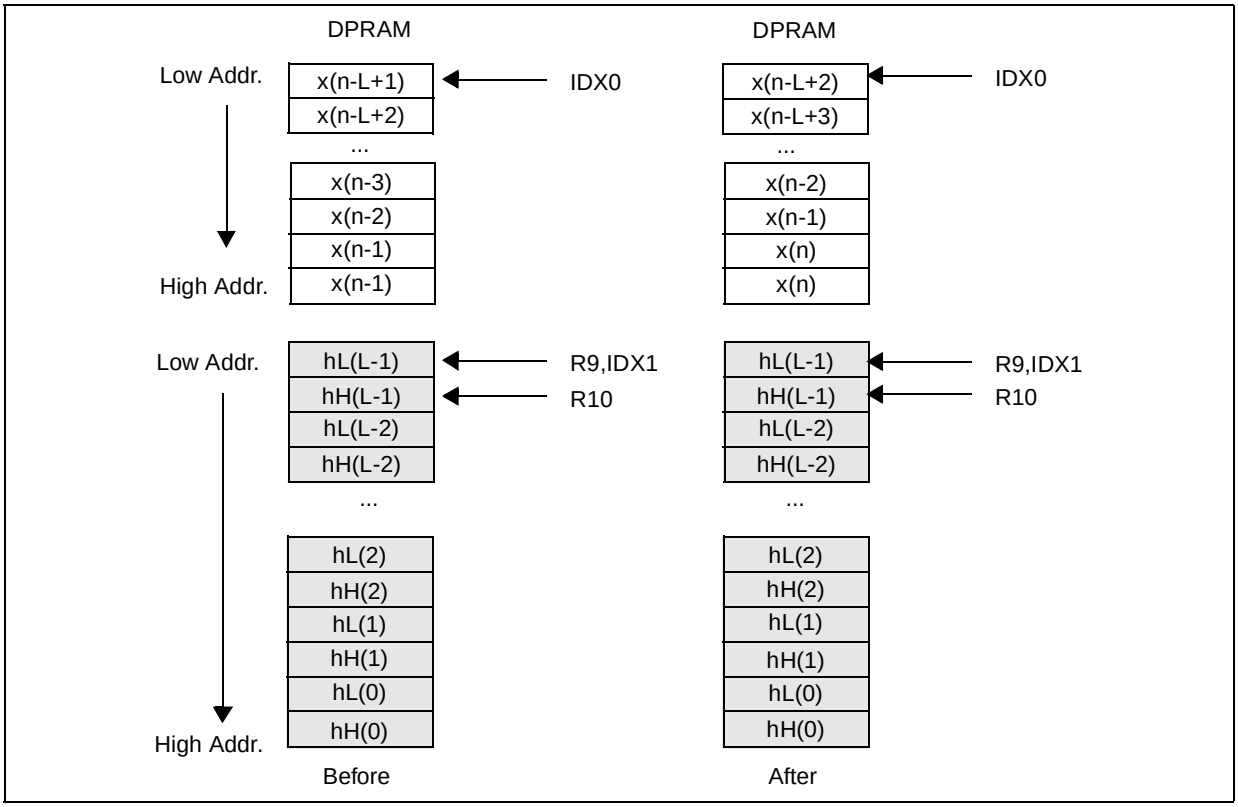
16-bit coefficients can be insufficient for LMS filtering. The following routine describes a LMS filter with 32-bit coefficients and 16-bit samples. In most applications 24-bit coefficients provide good results.

The Extended-precision LMS Adaptive filter uses the same naming convention as the single-precision LMS Adaptive filter: hL(k,n) and hH(k,n) represent the LS word and MS word (respectively) of the k’th coefficient at time n.

For simplicity, you are advised to clear MP of MCW.

Figure 15 shows the corresponding memory map. It is assumed that both coefficients and samples have been initialized by another routine. Note that like the “Single-Precision LMS Adaptive Filter” on page 35, this loop is not “unrolled”.

Figure 15 : Memory Map for Extended Precision LMS Adaptative Filter



This routines assumes that the following general purpose and co-processor registers (SFRs) have been initialized once for ever and that L is less than 31:

– R0 with 0000h
– R1 with the 16-bit MAXimum tolerated value
– R2 contains the 16-bit MINimum tolerated value
– R8 contains the adaptive gain value Mu
– R9 contains the hL(L-1) address
– R10 contains the hH(L-1) address
– IDX0 contains the x(n-L+1) address
– QX0 with 2*(L-1)
– QX1 and QR0 with 4
– QR1 with 4*(L-1)

```
                    ;
                    ; Read the new filter input from a (E)SFR and move it into the DPRAM at
                    ; x(n-1) address therefore overwriting x(n-1).
                    ;
                    MOV         @x(n),          ADC_sfr         ; move the new input x(n)
                    ;
                    ; FIR prolog (LSWs of Impulse response): first multiplication
                    ;
                    CoMULsu     [IDX0+],        [R9+QR0]        ; (ACC) ← hL(L-1)*x(n-L+1)
                                                                ; (IDX0) ← (IDX0)+2,
                                                                ; (R9) ← (R9)+4.
                    ;
                    ; FIR loop (LSWs of Impulse response): Repeat L-2 times the same MAC instruction.
                    ;
REPEAT L-3 TIMES    CoMACsu     [IDX0+],        [R9+QR0]        ; (ACC) ← (ACC)+hL(i)*x(n-i)
                                                                ; (IDX0) ← (IDX0)+2,
                                                                ; (R9) ← (R9)+4.
                    ;
                    ; FIR epilog (LSWs of Impulse response): last MAC instruction and provide
                    ; y(n) in an appropriate format
                    ;
                    CoMACsu     [IDX0-QX1],     [R9-QR1],   rnd ; (ACC) ← (ACC)+hL(0)*x(n)+rnd
                                                                ; & x(n-L+1) ← x(n-L+2),
                                                                ; (IDX0) ← (IDX0)-2*(L-1),
                                                                ; (R9) ← (R9)-2*(2L-1).
                    ;
                    ; Shift
                    ;
                    CoASHR      8,                              ; (ACC)=(ACC)>>8
```

```
                CoASHR      8,                              ; (ACC)=(ACC)>>8
                ;
                ; FIR prolog (MSWs of Impulse response): first multiplication
                ;
                CoMAC       [IDX0+],        [R10+QR0]       ; (ACC) ← hH(L-1)*x(n-L+1)
                                                            ; (IDX0) ← (IDX0)+2,
                                                            ; (R10) ← (R10)+4.
                ;
                ; FIR loop (MSWs of Impulse response): Repeat L-2 times the same MAC instruction.
                ;
REPEAT L-3 TIMES CoMAC      [IDX0+],        [R10+QR0]       ; (ACC) ← (ACC)+hH(i)*x(n-i)
                                                            ; & x(n-i-1) ← x(n-i),
                                                            ; (IDX0) ← (IDX0)+2,
                                                            ; (R10) ← (R10)+4.
                ;
                ; FIR epilog (MSWs of Impulse response): last MAC instruction and provide
                ; y(n) in an appropriate format
                ;
                CoMAC       [IDX0-QX1],     [R10-QR1]       ; (ACC) ← (ACC)+hH(0)*x(n)
                                                            ; & x(n-L+1) ← x(n-L+2),
                                                            ; (IDX0) ← (IDX0)-2*(L-1),
                                                            ; (R10) ← (R10)-2*(2L-1).
                ;
                ; Shift & Rounding
                ;
                CoASHR      #data3,             rnd         ; (ACC) ← (ACC)>>a #data3
                                                            ;+rnd
                ;
                ; Limiting
                ;
                CoMIN       R0,             R1              ; (ACC) ← Min((ACC), MAX).
                CoMAX       R0,             R2              ; (ACC) ← Max((ACC),MIN).
                ;
                ;Write the new filter output y(n) into a (E)SFR.
                ;
                NOP                                         ; Pipeline Effect.
                MOV         DAC_sfr,        MAH             ; move the new output y(n).
                ;
```

```
                ; Read d(n) and move it into a GPR.
                ;
                MOV         R5,         @d(n)           ; (R5) ← d(n)
                ;
                ; Error, e(n), Calculation
                ;
                SUB         R5,         R6              ; (R5) ← d(n)-y(n)=e(n)
                MOV         @e(n),      R5              ; e(n-1) ← e(n)
                CoMUL       R5,         R8              ; (ACC) ← Mu*e(n)
                CoNEG                                   rnd ; (ACC) ← -(ACC)+rnd
                CoSTORE     R11,        MAS             ; (R11) ← -Mu*e(n).
                ;
                ; Coefficients' Updating
                ;
                MOV         R12,        IDX0            ; (R12) ← (IDX0)
                MOV         IDX1,       R9              ; (IDX1) ← (R9)
                MOV         R3,         #L-2            ; (R3) ← L
                ;
                ; Coefficient Update Prolog.
                ;
                CoLOAD      [IDX1+QX1],  [R10-]         ; (ACC) ← h(L-1,n)
                                                        ; (IDX1) ← (IDX1)+4.
                                                        ; (R10) ← (R10)-2.
                CoMAC       R11,         [R12+]         ; (ACC) ← h(L-1,n) -
                                                        ; Mu.e(n)*x(n-L+1)+rnd
                                                        ; (R12) ← (R12)+2.
                CoSTORE     [R10+],      MAL            ; hL(L-1,n) ← hL(L-1,n+1).
                                                        ; (R10) ← (R10)+2.
                CoSTORE     [R10+QR0],   MAH            ; hH(L-1,n) ← hH(L-1,n+1).
                                                        ; (R10) ← (R10)+2.
                ;
                ; Coefficient Update Loop.
                ;
EXT_LMS_LOOP:
                ;
                CoLOAD      [IDX1+QX1],  [R10-]         ; (ACC) ← h(k,n)
                                                        ; (IDX1) ← (IDX1)+4.
                                                        ; (R10) ← (R10)-2.
```

```
CoMACM      R11,        [R12+]          ; (ACC) ← h(k,n) -Mu.e(n)*x(n-k)
                                        ;+rnd
                                        ; x(n-k-1) ← x(n-k).
                                        ; (R12) ← (R12)+2.
CoSTORE     [R10+],     MAL             ; hL(k,n) ← hL(k,n+1).
                                        ; (R10) ← (R10)+2.
CoSTORE     [R10+QR0],  MAH             ; hH(k,n) ← hH(k,n+1).
                                        ; (R10) ← (R10)+2.
;
; End_of_loop Checking.
;
CMPD1       R3          #0h             ; (R3) ← (R3) -1.
JMPR        cc_NZ       EXT_LMS_LOOP    ; End-of-Loop test & branch.
;
; Coefficient Update epilog.
;
CoLOAD      [IDX1+QX1], [R10-]          ; (ACC) ← h(0,n)
                                        ; (IDX1) ← (IDX1)+4.
                                        ; (R10) ← (R10)-2.
CoMACM      R11,        [R12+]          ; (ACC) ← h(0,n) -Mu.e(n)*x(n)
                                        ;+rnd
                                        ; x(n-1) ← x(n).
                                        ; (R12) ← (R12)+2.
CoSTORE     [R10+],     MAL             ; hL(0,n) ← hL(0,n+1).
                                        ; (R10) ← (R10)+2.
CoSTORE     [R10-QR0],  MAH             ; hH(0,n) ← hH(0,n+1).
                                        ; (R10) ← (R10)-(2L-1).
```

	Instruction Cycles	**Program Words**
Read Input samples	2	4
Initialization	4	8
EXT LMS Loop	6L+2(L-2)+3	39
Post/Pre -Processing	9	16
Write Output sample	2	4
Total	6L+2(L-2)+20	71

5.6 - Operations on Tables

ST10-DSP is usefull to compute on tables or collection of samples. Additionally, the concatenation unit can be used to compute on 32-bit operands.

5.6.1 - Detection of the Minimum or Maximum in a Collection of Samples

ST10-DSP instructions (CoMIN and CoMAX) allow to detect the maximum or the minimum between the accumulator and an external operand. Using a hardware loop with pointer auto-modification, the minimum or the maximum of a parameter in a collection of samples can be detected at a rate of 1 clock per sample for 16-bit operands and for 32-bit operands.

5.6.2 - Computing the Sum of a Collection of Samples

Using the repeated instruction CoADD, ST10-DSP allows to compute the sum of a collection of samples at a rate of 1 clock per sample for 16-bit operand and for 32-bit operands.

5.6.3 - Search for an Element Within a Collection of Samples

Using the repeated instruction CoCMP, ST10-DSP allows to compare an external operand with the accumulator at a rate of 1 clock per sample for both 16-bits and 32-bits operands.

The following paragraphs will show 3 examples of table oriented routines.

5.6.4 - Table Move

This routine moves a table of L 16-bit data items from one memory location to another (where L is the number of data items). “Orig_Address” is the location of the first element of the table and “Dest_Address” is its location after the table move.

```
                    ; MAC dedicated registers' initialization:
                    ;
                    MOV         MRW,        #L-1            ; (MRW) ← L-1.
                    MOV         IDX0,       #Dst_Address    ; (IDX0) ← Dst_Address.
                    ;
                    ; GPR initialization:
                    MOV         R1,         #Orig_Address   ; (R1) ← Orig_Address
                    ;
                    ; Move the table
REPEAT MRW TIMES    CoMOV       [IDX0+],    [R1+]           ; ((IDX0)) ← ((R1))
                                                            ; (IDX0) ← (IDX0)+2
                                                            ; (R1) ← (R1)+2.
```

	Instruction Cycles	Program Words
Total	L+3	8

5.6.5 - Find the index of a Maximum Value in a Table

This routine finds the index of the maximum value of data x(i) for i=1 to L, contained in a table. The first element of the index is located at “Orig_Address”. The operation is performed in two steps, the maximum value is detected, and then the corresponding index is detected. At the end of the routine, the maximum value is stored in the co-processor accumulator and the index is stored in R1 (GPR).

```
                ; MAC dedicated registers' initialization:
                MOV             MRW,    #L-1            ; (MRW) ← L-1.
                ;
                ; Accumulator Initialization.
                MOV             MAH,    #FFFFh          ; (MAH) ← FFFFh,
                                                        ; (MAE) ← FFh,
                                                        ; (MAL) ← 0000h.
                MOV             MAL,    #FFFFh          ; (MAL) ← FFFFh,
                ;
                ; GPRs initialization:
                MOV             R0,     #0000h          ; (R0) ← 0000h
                MOV             R1,     #Orig_Address   ; (R1) ← Orig_Address
                ;
                ; First Iteration: Detection of the maximum value
REPEAT MRW TIMES CoMAX          R0,      [R1+]          ; (ACC) ← Max((ACC),x(i))
                                                        ; (R1) ← (R1)+2
                ;
                ; Re-initialization:
                MOV             MRW,    #L-1            ; (MRW) ← L-1.
                MOV             R1,     #Orig_Address   ; (R1) ← Orig_Address
                ;
                ; Second Iteration: Detection of the corresponding index
INDEX_LOOP      CoCMP            R0     [R1+]           ; (MSW) ← (ACC)-x(i)
                                                        ; (R1) ← (R1)+2
                NOP                                     ; Pipieline effect
                JNB             MSW.9,  INDEX_LOOP      ; loop if MSW-Z flag not set
                ;
                ;Store result
                SUB             R1,     #2              ; R1 is the index of the
                                                        ; Maximum value.
```

	Instruction Cycles	Program Words
Total	3L/2+10 [1]	21

Note: 1. On average.

5.6.6 - Compare for Search

This routine finds the index of the first piece of data in a table which matches a specified condition “cc_cond” when compared to the contents of the accumulator. It assumes that data is stored in numerical order in the table. The same assumptions are made as for Section 5.6.4 - and Section 5.6.5 -. When a match is made, the index is stored in R1.

```
                ; Initialization:
                ;
                MOV         MRW,    #L-1                ; (MRW) ← L-1.
                MOV         R0,     #0000h              ; (R0) ← 0000h
                MOV         R1,     #Orig_Address       ; (R1) ← Orig_Address
                ;
                ; Accumulator Initialization.
                MOV         MAH,    #data16             ; (MAH) ← #data16,
                                                        ; (MAE) ← 8 times (MAH15),
                                                        ; (MAL) ← 0000h.
                ;
                ; Second Iteration: Detection of the corresponding index
NO_MATCH        CoCMP       cc_GT   R0      [R1+]       ; (MSW) ← (ACC)-x(i)
                                                        ; (R1) ← (R1)+2
                NOP                                     ; pipeline effect
                JNB         MSW.12, NO_MATCH            ; test C-flag of MSW and
                                                        ; jump if no match
                ;
                ; Storing index
                SUB         R1,     #2                  ; R1 is the index of the
                                                        ; matching element
```

	Instruction Cycles	Program Words
Total	L/2+9 [1]	13

Note: 1. On average.

5.7 - Summary of Routines

Table 2 : Summary of routines

		Instruction Cycles	Program Words	Page Number
Co-Processor Initialization		10	19	10
Mathematics	32 by 32 signed multiplication	12	24	10
	Nth Order Power Series	9N/2+3	22	11
	[NxN][Nx1] Matrix Multiply	N^2+5N+5	24	13
	N-Real Multiply (Windowing)	2.N+3.N/URF [1]	5+4.URF	15
DSP Routines [2]	32x16 L-tap FIR	2L+3	18	17
	DF1 [3] Nth Order IIR filter	2N	10	20
	DF2 [4] Nth Order IIR filter	2N+1	12	23
	DF2 N-cascaded Biquads	11N-1	19	27
	TF [5] N-cascaded Biquads	14N-15	24	31
	16x16 L-tap LMS	4L+2(L-2)/URF +1	51+2(URF-1)	35
	32x16 L-tap LMS	6L+2(L-2)/URF +20	71+2(URF-1)	39
Operations on Tables	Table Move (L items)	L+3	8	44
	Find the Index of a Maximum Value in a table (L items)	3L/2+10 [6]	21	45
	“Compare For Search” [7] (L items)	L/2+7 [8]	13	46

Notes: 1. “URF” stands for “UnRolling Factor”.

2. Representative part of the routine only.

3. Direct Form 1.

4. Direct Form 2.

5. Transpose Form.

6. On average.

7. First data in a table that matches a specified condition.

8. On average.

6 - APPLICATION NOTE VERSION INFORMATION

This document has been released one the 6th of November 2001.

littlediode
COMPONENTS FOR ALL

Littlediode supplies new, hard to find or obsolete electronic components and semiconductors all over the world.

With over two million different components listed you are sure to find the part you need.

Feel free to visit us today at our online store:

LittleDiode.com

Looking forward to providing you with the best possible service.