Intel® IXP400 Software: Intel XScale® Microarchitecture Multiply Accumulate Instructions — FIR / IIR Filters and FFT Examples

Application Note

October 2004

Document Number: 302142-001

Revision History

Date	Revision	Description
October 2004	001	Initial release.

Contents

1.0 Introduction 7
1.1 FIR Filter 7
1.2 IIR Filter 7
1.3 Fast Fourier Transform 7
1.4 Related Documents 8
1.5 Acronyms 8
2.0 Intel XScale® Microarchitecture and Multiply Accumulate DSP Instructions Description 8
2.1 DSP – MAC Instructions Overview 8
2.2 DSP Coprocessor 0 (CP0) 9
2.2.1 Multiply With Internal Accumulate Format 9
2.2.2 Internal Accumulator Access Format 12
3.0 FIR Filter Example 14
3.1 Filter Description 14
3.2 Testing Function – testFIR() 15
3.2.1 FIR Testing Results 15
3.2.2 FIR ARM* ASM Code Using DSP Coprocessor 16
3.2.3 FIR ARM*ASM Code Without DSP Coprocessor 16
3.2.4 FIR Straight C Code Without DSP Coprocessor 16
3.2.5 FIR Initialization 17
4.0 IIR Filter Example 17
4.1 IIR Filter Description 17
4.2 Testing Function TESTIIR() 17
4.2.1 IIR Testing Results 17
4.2.2 IIR – ARM* ASM Code Using DSP Coprocessor 19
4.2.3 IIR – ARM ASM Code without DSP Coprocessor 19
4.2.4 IIR – Straight C Code Without DSP Coprocessor 19
5.0 FFT Example 20
5.1 FFT Description – Split-Radix FFT Implementation on Intel® IXP425 Network Processor 20
5.1.1 FFT Formula Details 20
5.2 Implementation 21
5.2.1 FFT Results 21
6.0 Source Code Examples 24
6.1 FIR Filter 24
6.2 IIR Filter Source Code 37
6.3 FFT Source Code Example 59

Figures

1 Sine Waveform 22
2 FFT of the Sine Wave 22

3 FIR Filter Coded in C Language 24
4 FIR Filter Example — Optimized Using MAC Instructions 31
5 IIR Filter Example, C Code 37
6 IIR Filter Example, Assembly Code 47
7 FFT Example, C Code 59
8 FFT Example, Assembly Code 78

Tables

1 Multiply with Internal Accumulate Format 9
2 MIA{<cond>} acc0, Rm, Rs 10
3 MIAPH{<cond>} acc0, Rm, Rs 11
4 MIAxy{<cond>} acc0, Rm, Rs 12
5 Internal Accumulator Access Format 13
6 MRA{<cond>} RdLo, RdHi, acc0 14

This page is intentionally left blank.

1.0 Introduction

This application note provides source code examples of several typical algorithm functions used for signal processing, including a Finite Impulse Response (FIR) filter, an Infinite Impulse Response (IIR) Filter, and a Fast Fourier Transform (FFT). The FIR and IIR Filter examples show the performance advantages of using the Multiply Accumulate (MAC) instructions shown in the optimized examples of each filter. The FFT example illustrates an optimized assembly language example, but the MAC instructions of the Intel XScale® Core do not help the FFT algorithm to execute any faster due to the particular nature of how the Fast Fourier function operates internally. With study, a programmer can understand how these signal-processing algorithms can be implemented for high performance. A brief description of each function follows.

1.1 FIR Filter

A FIR filter can be described as a discrete linear time-invariant system with output based upon the weighted summation of a finite number of past inputs. FIR filters do not use feedback, so for a FIR filter with *N* coefficients, the output always becomes zero after putting in *N* samples of an impulse response.

This document first presents an example FIR filter written in Assembly Language. Then an optimized FIR filter is presented, also coded in Assembly Language, but employing Intel XScale core instruction set digital signal processing (DSP) opcodes for much higher efficiency — 16x improvement in data throughput over C Language, and 1.5x improvement over standard ARM* Assembly Language. The optimized example also includes a baseline version of the filter in C Language.

1.2 IIR Filter

The impulse response of the IIR Filter is ‘infinite’ because there is feedback in the filter; if you put in an impulse (a single ‘1’ sample followed by many ‘0’ samples), theoretically an infinite number of non-zero values will be output.

As with the FIR example, this document presents an example IIR filter written in Assembly Language. Then an optimized IIR filter is presented, also coded in Assembly Language, but employing Intel XScale core instruction set digital signal processing (DSP) opcodes for much higher efficiency — with improvements in data throughput over C Language, and over standard ARM* Assembly Language. The optimized example also includes a baseline version of the IIR filter in C Language.

1.3 Fast Fourier Transform

In this report, a optimized Split-Radix Fast Fourier Transform (FFT) algorithm is implemented and described.

Following the format used to describe the FIR and IIR filter examples, the FFT section shows an example FFT function written in Assembly Language. This particular example is the highest-performance version in the set of FFT examples. The next example of a coded FFT filter is presented, also in Assembly Language, but employing Intel XScale core instruction set digital

signal processing (DSP) opcodes. As it turns out, for the FFT algorithm, the MAC instructions do not increase performance, for reasons described in the details of the FFT section. The optimized example also includes a baseline version of the filter in C Language.

1.4 Related Documents

Document	Document Number
Intel® IXP400 Digital Signal Processing (DSP) Software Version 2.4 Programmer's Guide	252725_v2_4.pdf
Intel® IXP400 Digital Signal Processing (DSP) Software: Voice Over Internet Protocol Application Note	300320

1.5 Acronyms

Acronym	Description
ANSI	American National Standards Institute
ARM*	ARM* Ltd. [company]
ASM	Assembly [Language]
FIR	Finite Impulse Response
DSP	Digital Signal Processing
MAC	Multiply Accumulate Instruction

2.0 Intel XScale® Microarchitecture and Multiply Accumulate DSP Instructions Description

The Intel XScale® Microarchitecture features a special set of instructions that enable a software programmer to implement signal-processing algorithms that deliver high efficiency and performance. Many types of functions, such as digital filters, use a sum-of-products computation whose internal algorithm requirements are efficiently handled by the Intel XScale core MAC instruction set of the Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor. The FIR and IIR filter examples, in particular, use these instruction enhancements to good advantage in their respective highest-performance versions of sample code.

2.1 DSP – MAC Instructions Overview

Sixteen-bit integers, the data type normally associated with audio signal processing, are used by the Multiply Accumulate instructions. The following sections show the Intel XScale core MAC instructions available to the programmer.

2.2 DSP Coprocessor 0 (CP0)

The Intel XScale core adds a DSP coprocessor to the architecture for the purpose of increasing the performance and the precision of audio processing algorithms. This coprocessor contains a 40-bit accumulator and eight new instructions.

The 40-bit accumulator is referenced by several new instructions that were added to the architecture; **MIA**, **MIAPH** and **MIAxy** are multiply/accumulate instructions that reference the 40-bit accumulator instead of a register specified accumulator. **MAR** and **MRA** provide the ability to read and write the 40-bit accumulator.

Access to CP0 is always allowed in all processor modes when bit 0 of the Coprocessor Access Register is set. Any access to CP0 when this bit is clear will cause an undefined exception. Note that only privileged software can set this bit in the Coprocessor Access Register located in CP15.

The 40-bit accumulator will need to be saved on a context switch if multiple processes are using it.

Two new instruction formats were added for coprocessor 0: Multiply with Internal Accumulate Format and Internal Accumulate Access Format. The formats and instructions are described next.

2.2.1 Multiply With Internal Accumulate Format

A new multiply format has been created to define operations on 40-bit accumulators. Table 1 shows the layout of the new format. The op code for this format lies within the coprocessor register transfer instruction type. These instructions have their own syntax.

Table 1. Multiply with Internal Accumulate Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	0	1	0	opcode_3				Rs				0	0	0	0	acc			1	Rm			

Bits	Description	Notes
31:28	**cond** - ARM condition codes	-
19:16	**opcode_3** - specifies the type of multiply with internal accumulate	Intel XScale core defines the following: 0b0000 = **MIA** 0b1000 = **MIAPH** 0b1100 = **MIABB** 0b1101 = **MIABT** 0b1110 = **MIATB** 0b1111 = **MIATT** The effect of all other encodings are unpredictable.
15:12	**Rs** - Multiplier	
7:5	**acc** - select 1 of 8 accumulators	Intel XScale core only implements acc0; access to any other acc has unpredictable effect.
3:0	**Rm** - Multiplicand	-

Two new fields were created for this format, *acc* and *opcode_3*. The *acc* field specifies one of eight internal accumulators to operate on and *opcode_3* defines the operation for this format. The Intel XScale core defines a single 40-bit accumulator referred to as *acc0;* future implementations may define multiple internal accumulators. The Intel XScale core uses opcode_3 to define six instructions, **MIA**, **MIAPH**, **MIABB, MIABT, MIATB and MIATT.**

Table 2. MIA{<cond>} acc0, Rm, Rs

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	0	1	0	0	0	0	0	Rs				0	0	0	0	0	0	0	1	Rm			

```
Operation:  if ConditionPassed(<cond>) then
                acc0 = (Rm[31:0] * Rs[31:0])[39:0] + acc0[39:0]
Exceptions: none
Qualifiers  Condition Code
            No condition code flags are updated
Notes:      Early termination is supported. Instruction timings can be found
            Specifying R15 for register Rs or Rm has unpredictable results.
            acc0 is defined to be 0b000 on Intel XScale core.
```

The **MIA** instruction operates similarly to **MLA** except that the 40-bit accumulator is used. **MIA** multiplies the signed value in register Rs (multiplier) by the signed value in register Rm (multiplicand) and then adds the result to the 40-bit accumulator (acc0).

MIA does not support unsigned multiplication; all values in Rs and Rm will be interpreted as signed data values. **MIA** is useful for operating on signed 16-bit data that was loaded into a general purpose register by **LDRSH**.

The instruction is only executed if the condition specified in the instruction matches the condition code status.

Table 3. MIAPH{<cond>} acc0, Rm, Rs

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	0	1	0	1	0	0	0	Rs				0	0	0	0	0	0	0	1	Rm			

```
Operation:  if ConditionPassed(<cond>) then
                acc0 = sign_extend(Rm[31:16] * Rs[31:16]) +
                       sign_extend(Rm[15:0] * Rs[15:0]) +
                       acc0[39:0]
Exceptions: none
Qualifiers  Condition Code
            S bit is always cleared; no condition code flags are updated
Notes:      Instruction timings can be found
            Specifying R15 for register Rs or Rm has unpredictable results.
            acc0 is defined to be 0b000 on Intel XScale core
```

The **MIAPH** instruction performs two16-bit signed multiplies on packed half word data and accumulates these to a single 40-bit accumulator. The first signed multiplication is performed on the lower 16 bits of the value in register Rs with the lower 16 bits of the value in register Rm.

The second signed multiplication is performed on the upper 16 bits of the value in register Rs with the upper 16 bits of the value in register Rm. Both signed 32-bit products are sign extended and then added to the value in the 40-bit accumulator (acc0).

The instruction is only executed if the condition specified in the instruction matches the condition code status.

Table 4. MIAxy{<cond>} acc0, Rm, Rs

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	0	1	0	1	1	x	y	Rs				0	0	0	0	0	0	0	1	Rm			

```
Operation: if ConditionPassed(<cond>) then
               if (bit[17] == 0)
                   <operand1> = Rm[15:0]
               else
                   <operand1> = Rm[31:16]

               if (bit[16] == 0)
                   <operand2> = Rs[15:0]
               else
                   <operand2> = Rs[31:16]

               acc0[39:0] = sign_extend(<operand1> * <operand2>) + acc0[39:0]

Exceptions:none
Qualifiers Condition Code
           S bit is always cleared; no condition code flags are updated
Notes:     Instruction timings can be found
           Specifying R15 for register Rs or Rm has unpredictable results.
           acc0 is defined to be 0b000 on Intel XScale core.
```

The **MIAxy** instruction performs one16-bit signed multiply and accumulates these to a single 40-bit accumulator. **x** refers to either the upper half or lower half of register Rm (multiplicand) and **y** refers to the upper or lower half of Rs (multiplier). A value of 0x1 will select bits [31:16] of the register which is specified in the mnemonic as T (for top). A value of 0x0 will select bits [15:0] of the register which is specified in the mnemonic as B (for bottom).

MIAxy does not support unsigned multiplication; all values in Rs and Rm will be interpreted as signed data values.

The instruction is only executed if the condition specified in the instruction matches the condition code status.

2.2.2 Internal Accumulator Access Format

The Intel XScale core defines a new instruction format for accessing internal accumulators in CP0. Table 5, “Internal Accumulator Access Format” on page 13 shows that the op code falls into the coprocessor register transfer space.

The *RdHi* and *RdLo* fields allow up to 64 bits of data transfer between ARM registers and an internal accumulator. The *acc* field specifies 1 of 8 internal accumulators to transfer data to/from. The Intel XScale core implements a single 40-bit accumulator referred to as acc0; future implementations can specify multiple internal accumulators of varying sizes, up to 64 bits.

Access to the internal accumulator is allowed in all processor modes (user and privileged) as long bit 0 of the Coprocessor Access Register is set.

The Intel® IXP42X product line implements two instructions **MAR** and **MRA** that move two ARM registers to acc0 and move acc0 to two ARM registers, respectively.

Table 5. Internal Accumulator Access Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	0	0	1	0	L	RdHi				RdLo				0	0	0	0	0	0	0	0	0	acc		

Bits	Description	Notes
31:28	**cond** - ARM condition codes	-
20	**L** - move to/from internal accumulator 0= move to internal accumulator (MAR) 1= move from internal accumulator (MRA)	-
19:16	**RdHi** - specifies the high order eight (39:32) bits of the internal accumulator.	On a read of the acc, this 8-bit high order field will be sign extended. On a write to the acc, the lower 8 bits of this register will be written to acc[39:32]
15:12	**RdLo** - specifies the low order 32 bits of the internal accumulator	-
7:4	Should be zero	
3	Should be zero	-
2:0	**acc** - specifies 1 of 8 internal accumulators	Intel XScale core only implements acc0; access to any other acc is unpredictable

Note: **MAR** has the same encoding as **MCRR** (to coprocessor 0) and **MRA** has the same encoding as **MRRC** (to coprocessor 0). These instructions move 64-bits of data to/from ARM registers from/to coprocessor registers. **MCRR** and **MRRC** are defined in ARM's DSP instruction set.

Disassemblers not aware of **MAR** and **MRA** will produce the following syntax:

```
MCRR{<cond>} p0, 0x0, RdLo, RdHi, c0
```

The MAR instruction moves the value in register RdLo to bits[31:0] of the 40-bit accumulator (acc0) and moves bits[7:0] of the value in register RdHi into bits[39:32] of acc0.

The instruction is only executed if the condition specified in the instruction matches the condition code status.

This instruction executes in any processor mode.

Table 6. MRA{<cond>} RdLo, RdHi, acc0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	0	0	1	0	1	RdHi				RdLo				0	0	0	0	0	0	0	0	0	0	0	0

```
Operation: if ConditionPassed(<cond>) then
               RdHi[31:0] = sign_extend(acc0[39:32])
               RdLo[31:0] = acc0[31:0]
Exceptions:none
Qualifiers Condition Code
           No condition code flags are updated
Notes:     Instruction timings can be found in
           Specifying the same register for RdHi and RdLo has unpredictable
           results.
           Specifying R15 as either RdHi or RdLo has unpredictable results.
```

The MRA instruction moves the 40-bit accumulator value (acc0) into two registers. Bits[31:0] of the value in acc0 are moved into the register RdLo. Bits[39:32] of the value in acc0 are sign extended to 32 bits and moved into the register RdHi.

The instruction is only executed if the condition specified in the instruction matches the condition code status.

This instruction executes in any processor mode.

3.0 FIR Filter Example

The main body of the FIR filter source code example is coded in high-level C Language. Programming signal processing algorithms in C Language is not efficient; therefore, this implementation is the lowest-performance version in the example. This FIR exercise employs three variations of the filter function code produced for comparison: ARM* / Intel XScale core Assembly Language implementation, ARM* / Intel XScale core Assembly Language using the DSP Extension MAC instructions implementation, and a regular ANSI C language implementation.

3.1 Filter Description

The source code for the filter shown in this example is a general-purpose FIR filter, and the chosen filter coefficients illustrated in this example is a square-root-raised cosine function. This is a commonly used matched filter in communication systems, offering the best signal-to-noise ratio for processing. The table of coefficients that comprise the primary characteristics of the FIR filter example could be changed for other types of filters. For example, by changing the coefficients table, a programmer could turn this filter into a low-pass filter, high-pass filter, or a band-pass filter.

3.2 Testing Function – testFIR()

The testing function testFIR() calls function profile() to run each filter implementation 100 times to measure performance. The performance data is gathered and printed during the running of each routine, and this can then be used for comparing each implementation of the FIR filter.

testFIR() also calls testSequentialBlockProcessing(), which shows how each filter function can be called to process input data sequentially block by block.

The data output of each filter is also compared to make sure the results match.

3.2.1 FIR Testing Results

Each of the versions of the FIR filter shown in this example will display performance data (output results are shown in the following examples). This test was run using a 533-MHz Intel® IXP425 Network Processor.

3.2.1.1 FIR ASM Code Using DSP Coprocessor

total cycles = 2289074

filter order = 63

number of Run = 100

number of output per Run = 128

average cycle per tap = 2.838633

3.2.1.2 FIR ASM Code Without DSP Coprocessor

total cycles = 3603796

filter order = 63

number of Run = 100

number of output per Run = 128

average cycle per tap = 4.468993

3.2.1.3 FIR C Code

total cycles = 36442336

filter order = 63

number of Run = 100

number of output per Run = 128

average cycle per tap = 45.191389

3.2.2 FIR ARM* ASM Code Using DSP Coprocessor

The function prototype for this version of the FIR filter is called `real_FIR_asm`, and is coded in ARM Assembly Language. This version uses the DSP coprocessor to execute multiply accumulate instructions in the function. Thanks to the efficiency of the DSP instructions, and the pipelined architecture of the MAC instruction execution unit in the Intel XScale core, this version of the filter will deliver the highest performance of the three examples, and is approximately 16x faster than the C Language version filter. Further, this version is 1.5x faster than the ARM ASM version that does not use the DSP coprocessor instructions.

```
int real_FIR_asm(short *x,short *h,short *y,short L,short M,short
N)
```

For efficiency, this function processes eight filter coefficients in each pass of the inner loop. If the original filter length *M* is not divisible by 8, zeros must be added to pad the end of the filter coefficients. The new filter length *L* is divisible by 8. It is possible to modify the code so that fewer coefficients are processed in the inner loop and hence fewer zeros are needed to pad the end.

The function calculates two filter outputs in each pass of the outer loop. Because of the requirement of address alignment for 32-bit data access, the original filter coefficients are rearranged. The new filter coefficients *h* has two *L* components. The first *L* components are used to calculate the even-number indexed filter outputs, while the second *L* components are used to calculate the odd-number indexed filter outputs. For the first *L* components of *h*, the algorithm starts with the *M* original filter coefficients, followed by *L*-*M* zeros. For the second *L* components of *h*, it starts with one 0, then *M* original filter coefficients, followed by *L*-*M*-1 zeros. Note that there is one sample time offset between the first *L* components and the second *L* components in *h*.

Rearranging and padding zeros to the original filter coefficients are done in the initialization function *InitiFIR()* in Section 3.2.5.

3.2.3 FIR ARM*ASM Code Without DSP Coprocessor

The function prototype for this version of the FIR filter is called `Fir_noCopro`, is coded in ARM Assembly Language, and does not use the DSP coprocessor. This version of the filter will be the second highest performer — approximately 10x faster than the C Language version filter.

```
int Fir_noCopro(short *x, short *h, short *y, short L, short N);
```

This function processes two filter coefficients in each pass of the inner loop. A zero is needed to pad in the end of the filter coefficients if the filter length *L* is an odd number. This is done in the initialization function *InitiFIR()* in Section 3.2.5.

3.2.4 FIR Straight C Code Without DSP Coprocessor

The function prototype for this version of the FIR filter is called `real_FIR`, is coded in generic ANSI C Language, and does not use the DSP coprocessor. This version of the filter is (by far) the lowest performer — many magnitudes slower than the optimized versions.

```
real_FIR(x, RaisedCos,FIR_output3, FIR_length, N)
```

This is mainly used to verify the numerical result of the assembly code implementations.

3.2.5 FIR Initialization

This function allocates memory for the filter state and filter input. It also rearranges the filter coefficients for the assembly function `real_FIR_asm()` using the DSP coprocessor instructions as explained in Section 3.2.2.

void InitiFIR(short *coef, short M, short N, short **h, short **s, short **x, short *L);

The complete source code for the FIR filter implementations described above is provided in Figure 3 and Figure 4. The C Language mainline for the testFIR() routine that calls each of the FIR filter examples, plus related routines and the C Language-version of the FIR filter, is in the first section shown in Figure 3. The Assembly Language-version implementations of the filter function follow in Figure 4.

4.0 IIR Filter Example

The main body of the IIR filter source code example is coded in high-level C Language. This implementation is the lowest-performance version in the examples. This IIR exercise again employs three variations of the filter function code produced for comparison: ARM* / Intel XScale core Assembly Language implementation, ARM* / Intel XScale core Assembly Language using the DSP Extension MAC instructions implementation, and a regular ANSI C Language implementation.

4.1 IIR Filter Description

The source code for the filter shown in this example is a general-purpose Infinite Impulse Response filter, the IIR filter uses feedback in calculation. and the chosen filter coefficients illustrated in this example is similar to the FIR example, but in this instance is a square-root-raised cosine function with a beta of 0.15.

4.2 Testing Function TESTIIR()

The testing function testIIR() calls function profile() to run each filter implementation 100 times to measure performance. The performance data is gathered and printed during the running of each routine, and this can then be used for comparing each implementation of the FIR filter.

testFIR() also calls testSequentialBlockProcessing(), which shows how each filter function can be called to process input data sequentially block by block. The data output of each filter is also compared to make sure the results match.

4.2.1 IIR Testing Results

Each of the versions of the IIR filter shown in this example will display performance data (output results are shown in the following examples). This test was run using a 533-MHz Intel® IXP425 Network Processor.

4.2.1.1 IIR – ASM Code Using DSP Instructions, M & N Must be Divisible by 4

Function: IIR_asm_DSP4

total cycles =1275698

number Of Run =100

number Of output per Run =100

order M =16

order N =16

average cycle per tap (totalCycles/(M+N)) =3.986556

4.2.1.2 IIR – ASM Code Using DSP Instructions, M & N Must be Even Numbers

Function: IIR_asm_DSP

total cycles =1528704

number Of Run =100

number Of output per Run =100

order M =16

order N =16

average cycle per tap (totalCycles/(M+N)) =4.777200

4.2.1.3 IIR – ASM Code not Using DSP Instructions

Function: IIR_asm (does not use DSP instructions)

total cycles =1749210

number Of Run =100

number Of output per Run =100

order M =16

order N =16

average cycle per tap (totalCycles/(M+N)) =5.466281

4.2.1.4 IIR – Straight C Code, not Using DSP Coprocessor

Function: IIR_C (C implementation)

total cycles =16622907

number Of Run =100

number Of output per Run =100

order M =16

order N =16

average cycle per tap (totalCycles/(M+N)) =51.946584

4.2.2 IIR – ARM* ASM Code Using DSP Coprocessor

The function prototype for this version of the IIR filter is called IIR_asm_DSP, and is coded in ARM Assembly Language. This version uses the DSP coprocessor to execute multiply accumulate instructions in the function. This version of the filter will deliver the highest performance of the four examples, and is approximately 13x faster than the C Language version filter.

Void IIR_arm_DSP4(short *x,short *h,short *y,short L,short M,short N)

IIR_asm_DSP4 use four coefficients in each pass of the inner loop, while IIR_asm_DSP uses only one coefficient in each pass of the inner loop. As a result, the IIR_asm_DSP function has more overhead. That is why IIR_asm_DSP4 is more efficient. IIR_asm_DSP4 is useful for instances where M and N are divisible by 4. Note that if N or M are not even or not divisible by 4, 0 coefficients can always be padded, to make N and M become even or divisible by 4.

4.2.3 IIR – ARM ASM Code without DSP Coprocessor

The function prototype for this version of the IIR filter is called IIR_asm, and is coded in ARM Assembly Language.

```
void IIR_asm(short *a, short *b,  short *w, short *x, short *y, int
N, int M,  int L);
```

4.2.4 IIR – Straight C Code Without DSP Coprocessor

The function prototype for this version of the IIR filter is called IIR_C and is written in generic C Language. The MAC / DSP Instructions are not used in the formula.

```
void IIR_C(short *a, short *b,  short *w, short *x, short *y, int
N, int M,  int L)

Input parameters:

x:  input
w:  state
y:  output
M: order of b
N:  order of a
L:  input block length
```

The complete source code for the IIR filter implementations described above is provided in Chapter 6.0 in Figure 3. and Figure 4. The C Language mainline for the testIIR() routine that calls each of the IIR filter examples, plus related routines and the C Language-version of the IIR filter, is in the first section shown in Figure 3. The Assembly Language-version implementations of the filter function follow in Figure 4.

5.0 FFT Example

The main body of the FFT source code example is coded in high-level C Language, this implementation is, like the FIR and IIR filter examples presented before this section, the lowest-performance version of all the FFT examples.

5.1 FFT Description – Split-Radix FFT Implementation on Intel® IXP425 Network Processor

The IXP425 network processor has sufficient performance to perform computation intensive DSP functions, such as FFT, FIR filters, IIR filters, etc. This example presents an FFT implementation on the IXP425 network processor, which can be used in signal detection and estimation applications.

5.1.1 FFT Formula Details

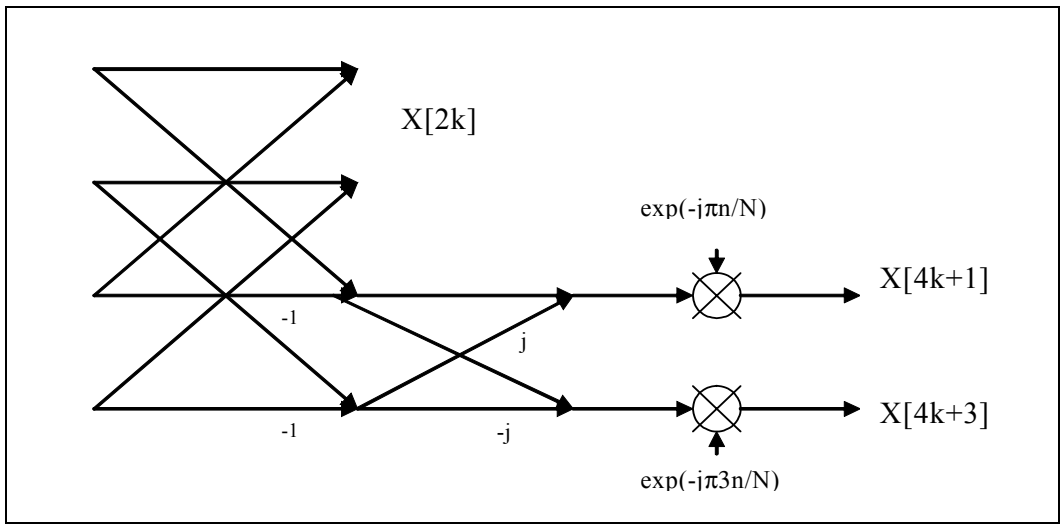
The discrete Fourier transform (DFT) $X[k]$ of a complex sequence $x[n]$ of length N is calculated by:

$$X[k] = \Sigma_{n=0}^{N-1} x[n]*\exp(-j2\pi nk/N) \qquad \text{for } k=0, 1,2\ 0.N-1$$

A direct calculation of N complex values of $X[k]$ will require $4N^2$ multiplications and 4N(N-1) additions given the trigonometric function values. For example, if N=128, 65536 multiplications and 65024 additions are required.

In this report, a Split-Radix FFT algorithm is implemented. This algorithm splits the input sequence x[n] into two subsequences: an even-indexed sequence, and an odd-indexed sequence; then apply radix-2 FFT on the N/2 point even-indexed subsequence sequence, and apply radix-4 FFT on the N/2 point odd-indexed subsequence. This iteration is repeated until finally 2 point FFTs are applied at the final stage.

The fundamental operation of the algorithm is described in the following L-shaped butterfly:



5.2 Implementation

In this report, the implementation consists of:

- C code floating-point direct implementation of the DFT: ***DFT_FloatingPoint_N()***
- C code fixed-point direct implementation of the DFT: ***DFT_FixedPoint_N()***
- C code fixed-point implementation of Split-radix FFT: ***Split_Radix_FFT_C()***
- Assembly code fixed-point implementation of Split-radix FFT using Intel XScale core regular assembly instructions: ***Split_Radix_FFT_asm()***
- Assembly code fixed-point implementation of Split-radix FFT using DSP coprocessor: ***Split_Radix_FFT_asm_DSP()***

The L-shaped butterfly is implemented in:

- Split_Radix_ButterFly_optimized()
- Split_Radix_ButterFly_asm()
- Split_Radix_ButterFly_asm_DSP()

Note that these modules call themselves in the end, iterating through all the stages with different order until 2- or 4-point DFT are performed.

This code applies to complex data with ANY length equal to power of 2.

5.2.1 FFT Results

A sine wave with frequency 343.75 Hz sampled at 8 KHz is used to test the code. The waveform used as input data for processing is shown in Figure 1. Its FFT is shown in Figure 2. Their values are in 16-bit fixed-point format.

Figure 1. Sine Waveform

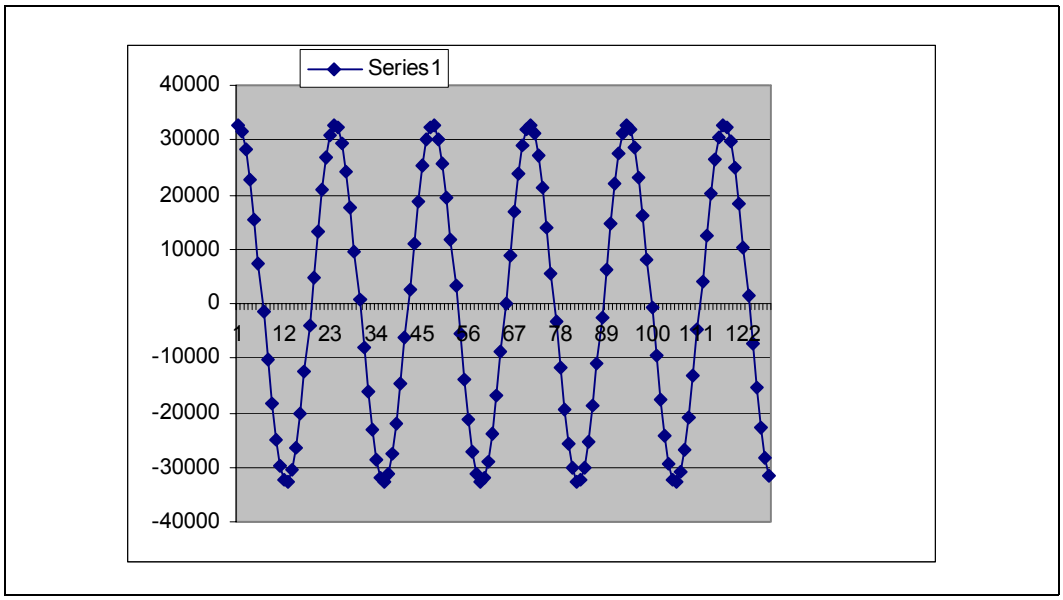
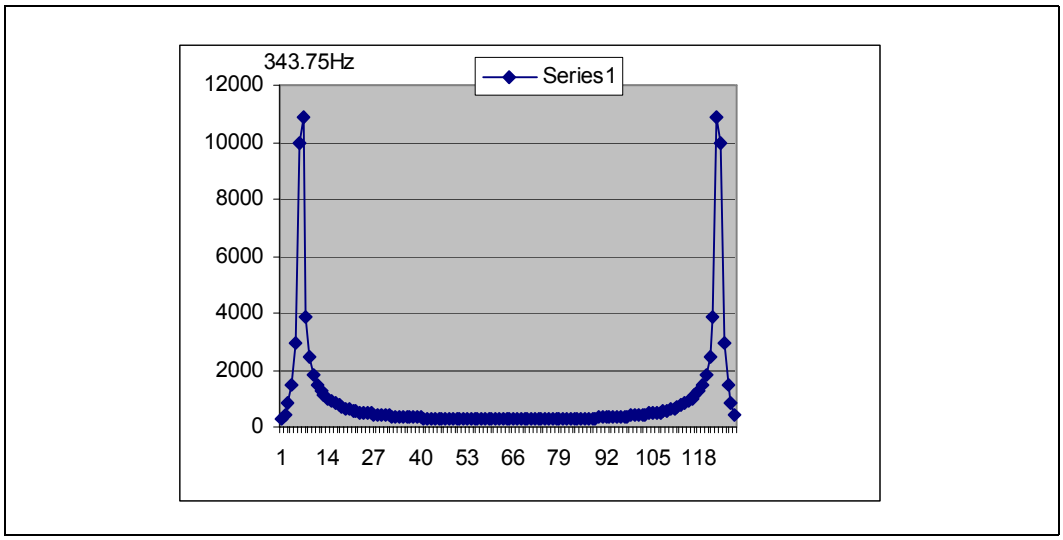


Figure 2. FFT of the Sine Wave



The assembly code `Split_Radix_FFT_asm()` is about 6.5 times faster than the C code implementation `Split_Radix_FFT_C()`. For a 128 point FFT, for example, it takes about 138 cycles to calculate one point output, thus this example is quite efficient.

The C code floating-point and fixed-point implementation `DFT_FloatingPoint_N()` and `DFT_FixedPoint_N()` were used to check if the numerical result is correct. Results of the assembly code match exactly with the result of the C fixed-point code, but has very tiny numerical error compared with the floating-point implementation due to precision in fixed point calculation.

The module `Split_Radix_FFT_asm_DSP()` using Intel XScale core DSP coprocessor instructions is slightly slower then the module `cSplit_Radix_FFT_asm()` using the regular Intel XScale core instructions. It turns out that the ARM / DSP coprocessor instructions are not that useful for FFT because extra cycles are needed to set up the DSP coprocessor for use by the function algorithm.

The following table provide some results for N=128 and N=256. To obtain an average cycle-per-point time, each module is run 100 times.

	`Split_Radix_FFT_C()`	`Split_Radix_FFT_asm()`	`Split_Radix_FFT_asm_DSP()`
N=128 Points			
total cycles	11506273	1767859	1847674
Average cycle per point	898.927578	138.113984	144.349531
N=256 Points			
total cycles	26439362	3972541	4174744
Average cycle per point	1032.78757	155.177383	163.075938

6.0 Source Code Examples

6.1 FIR Filter

Figure 3. FIR Filter Coded in C Language (Sheet 1 of 7)

```
/**
*
* @author Intel Corporation
* @date 17 June 2004
*
*
*
*
* Copyright 2004 Intel Corporation All Rights Reserved.
*
*
* The source code contained or described herein and all documents
* related to the source code ("Material") are owned by Intel Corporation
* or its suppliers or licensors. Title to the Material remains with
* Intel Corporation or its suppliers and licensors. The Material
* contains trade secrets and proprietary and confidential information of
* Intel or its suppliers and licensors. The Material is protected by
* worldwide copyright and trade secret laws and treaty provisions. Except for
* the licensing of the source code hereunder, no part of the Material may be
* used,copied, reproduced, modified, published, uploaded, posted,
* transmitted, distributed, or disclosed in any way without Intel's prior
* express written permission.
*
*
* Except for the licensing of the source code as provided hereunder, no license
* under any patent, copyright, trade secret or other intellectual property
* right is granted to or conferred upon you by disclosure or delivery of the
* Materials, either expressly, by implication, inducement, estoppel or
* otherwise and any license under such intellectual property rights must be
* express and approved by Intel in writing.
*
*
* For further details, please see the file README.TXT distributed with
* this software.
* -- End Intel Copyright Notice --
*/


#include "vxWorks.h"
#include "intLib.h"
#include "errnoLib.h"
#include "errno.h"
#include "stdio.h"
#include "memLib.h"
#include "stdlib.h"

void real_FIR(short *x, short *h, short *y, short L, short N);
extern int real_FIR_asm(short *x, short *h, short *y, short L, short M, short
N);
extern int Fir_noCopro(short *x, short *h, short *y, short L, short N);
```

Figure 3. FIR Filter Coded in C Language (Sheet 2 of 7)

```
void InitiFIR(short *coef, short M, short N, short **h, short **s, short **x,
short *L);

void profile();
void checkDifference(short *y1, short *y2, int N);

extern int writePerfrmCtrl(int x);
extern int readCycleCounter();
long startClock, stopClock;



/* Square root raised cosin function with Beta=0.15 */
short RaisedCos[63]={
     -99,     -17,      91,      84,     -37,    -113,     -31,     112,
     111,     -70,    -206,     -68,     235,     313,     -36,    -479,
    -426,     234,     829,     535,    -578,   -1344,    -633,    1190,
    2191,     709,   -2453,   -4055,    -758,    7325,   15982,
   19704,
   15982,    7325,    -758,   -4055,   -2453,     709,    2191,
    1190,    -633,   -1344,    -578,     535,     829,     234,    -426,
    -479,     -36,     313,     235,     -68,    -206,     -70,     111,
     112,     -31,    -113,     -37,      84,      91,     -17,     -99};

void InitiFIR(short *coef,short M, short N, short **h,
short **s, short **x, short *L)
{

/** Create new filter coefficients suitable for the assembly code
    coef:  the original filter coefficients
    M: the original filter length, it can be any value
    N: max block length for the filter input
    h: the new filter coefficients
    s: array for the filter state and the filter input
    L: new filter length, divisible by 8, 0 are appended to *coef

    h={0, time reverse of *coef, 0,0,0....
        time reverse of *coef, 0, 0,0,0...}

    K=L is divisible by 8 because real_FIR_asm() calculates 8 taps per pass,
    size of h is 2L because odd-index and even-index output will be calculated
    per pass
**/

    int i, K;

    K=M+1; /* to add a 0 to coef for 32 bit data access allignment*/
    K=((K+7)>>3)<<3; /* because 8 tap will be calculated when using DSP co-
    processor subroutine */

    /* printf("K=%d \n", K); */

    /* allocate space for the new filter */
    /* h has the structure:
    {coef[M-1...0], 0, 0,0,    first K elements
     0, coef[M-1...0], 0,0}    second K elements */
```

Figure 3. FIR Filter Coded in C Language (Sheet 3 of 7)

```
    *h=(short *) malloc(2*K*sizeof(short));

    for (i=0; i<M; i++)
    {
        (*h)[i]=coef[M-1-i];
        (*h)[K+1+i]=coef[M-1-i];
    }
    (*h)[M]=0;
    (*h)[K]=0;

    /* pad more 0 to the end */
    for(i=M+1; i<K; i++)
    {
       (*h)[i]=0;
       (*h)[K+i]=0;
    }

    /* allocate space for the filter state and the filter input */

    (*s)=(short *) malloc((K+N)*sizeof(short));
    *x=(*s)+M-1;   /* the filter input follows the filter state */

    /* initialize the initial state */
    for(i=0; i<M-1; i++)
    (*s)[i]=0;
    *L=K;
}


void testSequentialBlockProcessing()
{

    /* N: filter will accept block of maximium N samples each time sequen-
    tially */
    short *h, *s, *xx, L;
    int i;
    short N, FIR_output1[128], FIR_output2[128], FIR_output3[128],
    FIR_output4[128];
    short filterLength;
    filterLength=63;
    N=128;

    /* initialization */

    InitiFIR(RaisedCos, filterLength, N, &h, &s, &xx, &L);
    printf("L=%d \n", L);

    for(i=0; i<N; i++)
        xx[i]=0;

    /* to check result */
    for(i=0; i<filterLength-1; i++) /* initialize the state */
        s[i]=0;

    for(i=0; i<63; i++)
        xx[i]=RaisedCos[i];
```

Figure 3. FIR Filter Coded in C Language (Sheet 4 of 7)

```
    real_FIR_asm(s, h,FIR_output1,  L, filterLength,  N);
    /* call as a single block */
    /* sequential block processing */

    /* test real_FIR_asm() */
    /* first input block */
    for(i=0; i<filterLength-1; i++) /* initialize the state */
       s[i]=0;

    for(i=0; i<63; i++)
       xx[i]=RaisedCos[i];

    real_FIR_asm(s, h,FIR_output2,  L, filterLength, 64);

    /* second input block */
    for(i=0; i<63; i++)
       xx[i]=0;

    real_FIR_asm(s, h,FIR_output2+64, L, filterLength,64);

    /* test real_FIR() */
    /* first input block */
    for(i=0; i<filterLength-1; i++) /* initialize the state */
       s[i]=0;
    for(i=0; i<63; i++)
       xx[i]=RaisedCos[i];

    real_FIR(s, h,FIR_output3,  filterLength, 64);

    /* second input block */
    for(i=0; i<63; i++)
       xx[i]=0;
    real_FIR(s, h,FIR_output3+64,  filterLength, 64);

    /* test Fir_noCopro() */

    /* first input block */
    for(i=0; i<filterLength-1; i++) /* initialize the state */
       s[i]=0;
    for(i=0; i<63; i++)
       xx[i]=RaisedCos[i];
    Fir_noCopro(s, h,FIR_output4,  filterLength, 64);

    /* second input block */
    for(i=0; i<63; i++)
    xx[i]=0;
    Fir_noCopro(s, h,FIR_output4+64,  filterLength, 64);

    printf("check test SequentialBlockProcessing error ....\n");

    checkDifference(FIR_output1, FIR_output2, N);
    checkDifference(FIR_output1, FIR_output3, N);
    checkDifference(FIR_output1, FIR_output4, N);

}
```

Figure 3. FIR Filter Coded in C Language (Sheet 5 of 7)

```
void profile()
{
    int i,j, numberOfRun;
    short FIR_length,N, x[192], FIR_output1[128], FIR_output2[128],
    FIR_output3[128];

    short *h, *s, *xx, L;
    FIR_length=63;
    N=128;
    numberOfRun=100;

    for(i=0; i<64; i++)
    {
        x[i]=0;
        x[i+64]=0;
        x[i+128]=0;
    }

    for(i=0; i<64; i++)
    x[i+64]=RaisedCos[i];

    /* initialization */
    InitiFIR(RaisedCos, FIR_length, N, &h, &s, &xx, &L);

    writePerfrmCtrl(0x07);  /* start all the counters*/

    printf("ASM code using DSP-coprocessor \n");

    startClock=readCycleCounter();
    for(j=0;j<numberOfRun; j++)   /* run 100 times for measurement*/
    real_FIR_asm(x, h,FIR_output1, L, FIR_length, N);
    stopClock=readCycleCounter();

    printf("total cycles =%d \n", stopClock-startClock);
    printf("filter order =%d \n", FIR_length);

    printf("number Of Run =%d \n", numberOfRun);
    printf("number Of output per Run =%d \n", \n", N);
    printf("average cycle per tap =%f \n", \n", (stopClock-start-
    Clock)*1.0/numberOfRun/N/FIR_length);

    printf("ASM code without DSP coprocessor \n");
    startClock=readCycleCounter();
    for(j=0;j<numberOfRun; j++)   /* run 100 times for measurement*/
    Fir_noCopro(x, h,FIR_output2,  FIR_length, N);
    stopClock=readCycleCounter();

    printf("total cycles =%d \n", stopClock-startClock);
    printf("filter order =%d \n", FIR_length);

    printf("number Of Run =%d \n", numberOfRun);
    printf("number Of output per Run =%d \n", \n", N);
    printf("average cycle per tap =%f \n", \n", (stopClock-start-
    Clock)*1.0/numberOfRun/N/FIR_length);

    printf("C code \n");
```

Figure 3. FIR Filter Coded in C Language (Sheet 6 of 7)

```
    startClock=readCycleCounter();
    for(j=0;j<numberOfRun; j++)   /* run 100 times for measurement*/
    real_FIR(    x, h,FIR_output3, FIR_length, N);
    stopClock=readCycleCounter();

    printf("total cycles =%d \n", stopClock-startClock);
    printf("filter order =%d \n", FIR_length);

    printf("number Of Run =%d \n", numberOfRun);
    printf("number Of output per Run =%d \n", \n", N);
    printf("average cycle per tap =%f \n", \n", (stopClock-start-
    Clock)*1.0/numberOfRun/N/FIR_length);


    for(i=0; i<N; i++)
    {

    printf("FIR_output1[%d]=0x%x,FIR_output2[%d]=0x%x,,FIR_output3[%d]=0x%x
        \n",i,FIR_output1[i],i,FIR_output2[i],i,FIR_output3[i]);
     }


    printf("check one block call error.... \n";

    checkDifference(FIR_output1, FIR_output3, N);
    checkDifference(FIR_output2, FIR_output3, N);
}

void testFIR()
{
     profile();
     testSequentialBlockProcessing();
}


void checkDifference(short *y1, short *y2, int N)
{
    int i, j;
    for(j=0, i=0; i<N; i++)
    {

          /* printf("y1[%d]=0x%x,y2[%d]=0x%x \n", i,y1[i],i,y2[i]); */

          if(y1[i]!=y2[i])
          {
                printf("!!!y1[%d]=0x%x,y2[%d]=0x%x \n", i,y1[i],i,y2[i]);

                j++;
          }
    }

    if(j==0)
     printf("   no error \n");

}
```

Figure 3. FIR Filter Coded in C Language (Sheet 7 of 7)

```
void checkDifference(short *y1, short *y2, int N)
{
    int i, j;
    for(j=0, i=0; i<N; i++)
    {

          /* printf("y1[%d]=0x%x,y2[%d]=0x%x \n", i,y1[i],i,y2[i]); */

          if(y1[i]!=y2[i])
          {
                printf("!!!y1[%d]=0x%x,y2[%d]=0x%x \n", i,y1[i],i,y2[i]);

                j++;
          }
    }

    if(j==0)
     printf("   no error \n");

}

void real_FIR(short *x, short *h, short *y, short L, short N)
{
     int i, j, z;

     for (j=0; j<N; j++)
      {
            for (z=0,i=0; i<L; i++)
                z+=x[i+j]*h[i];

            z=z>>15;
                    if (z > 32767) z = 32767;
            else if (z < -32768) z = -32768;

            *y++=(short)z;
            }

     /* update state */

     for (j=0; j<L-1; j++)
     {
           x[j]=x[N+j];
     }
}
```

Figure 4. FIR Filter Example — Optimized Using MAC Instructions (Sheet 1 of 6)

```
@@/**
@@* @author Intel Corporation
@@* @date 17 June 2004
@@*
@@* -- Intel Copyright Notice --
@@*
@@* Copyright 2004 Intel Corporation All Rights Reserved.
@@*
@@* The source code contained or described herein and all documents
@@* related to the source code ("Material") are owned by Intel
@@* Corporation or its suppliers or licensors. Title to the Material
@@* remains with Intel Corporation or its suppliers and licensors.
@@* The Material contains trade secrets and proprietary and confidential
@@* information of Intel or its suppliers and licensors. The Material
@@* is protected by worldwide copyright and trade secret laws and treaty
@@* provisions. Except for the licensing of the source code hereunder,
@@* no part of the Material may be used, copied, reproduced, modified,
@@* published, uploaded, posted, transmitted, distributed, or disclosed
@@* in any way without Intel's prior express written permission.
@@*
@@* Except for the licensing of the source code as provided hereunder,
@@* no license under any patent, copyright, trade secret or other
@@* intellectual property right is granted to or conferred upon you by
@@* disclosure or delivery of the Materials, either expressly, by
@@* implication, inducement, estoppel or otherwise and any license under
@@* such intellectual property rights must be express and approved by
@@* Intel in writing.
@@*
@@* For further details, please see the file README.TXT distributed with
@@* this software.
@@* -- End Intel Copyright Notice -- @@*/

#include <arch/arm/arm.h>

@ real_FIR(short *x, short *h, short *y, short L,, short M, short N)

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@ function parameter:
@@
@@ r0 = x: input pointer
@@ r1 = h: filter coef
@@ r2 = y: Output pointer
@@ r3 = L: filter length
@@ [sp,#40] = M:original filter length
@@ [sp,#44] = N:length of output to calculate
@@
@@ for j=0...N-1
@@ y[j]=sum of {x[i+j]*h[i]} for i=0...L-1
@@ and 0x8000<= y[j] <= 0x7fff
@@
@@ h has the structure:  {h[0...L-2],0, 0, h[0....L-2]}
@@ it is assumed that L is divisible by 8
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Figure 4. FIR Filter Example — Optimized Using MAC Instructions (Sheet 2 of 6)

```
.global _real_FIR_asm
.global real_FIR_asm__FPsN20sss

.global _Fir_noCopro
.global Fir_noCopro__FPsN20ss

.balign 4

_real_FIR_asm:
real_FIR_asm__FPsN20sss:
   stmdb  sp!,{r4-r12,lr}

   ldr r12, L$SAT0x7fff @ r12=0x7fff, used for satuation

   mov r9,  r0 @ x
   mov r10, r1 @ h
   mov r11, r3 @ L

   ldr r4, [sp,#44] @ N
   movs r4, r4, lsr #1 @ N/2
   beq checkNagain

loop1:
   mov r0, r9 @x
   mov r1, r10 @h
   mov r3, r11 @L

   @first output
   sub r5, r5, r5
   mar acc0, r5, r5 @ acc0=0
loop0:
   ldr r5, [r0], #4
   ldr r6, [r1], #4
   ldr r7, [r0], #4
   ldr r8, [r1], #4

   miaph acc0, r6, r5
   miaph acc0, r8, r7

   ldr r5, [r0], #4
   ldr r6, [r1], #4

   ldr r7, [r0], #4
   ldr r8, [r1], #4
   subs r3,r3, #8 @ 8 taps per loop
   miaph acc0, r6, r5
   miaph acc0, r8, r7

   bne loop0

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17
   orr r6, r6, r5, lsr #15 @ acc0>>15
```

Figure 4. FIR Filter Example — Optimized Using MAC Instructions (Sheet 3 of 6)

```
@satuation
   cmp r6, r12 @ compare with 0x7fff
   movgt r6, r12
   mvn r12, r12
   cmp r6, r12 @ compare with 0x8000
   movlt r6, r12
   strh r6, [r2],#+2 @ save the output

   @second output
   mov r0, r9@ x
   mov r3, r11@ L
   sub r5, r5, r5
   mar acc0, r5, r5

loop00:
   ldr r5, [r0], #4
   ldr r6, [r1], #4
   miaph acc0, r6, r5

   ldr r7, [r0], #4
   ldr r8, [r1], #4
   miaph acc0, r8, r7

   ldr r5, [r0], #4
   ldr r6, [r1], #4
   miaph acc0, r6, r5

   ldr r7, [r0], #4
   ldr r8, [r1], #4
   miaph acc0, r8, r7

   subs r3,r3, #8
   bne loop00

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17
   orr r6, r6, r5, lsr #15

@satuation
   cmp r6, r12 @compare with 0x8000
   movlt r6, r12
   mvn r12, r12
   cmp r6, r12 @compare with 0x7fff
   movgt r6, r12

   strh r6, [r2],#+2 @ save the output

   add r9, r9, #4 @ point to the next input
   subs r4,r4, #1
   bne loop1

   @ check if N is an odd number
```

Figure 4. FIR Filter Example — Optimized Using MAC Instructions (Sheet 4 of 6)

```
checkNagain:
   ldr r4, [sp,#44] @ N
   ands r4, r4, #1 @ is N odd ?
   beq doneNow

   mov r0, r9 @x
   mov r1, r10 @h
   mov r3, r11 @L

   @first output
   sub r5, r5, r5
   mar acc0, r5, r5 @ acc0=0

loop000:
   ldr r5, [r0], #4
   ldr r6, [r1], #4
   miaph acc0, r6, r5

   ldr r7, [r0], #4
   ldr r8, [r1], #4
   miaph acc0, r8, r7

   ldr r5, [r0], #4
   ldr r6, [r1], #4
   miaph acc0, r6, r5

   ldr r7, [r0], #4
   ldr r8, [r1], #4
   miaph acc0, r8, r7

   subs r3,r3, #8 @ 8 taps per loop
   bne loop000

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17
   orr r6, r6, r5, lsr #15 @ acc0>>15

   @satuation
   cmp r6, r12 @compare with 0x7fff
   movgt r6, r12
   mvn r12, r12
   cmp r6, r12 @compare with 0x8000
   movlt r6, r12
    strh r6, [r2],#+2 @ save the output
   add r9, r9, #2 @ point to the next input

doneNow:
   ldr r4, [sp,#44] @ N
   ldr r11, [sp,#40] @ M
   sub r2, r9, r4, asl #1 @ r9=r9-2N, point to the begining of x
   sub r11,r11, #1 @ copy L-1 samples

loop80:
   ldrsh   r10, [r9], #+2 @ load it
   strh    r10, [r2], #+2 @ save it
   subs    r11, r11, #1
   bne loop80
   ldmia sp!,{r4-r12,pc}        @ return
```

Figure 4. FIR Filter Example — Optimized Using MAC Instructions (Sheet 5 of 6)

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@ r0 = x: input pointer
@@ r1 = h: filter coef
@@ r2 = y: Output pointer
@@ r3 = L: filter length
@@ [sp,#40] = N:length of output to calculate
@@
@@ if L is an odd number, h[L] has to be a 0
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

.align 4

_Fir_noCopro:
Fir_noCopro__FPsN20ss:
   stmdb   sp!,{r4-r12,lr}      @ push registers

   ldr     r4,[sp,#40]          @ r4=N
   stmdb   sp!,{r3-r4}          @ save L, N
   and r5, r3, #1 @ make L even
   add r3, r3, r5


10:     @ Outer loop
   mov     r5, #0               @ r5
   mov     r6, #0               @ r6

   mov     r7, r0               @ r7 = x
   mov     r8, r1               @ r9 = h
   mov     r9, r3               @ r8 = L

20:     @ inner loop.
        ldrsh       r10, [r7], #+2          @
        ldrsh       r11, [r8], #+2          @
        ldrsh       r12, [r7], #+2          @
        ldrsh       r14, [r8], #+2          @
        subs        r9, r9, #2              @
        smlabb      r5, r10, r11, r5        @
        smlabb      r6, r12, r14, r6        @
        bne         20b
    add     r6,  r6, r5             @ add them

   ldr   r12, L$SAT0x7fff
@ r12=0x7fff, used for satuation

   mov   r6,  r6, asr #15       @ move the value to lower 16-bit
   cmp     r6,  r12 @ compare with 0x7fff
   movgt  r6,  r12
   mvn     r12, r12
   cmp     r6,  r12 @ compare with 0x8000
   movlt  r6,  r12

   subs    r4, r4, #1

   strh    r6, [r2], #+2    @ store the result
   add     r0, r0, #2 @ advance the input pointer
   bne     10b
```

Figure 4. FIR Filter Example — Optimized Using MAC Instructions (Sheet 6 of 6)

```
updateState:
   ldmia    sp!,{r3-r4} @ L, N
   sub     r2, r0, r4, asl #1 @ r2=r0-2N, point to the begining of x
   sub     r3,r3, #1@ copy  L-1 samples

loop81:
   ldrsh    r10, [r0], #+2 @ load it
   strh     r10, [r2], #+2 @ save it
   subs     r3, r3, #1
   bne     loop81

   ldmia    sp!,{r4-r12,pc}       @ return

.align 4
L$SAT0x7fff:          .long    0x07fff
```

6.2 IIR Filter Source Code

Figure 5. IIR Filter Example, C Code (Sheet 1 of 10)

```
@@/**
@@* @author Intel Corporation
@@* @date 17 June 2004
@@*
@@* -- Intel Copyright Notice --
@@*
@@* Copyright 2004 Intel Corporation All Rights Reserved.
@@*
@@* The source code contained or described herein and all documents
@@* related to the source code ("Material") are owned by Intel
@@* Corporation or its suppliers or licensors. Title to the Material
@@* remains with Intel Corporation or its suppliers and licensors.
@@* The Material contains trade secrets and proprietary and confidential
@@* information of Intel or its suppliers and licensors. The Material
@@* is protected by worldwide copyright and trade secret laws and treaty
@@* provisions. Except for the licensing of the source code hereunder,
@@* no part of the Material may be used, copied, reproduced, modified,
@@* published, uploaded, posted, transmitted, distributed, or disclosed
@@* in any way without Intel's prior express written permission.
@@*
@@* Except for the licensing of the source code as provided hereunder,
@@* no license under any patent, copyright, trade secret or other
@@* intellectual property right is granted to or conferred upon you by
@@* disclosure or delivery of the Materials, either expressly, by
@@* implication, inducement, estoppel or otherwise and any license under
@@* such intellectual property rights must be express and approved by
@@* Intel in writing.
@@*
@@* For further details, please see the file README.TXT distributed with
@@* this software.
@@* -- End Intel Copyright Notice -- @@*/

#include "vxWorks.h"
#include "intLib.h"
#include "errnoLib.h"
#include "errno.h"
#include "stdio.h"
#include "math.h"
#include "memLib.h"
#include "stdlib.h"


extern int writePerfrmCtrl(int x);
extern int readCycleCounter();
long startClock, stopClock;
int numberOfRun=100;
int getNum(char *str);
void checkDifference(short *wR, short *vR, int N, bool flag);
void real_FIR(short *x, short *h, short *y, short L, short N);
```

Figure 5. IIR Filter Example, C Code (Sheet 2 of 10)

```
void checkSectionB();
void checkSectionA();
void Profile();

void InitiIIR(short *A, short *B, short M, short N, short L, short **a, short
**b, short **w);

void IIR_C(short *a, short *b,  short *w, short *x, short *y, int N, int M,
int L);
extern int IIR_asm(short *a, short *b,  short *w, short *x, short *y, int N,
int M,  int L);
extern int IIR_asm_DSP(short *a, short *b,  short *w, short *x, short *y, int
N, int M,  int L);
extern int IIR_asm_DSP4(short *a, short *b,  short *w, short *x, short *y, int
N, int M,  int L);

/* Square root raised cosin function with Beta=0.15 */
short RaisedCos[64]={
     -99,     -17,      91,      84,     -37,    -113,     -31,     112,
     111,     -70,    -206,     -68,     235,     313,     -36,    -479,
    -426,     234,     829,     535,    -578,   -1344,    -633,    1190,
    2191,     709,   -2453,   -4055,    -758,    7325,   15982,
   19704,
   15982,    7325,    -758,   -4055,   -2453,     709,    2191,
    1190,    -633,   -1344,    -578,     535,     829,     234,    -426,
    -479,     -36,     313,     235,     -68,    -206,     -70,     111,
     112,     -31,    -113,     -37,      84,      91,     -17,     -99,0};

void testIIR()
{
      printf("check setion A...\n”);
      checkSectionA();

      printf("check setion B...\n”);
      checkSectionB();

      Profile();
}

void Profile()
{
      short *a, *b, *w, *A, *B;
      short *x, *y, M, N, L;
      short *w1, *y1, *w2, *y2, *w3, *y3;
      int i, j;
      float d=0.5;

      numberOfRun=100;

      N=16;
      M=16;
      L=100;
```

Figure 5. IIR Filter Example, C Code (Sheet 3 of 10)

```
A=(short *) malloc(N*sizeof(short));
B=(short *) malloc(M*sizeof(short));

x=(short *) malloc(L*sizeof(short));
y=(short *) malloc(L*sizeof(short));

w1=(short *) malloc((M+L)*sizeof(short));
y1=(short *) malloc(L*sizeof(short));

w2=(short *) malloc((M+L)*sizeof(short));
y2=(short *) malloc(L*sizeof(short));

w3=(short *) malloc((M+L)*sizeof(short));
y3=(short *) malloc(L*sizeof(short));

for(i=0; i<M; i++)
{
   w1[i]=0;
   w2[i]=0;
   w3[i]=0;
}

/* coefficients do not matter for profiling */
for(i=0; i<N; i++)
   A[i]=i;

for(i=0; i<M; i++)
   B[i]=i;

/* input */
for(i=0; i<L; i++)/* input has only one pulse */
x[i]=0;
x[0]=32767;/* just to get impusle response */

InitiIIR(A, B, M, N, L, &a, &b, &w);
/* profile */

writePerfrmCtrl(0x07);  /* start all the counters*/

printf("IIR_asm_DSP \n");

startClock=readCycleCounter();
for(j=0;j<numberOfRun; j++)   /* run 100 times for measurement*/
    IIR_asm_DSP(a, b, w, x, y, N, M,L);
stopClock=readCycleCounter();

printf("total cycles =%d \n",stopClock-startClock);
printf("number Of Run =%d \n", numberOfRun);

printf("number Of output per Run =%d \n", L);
printf("order M =%d \n", M);
printf("order N =%d \n", N);
printf("average cycle per tap (totalCycles/(M+N)) =%f \n, \n",(stop-
Clock-startClock)*1.0/numberOfRun/L/(M+N));
```

Figure 5. IIR Filter Example, C Code (Sheet 4 of 10)

```
    printf("IIR_asm_DSP4 \n");

    startClock=readCycleCounter();
    for(j=0;j<numberOfRun; j++)   /* run 100 times for measurement*/
     IIR_asm_DSP4(a, b, w3, x, y3, N, M,L);

    stopClock=readCycleCounter();

    printf("total cycles =%d \n",stopClock-startClock);

    printf("number Of Run =%d \n", numberOfRun);
    printf("number Of output per Run =%d \n", L);
    printf("order M =%d \n", M);
    printf("order N =%d \n", N);
    printf("average cycle per tap (totalCycles/(M+N)) =%f \n, \n",(stop-
    Clock-startClock)*1.0/numberOfRun/L/(M+N));

    printf("IIR_asm \n");

    startClock=readCycleCounter();
    for(j=0;j<numberOfRun; j++)   /* run 100 times for measurement*/
    IIR_asm(a, b, w1, x, y1, N, M,L);
    stopClock=readCycleCounter();

    printf("total cycles =%d \n",stopClock-startClock);
    printf("number Of Run =%d \n", numberOfRun);

    printf("number Of output per Run =%d \n", L);
    printf("order M =%d \n", M);
    printf("order N =%d \n", N);
    printf("average cycle per tap (totalCycles/(M+N)) =%f \n, \n",(stop-
    Clock-startClock)*1.0/numberOfRun/L/(M+N));

    printf("IIR_C \n");

    startClock=readCycleCounter();
    for(j=0;j<numberOfRun; j++)   /* run 100 times for measurement*/
    IIR_C(a, b, w2, x, y2, N, M,L);
    stopClock=readCycleCounter();

    printf("total cycles =%d \n",stopClock-startClock);

    printf("number Of Run =%d \n", numberOfRun);

    printf("number Of output per Run =%d \n", L);
    printf("order M =%d \n", M);
    printf("order N =%d \n", N);
    printf("average cycle per tap (totalCycles/(M+N)) =%f \n, \n",(stop-
    Clock-startClock)*1.0/numberOfRun/L/(M+N));


    checkDifference(y,y2,L, true);
    checkDifference(y,y3,L, false);
    checkDifference(y1,y2,L, false);
    checkDifference(w,w2,M-1,false);
    checkDifference(w1,w2,M-1,false);
}
```

Figure 5. IIR Filter Example, C Code (Sheet 5 of 10)

```
void checkSectionA()
{
     /* a smooth filter
     y[n]=d*x[n]+(1-d)*y[n-1],   d<1
     impulse response h[n]=d(1-d)^n
     */
     short *a, *b, *w, *A, *B;
     short *x, *y, M, N, L;
     short *w1, *y1, *y2;
     int i, j;
     float d=0.5;

     numberOfRun=100;
     N=2;
     M=2;
     L=9;

     A=(short *) malloc(N*sizeof(short));
     B=(short *) malloc(M*sizeof(short));

     x=(short *) malloc(L*sizeof(short));
     y=(short *) malloc(L*sizeof(short));

     w1=(short *) malloc((M+L)*sizeof(short));
     y1=(short *) malloc(L*sizeof(short));

     y2=(short *) malloc(L*sizeof(short));

     A[0]=0;
     A[1]=(short) (-(1-d)*32767);
     B[0]=(short) (d*32767);
     B[1]=0;

     for(i=0; i<L; i++)/* input has only one pulse */
           x[i]=0;
     x[0]=32767;/* just to get impulse response */

     InitiIIR(A, B, M, N, L, &a, &b, &w);
     IIR_asm_DSP(a, b, w, x, y, N, M,L);


     for(i=0; i<M; i++) w1[i]=0;

           IIR_C(a, b, w1, x, y1, N, M,L);

     /* theoretical result */
     for(i=0; i<L; i++)
           y2[i]=(short)(d*pow(1-d,i)*32767);

     checkDifference(y,y2,L,true);
     checkDifference(y,y1,L,false);
     checkDifference(w,w1,M-1,false);
}
```

Figure 5. IIR Filter Example, C Code (Sheet 6 of 10)

```
void checkSectionB()
{
      short *a, *b, *w, *A, *B;
      short *x, *y, M, N, L;
      short *x1, *y1;
      short *w2, *y2;
      int i, j;

      numberOfRun=100;

      N=64;
      M=64;
      L=127;

      A=(short *) malloc(N*sizeof(short));
      B=(short *) malloc(M*sizeof(short));

      x=(short *) malloc(L*sizeof(short));
      y=(short *) malloc(L*sizeof(short));

      w2=(short *) malloc((L+M)*sizeof(short));
      y2=(short *) malloc(L*sizeof(short));

      /* check section B */
      for(i=0; i<N; i++)
      A[i]=0;/* turn off section A */

      /* input */
      for(i=0; i<L; i++)
            x[i]=0;

      for(i=0; i<M; i++)
            x[i]=RaisedCos[i];

      InitiIIR(A, RaisedCos, M, N, L, &a, &b, &w);
      IIR_asm_DSP(a, b, w, x, y, N, M,L);

      for(i=0; i<M; i++) w2[i]=0;
            IIR_C(a, b, w2, x, y2, N, M,L);

      /* compare Section B with FIR filter */

      x1=(short *) malloc((L+M)*sizeof(short));
      y1=(short *) malloc(L*sizeof(short));

      for(i=0; i<M+L; i++)
            x1[i]=0;
      for(i=0; i<M; i++)
            x1[M-2+i]=RaisedCos[i];
      real_FIR(x1, RaisedCos, y1, M, L);

      checkDifference(y,y1,L,false);
      checkDifference(y,y2,L,false);
      checkDifference(w,w2,N-1,false);
}
```

Figure 5. IIR Filter Example, C Code (Sheet 7 of 10)

```
void checkDifference(short *wR, short *vR, int N, bool flag)
{

     int n, tmp,  maxError, indMaxError;
     printf("    checking the difference...

     for(indMaxError=0, maxError=0, n=0; n<N; n++)
     {
           tmp=abs(wR[n]-vR[n]);

           if(maxError<tmp)
           {
                 maxError=tmp;
                 indMaxError=n;
           }
           if(flag==true)
                  printf("wR[%d]=%d,vR[%d]=%d
     }

     if(maxError==0)
           printf("        no difference

     else
           printf("        max difference maxError=%d, indMaxError=%d
}

int getNum(char *str)
{


     int c;
     int i = 0;
     char input[100];

     if(str && *str) printf("%s", str);
     do
     {
         c = getc(stdin);

         if (c == 0x08)
         {
             if(i) i--;
         }
         else
         {
             input[i++] = c;
         }
     } while(i<100 && c!='

     input[i] = '\0';


return atoi(input);
}
```

Figure 5. IIR Filter Example, C Code (Sheet 8 of 10)

```
void InitiIIR(short *A, short *B, short M, short N, short L, short **a, short
**b, short **w)

{
      /* require M and N be even number!
            A,B:  original  filter coef H[z]=B[z]/A[z]
            M:    order of B
            N:    order of A, N>=M
            L:    max block input length
            a:    new coef
            b:    new coef
            w:    filter state
      */

      /* short *b;   b[    0,     ... M-2 .    M-1]
         ={B[M-1], ... B[1],    B[0] }

         short *a;   a[    0,        N-2,   . N-1]
                          ={-A[N-1],.. -A[1].    0]}
      */

int i, j;
short *q;

*a=(short *) malloc(2*N*sizeof(short));
*b=(short *) malloc((2*M+2)*sizeof(short));
*w=(short *) malloc((N+L)*sizeof(short));

for(i=0; i<N-1; i++)
      (*a)[i]=-A[N-1-i];
(*a)[N-1]=0;

q=(*a)+N;
q[0]=0;

for(i=0; i<N-1; i++)
      q[1+i]=(*a)[i];

for(i=0; i<M; i++)
      (*b)[i]=B[M-1-i];

q=(*b)+M;
q[0]=0;

for(i=0; i<M; i++)
      q[1+i]=(*b)[i];

q[M+1]=0;

for(i=0; i<N; i++)
      (*w)[i]=0;

}
```

Figure 5. IIR Filter Example, C Code (Sheet 9 of 10)

```
void IIR_C(short *a, short *b,  short *w, short *x, short *y, int N, int M,
int L)
{
     /* x:  input
        w:  state
        y:  output
        M:  order of b
        N:  order of a
        L:  input block length
     */

      /* H(z)= {B[0]+B[1]*z[-1]+...+B[M-1]*z[-(M-1)]} /
      {1+A[1]*z[-1]+...+A[N-1]*z[-(N-1)]} */

     /* short *b;   b[    0,     ... M-2 .    M-1]
                 ={B[M-1], ... B[1],    B[0] }
        short *a;   a[    0,        N-2,   . N-1]
                       ={-A[N-1],.. -A[1].    0]}   */
int i, j, v;

for(j=0; j<L; j++)
{
      for(v=0, i=0; i<N-1; i++)
         v+=w[i+j]*a[i];

      v=v>>15;
      v+=x[j];

     if (v > 32767) v = 32767;
           else if (v < -32768) v = -32768;

           w[N-1+j]=v;

      for(v=0, i=0; i<M; i++)
         v+=w[N-M+i+j]*b[i];

      v=v>>15;

      if (v > 32767) v = 32767;
           else if (v < -32768) v = -32768;
     *y++=(short)v;
}

/* update state */

     for (j=0; j<N-1; j++)
     {
           w[j]=w[L+j];
     }
}
```

Figure 5. IIR Filter Example, C Code (Sheet 10 of 10)

```
void real_FIR(short *x, short *h, short *y, short L, short N)
{
      int i, j, z;

      for (j=0; j<N; j++)
      {
            for (z=0,i=0; i<L; i++)
                  z+=x[i+j]*h[i];

            /* printf("z[%d]=0x%x

             z=z>>15;
             if (z > 32767) z = 32767;
                  else if (z < -32768) z = -32768;
                  *y++=(short)z;

      }

      /* update state */

      for (j=0; j<L-1; j++)
      {
            x[j]=x[N+j];
      }
}
```

Figure 6. IIR Filter Example, Assembly Code (Sheet 1 of 12)

```
@@/**************************************************************************
@@* @author Intel Corporation
@@* @date 17 June 2004
@@*
@@* -- Intel Copyright Notice --
@@*
@@* Copyright 2004 Intel Corporation All Rights Reserved.
@@*
@@* The source code contained or described herein and all documents
@@* related to the source code ("Material") are owned by Intel
@@* Corporation or its suppliers or licensors. Title to the Material
@@* remains with Intel Corporation or its suppliers and licensors.
@@* The Material contains trade secrets and proprietary and confidential
@@* information of Intel or its suppliers and licensors. The Material
@@* is protected by worldwide copyright and trade secret laws and treaty
@@* provisions. Except for the licensing of the source code hereunder,
@@* no part of the Material may be used, copied, reproduced, modified,
@@* published, uploaded, posted, transmitted, distributed, or disclosed
@@* in any way without Intel's prior express written permission.
@@*
@@* Except for the licensing of the source code as provided hereunder,
@@* no license under any patent, copyright, trade secret or other
@@* intellectual property right is granted to or conferred upon you by
@@* disclosure or delivery of the Materials, either expressly, by
@@* implication, inducement, estoppel or otherwise and any license under
@@* such intellectual property rights must be express and approved by
@@* Intel in writing.
@@*
@@* For further details, please see the file README.TXT distributed with
@@* this software.
@@* -- End Intel Copyright Notice -- @@
@@*************************************************************************/

@ 2 point, 4 point, and fisrt pass in the second loop are treated specially
@ becuase no multiplication required

#include <arch/arm/arm.h>

@ extern void IIR_asm(short *a, short *b,  short *w, short *x, short *y, int
N, int M,  int L);

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@ IIR filter H(z)=B(z)/A(z)
@@ function parameter:
@@
@@ r0 = a: coefficient pointer
@@ r1 = b: coefficient pointer
@@ r2 = w: state pointer
@@ r3 = x: input pointer
@@ [sp,#40] = y:output pointer
@@ [sp,#44] = Norder of a
@@ [sp,#48] = Morder of b
@@ [sp,#52] = Linput length
@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Figure 6. IIR Filter Example, Assembly Code (Sheet 2 of 12)

```
.global IIR_ASM
.global _IIR_ASM
.global IIR_asm__FPsN40iii

.global IIR_ASM_DSP
.global _IIR_ASM_DSP
.global IIR_asm_DSP__FPsN40iii

.global IIR_ASM_DSP4
.global _IIR_ASM_DSP4
.global IIR_asm_DSP4__FPsN40iii

.balign 4

IIR_ASM_DSP:
_IIR_ASM_DSP:
IIR_asm_DSP__FPsN40iii:

   stmdb  sp!,{r4-r12,lr}

   ldr   r12, L$SAT0x7fff@ r12=0x7fff, used for satuation

   ldr    r4,  [sp,#40]         @ r4 =y


   ldrh   r5,  [sp,#54]         @ r5 = L
   strh   r5,  [sp,#52] @ save L to higher 16 bits
   movs   r5,  r5, lsr #1 @ L/2
   strh   r5,  [sp,#52]
   beq checkNagain

loop0:
   @ ---- first sample  -----
   @ A section

   mov    r7, r0               @ r7 =a
   mov    r9, r2               @ r9 =w
   ldr    r8, [sp,#44]         @ r8 =N

   sub    r5, r5, r5
   mar    acc0, r5, r5@ acc0=0
loop1:    @ inner loop.
   ldr r10, [r7], #+4
   ldr r11, [r9], #+4
   subs 8, r8, #2
   miaph   acc0, r11, r10
   bne         loop1

   ldrsh  r10,  [r3], #+2        @ x[j]

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17
   orr r6, r6, r5, lsr #15 @ acc0>>15


   add    r6,  r6, r10          @ add x[j]
```

Figure 6. IIR Filter Example, Assembly Code (Sheet 3 of 12)

```
@satuation
   cmp    r6,  r12 @ compare with 0x7fff
   movgt  r6,  r12
   mvn    r12, r12
   cmp    r6,  r12 @ compare with 0x8000
   movlt  r6,  r12

   strh   r6,  [r9,#-2] @ save state

   @ B section
   @ r9 pointed to w[N]
   mov    r7, r1               @ r7 =b
   ldr    r8, [sp,#48]         @ r8 =M
   sub    r5, r5, r5
   mar    acc0, r5, r5       @ acc0=0
   sub  r9, r9, r8, lsl #1   @ w[N-M]

loop2:   @ inner loop.
   ldr         r10, [r7], #+4
   ldr r11, [r9], #+4
   subs        r8, r8, #2
   miaph   acc0, r11, r10
   bne         loop2

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17
   orr r6, r6, r5, lsr #15 @ acc0>>15

  @satuation
   cmp r6, r12 @compare with 0x8000

   movlt r6, r12
mvn r12, r12
   cmp r6, r12 @compare with 0x7fff
   movgt r6, r12

   strh   r6, [r4], #+2         @ store the result


   @ ---- second sample  -----
   @ A section

   ldr    r8, [sp,#44]         @ r8 =N
   mov    r7, r0               @ r7 =a
   mov    r9, r2               @ r9 =w
   add    r7, r7, r8, lsl #1 @ point to one sample offset a coefficients

   sub    r5, r5, r5
   mar    acc0, r5, r5@ acc0=0

loop3:    @ inner loop.
   ldr         r10, [r7], #+4
   ldr         r11, [r9], #+4
   subs        r8, r8, #2
   miaph   acc0, r11, r10

   bne         loop3
```

Figure 6. IIR Filter Example, Assembly Code (Sheet 4 of 12)

```
ldrsh  r10,  [r3], #+2        @ x[j]

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17
   orr r6, r6, r5, lsr #15 @ acc0>>15

   add    r6,  r6, r10          @ add x[j]

  @satuation
   cmp    r6,  r12 @ compare with 0x7fff
   movgt  r6,  r12
   mvn    r12, r12
   cmp    r6,  r12 @ compare with 0x8000
   movlt  r6,  r12
   strh   r6,  [r9] @ save state

   @ B section
   @ r9 pointed to w[N]

   mov    r7, r1               @ r7 =b
   ldr    r8, [sp,#48]         @ r8 =M
   sub    r5, r5, r5
   mar    acc0, r5, r5       @ acc0=0
   sub  r9, r9, r8, lsl #1   @ w[N-M]
   add    r7, r7, r8, lsl #1   @ point to one sample offset a coefficients

loop4:   @ inner loop.
   ldr         r10, [r7], #+4
   ldr         r11, [r9], #+4
   subs        r8, r8, #2
   miaph   acc0, r11, r10
   bne         loop4

       @ one more for the offset
   ldr         r10, [r7], #+4
   ldr         r11, [r9], #+4
   miaph   acc0, r11, r10

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17
   orr r6, r6, r5, lsr #15 @ acc0>>15

  @satuation
   cmp r6, r12 @compare with 0x8000
   movlt r6, r12
   mvn r12, r12
   cmp r6, r12 @compare with 0x7fff
   movgt r6, r12

   strh   r6, [r4], #+2         @ store the result

   ldrsh  r8, [sp,#52]          @ r8 =L
   subs   r8, r8, #1
   strh   r8, [sp,#52]          @ r8 =L
   add    r2, r2, #4        @ advance the state pointer
   bne    loop0
```

Figure 6. IIR Filter Example, Assembly Code (Sheet 5 of 12)

```
checkNagain:  @ one output only

   ldrh r8, [sp,#54] @ L
   ands r8, r8, #1@ is L odd ?
   beq updateState

   @ ---- first sample  -----
   @ A section

   mov    r7, r0                @ r7 =a
   mov    r9, r2                @ r9 =w
   ldr    r8, [sp,#44]          @ r8 =N

   sub    r5, r5, r5
   mar    acc0, r5, r5@ acc0=0

loop51:    @ inner loop.
   ldr         r10, [r7], #+4
   ldr         r11, [r9], #+4
   subs        r8, r8, #2
   miaph   acc0, r11, r10
   bne         loop51

   ldrsh  r10,  [r3], #+2        @ x[j]

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17
   orr r6, r6, r5, lsr #15 @ acc0>>15

   add    r6,  r6, r10          @ add x[j]

   @satuation
   cmp    r6,  r12 @ compare with 0x7fff
   movgt  r6,  r12
   mvn    r12, r12
   cmp    r6,  r12 @ compare with 0x8000
   movlt  r6,  r12
   strh   r6,  [r9,#-2] @ save state

   @ B section
   @ r9 pointed to w[N]

   mov    r7, r1                @ r7 =b
   ldr    r8, [sp,#48]          @ r8 =M
   sub    r5, r5, r5
   mar    acc0, r5, r5       @ acc0=0
   sub  r9, r9, r8, lsl #1   @ w[N-M]

loop52:   @ inner loop.
   ldr         r10, [r7], #+4
   ldr         r11, [r9], #+4
   subs        r8, r8, #2
   miaph   acc0, r11, r10
   bne         loop52

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17
   orr r6, r6, r5, lsr #15 @ acc0>>15
```

Figure 6. IIR Filter Example, Assembly Code (Sheet 6 of 12)

```
  @satuation
   cmp r6, r12 @compare with 0x8000
   movlt r6, r12
   mvn r12, r12
   cmp r6, r12 @compare with 0x7fff
   movgt r6, r12

strh   r6, [r4], #+2          @ store the result

   add    r2, r2, #2         @ advance the state pointer

updateState:
   ldrh   r5, [sp,#54]           @ r5 = L
   sub   r9, r2, r5, asl #1 @ r2=r0-2L, point to the begining of x
   ldr    r8, [sp,#44]           @ r8 =N
   sub   r8, r8, #1@ copy  N-1 samples
loop5:
   ldrsh  r10, [r2], #+2 @ load it
   strh   r10, [r9], #+2 @ save it
   subs   r8, r8, #1
   bne   loop5

doneNow:

   ldmia sp!,{r4-r12,pc}         @ return


.balign 4

IIR_ASM:
_IIR_ASM:
IIR_asm__FPsN40iii:

   stmdb  sp!,{r4-r12,lr}

   ldr    r4,   [sp,#40]          @ r4 =y
   ldr    r5,   [sp,#52]          @ r5 = L
   orr   r5,  r5, r5, lsl #16  @ save L to higher 16 bits
str    r5,  [sp,#52]
loop10:
   @ A section
   mov    r7, r0                @ r7 =a
   mov    r9, r2                @ r9 =w
   ldr    r8, [sp,#44]          @ r8 =N
   mov    r5, #0                @ r5 =0
   mov    r6, #0                @ r6 =0

loop11:    @ inner loop.
   ldrsh       r10, [r7], #+2
   ldrsh       r11, [r9], #+2
   ldrsh       r12, [r7], #+2
   ldrsh       r14, [r9], #+2
   subs        r8, r8, #2
   smlabb      r5, r10, r11, r5
   smlabb      r6, r12, r14, r6

   bne         loop1
```

Figure 6. IIR Filter Example, Assembly Code (Sheet 7 of 12)

```
   add    r6,  r6, r5           @ add them
   ldrsh  r5,  [r3], #+2        @ x[j]

   ldr   r12, L$SAT0x7fff @ r12=0x7fff, used for satuation
   mov   r6,  r6, asr #15 @ move the value to lower 16-bit

   add    r6,  r6, r5           @ add x[j]

   cmp    r6,  r12@ compare with 0x7fff
   movgt  r6,  r12
   mvn    r12, r12
   cmp    r6,  r12@ compare with 0x8000
   movlt  r6,  r12

   strh   r6,  [r9,#-2]@ save state

   @ B section
   @ r9 pointed to w[N]

   mov    r7, r1               @ r7 =b
   ldr    r8, [sp,#48]         @ r8 =M
   mov    r5, #0               @ r5 =0
   mov    r6, #0               @ r6 =0
   sub  r9, r9, r8, lsl #1   @ w[N-M]

loop12:   @ inner loop.
   ldrsh       r10, [r7], #+2
   ldrsh       r11, [r9], #+2
   ldrsh       r12, [r7], #+2
   ldrsh       r14, [r9], #+2
   subs        r8, r8, #2
   smlabb      r5, r10, r11, r5
   smlabb      r6, r12, r14, r6
   bne         loop12

   add    r6,  r6, r5           @ add them

   ldr   r12, L$SAT0x7fff@ r12=0x7fff, used for satuation
   mov   r6,  r6, asr #15      @ move the value to lower 16-bit

   cmp    r6,  r12@ compare with 0x7fff
   movgt  r6,  r12
   mvn    r12, r12
   cmp    r6,  r12@ compare with 0x8000
   movlt  r6,  r12


   ldrsh  r8, [sp,#52]          @ r8 =L
   strh   r6, [r4], #+2         @ store the result

   subs   r8, r8, #1
   strh   r8, [sp,#52]          @ r8 =L
  add    r2, r2, #2        @ advance the state pointer

   bne    loop10
```

Figure 6. IIR Filter Example, Assembly Code (Sheet 8 of 12)

```
updateState1:

   ldrh   r5, [sp,#54]          @ r5 = L
   sub   r9, r2, r5, asl #1 @ r2=r0-2L, point to the begining of x
   ldr    r8, [sp,#44]          @ r8 =N
   sub   r8, r8, #1 @ copy  N-1 samples
loop13:
   ldrsh  r10, [r2], #+2 @ load it
   strh   r10, [r9], #+2 @ save it
   subs   r8, r8, #1
   bne   loop13


doneNow1:

   ldmia sp!,{r4-r12,pc}        @ return


.balign 4

IIR_ASM_DSP4:
_IIR_ASM_DSP4:
IIR_asm_DSP4__FPsN40iii:

   stmdb  sp!,{r4-r12,lr}

   ldr   r12, L$SAT0x7fff@ r12=0x7fff, used for satuation

   ldr    r4,  [sp,#40]         @ r4 =y


   ldrh   r5,  [sp,#54]         @ r5 = L
   strh   r5,  [sp,#52]@ save L to higher 16 bits
   movs   r5,  r5, lsr #1@ L/2
   strh   r5,  [sp,#52]
   beq checkNagain4


loop40:
   @ ---- first sample  -----
   @ A section
   mov    r7, r0               @ r7 =a
   mov    r9, r2               @ r9 =w
   ldr    r8, [sp,#44]         @ r8 =N

   sub    r5, r5, r5
   mar    acc0, r5, r5@ acc0=0
loop41:    @ inner loop.
   ldr         r10, [r7], #+4
   ldr         r11, [r9], #+4
   ldr         r5, [r7], #+4
   ldr r6, [r9], #+4
   subs        r8, r8, #4
   miaph   acc0, r11, r10
   miaph   acc0, r5, r6

   bne         loop41
```

Figure 6. IIR Filter Example, Assembly Code (Sheet 9 of 12)

```
ldrsh  r10,  [r3], #+2        @ x[j]

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17
   orr r6, r6, r5, lsr #15 @ acc0>>15

   add    r6,  r6, r10          @ add x[j]

   @satuation
   cmp    r6,  r12 @ compare with 0x7fff
   movgt  r6,  r12
   mvn    r12, r12
   cmp    r6,  r12 @ compare with 0x8000
   movlt  r6,  r12

   strh   r6,  [r9,#-2] @ save state

   @ B section
   @ r9 pointed to w[N]
   mov    r7, r1               @ r7 =b
   ldr    r8, [sp,#48]         @ r8 =M
   sub    r5, r5, r5
   mar    acc0, r5, r5       @ acc0=0
   sub  r9, r9, r8, lsl #1   @ w[N-M]

loop42:   @ inner loop.
   ldr         r10, [r7], #+4
   ldr         r11, [r9], #+4
   ldr         r5, [r7], #+4
   ldr         r6, [r9], #+4
   subs        r8, r8, #4
   miaph   acc0, r11, r10
   miaph   acc0, r5, r6
   bne         loop42

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17

   orr r6, r6, r5, lsr #15 @ acc0>>15

  @satuation
   cmp r6, r12 @compare with 0x8000
   movlt r6, r12
   mvn r12, r12
   cmp r6, r12 @compare with 0x7fff
   movgt r6, r12
   strh   r6, [r4], #+2         @ store the result

   @ ---- second sample  -----
   @ A section

   ldr    r8, [sp,#44]         @ r8 =N
   mov    r7, r0               @ r7 =a
   mov    r9, r2               @ r9 =w
   add    r7, r7, r8, lsl #1   @ point to one sample offset a coefficients

   sub    r5, r5, r5
   mar    acc0, r5, r5 @ acc0=0
```

Figure 6. IIR Filter Example, Assembly Code (Sheet 10 of 12)

```
loop43:    @ inner loop.
   ldr          r10, [r7], #+4
   ldr          r11, [r9], #+4
   ldr          r5, [r7], #+4
   ldr          r6, [r9], #+4
   subs         r8, r8, #4
   miaph    acc0, r11, r10
   miaph    acc0, r5, r6
   bne          loop43

   ldrsh  r10,  [r3], #+2        @ x[j]

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17

   orr r6, r6, r5, lsr #15 @ acc0>>15
   add    r6,  r6, r10          @ add x[j]


  @satuation
   cmp    r6,  r12 @ compare with 0x7fff
   movgt  r6,  r12
   mvn    r12, r12
   cmp    r6,  r12 @ compare with 0x8000
   movlt  r6,  r12

   strh   r6,  [r9] @ save state

   @ B section
   @ r9 pointed to w[N]

   mov    r7, r1               @ r7 =b
   ldr    r8, [sp,#48]         @ r8 =M
   sub    r5, r5, r5
   mar    acc0, r5, r5       @ acc0=0
   sub  r9, r9, r8, lsl #1   @ w[N-M]
   add    r7, r7, r8, lsl #1   @ point to one sample offset a coefficients

loop44:   @ inner loop.
   ldr          r10, [r7], #+4
   ldr          r11, [r9], #+4
   ldr          r5, [r7], #+4
   ldr          r6, [r9], #+4
   subs         r8, r8, #4
   miaph    acc0, r11, r10
   miaph    acc0, r5, r6
   bne          loop44

   @ one more for the offset

   ldr          r10, [r7], #+4
   ldr          r11, [r9], #+4
   miaph    acc0, r11, r10

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17
   orr r6, r6, r5, lsr #15 @ acc0>>15
```

Figure 6. IIR Filter Example, Assembly Code (Sheet 11 of 12)

```
@satuation
   cmp r6, r12 @compare with 0x8000
   movlt r6, r12
   mvn r12, r12
   cmp r6, r12 @compare with 0x7fff
   movgt r6, r12

   strh   r6, [r4], #+2          @ store the result


   ldrsh  r8, [sp,#52]           @ r8 =L
   subs   r8, r8, #1
   strh   r8, [sp,#52]           @ r8 =L
   add    r2, r2, #4         @ advance the state pointer
   bne    loop40


checkNagain4:  @ one output only

   ldrh r8, [sp,#54] @ L
   ands r8, r8, #1 @ is L odd ?
   beq updateState4

   @ ---- first sample  -----
   @ A section

   mov    r7, r0                @ r7 =a
   mov    r9, r2                @ r9 =w
   ldr    r8, [sp,#44]          @ r8 =N

   sub    r5, r5, r5
   mar    acc0, r5, r5@ acc0=0
loop451:    @ inner loop.
   ldr         r10, [r7], #+4
   ldr         r11, [r9], #+4
   ldr         r5, [r7], #+4
   ldr         r6, [r9], #+4
   subs        r8, r8, #4
   miaph   acc0, r11, r10
   miaph   acc0, r5, r6
   bne         loop451

   ldrsh  r10,  [r3], #+2        @ x[j]

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17
   orr r6, r6, r5, lsr #15 @ acc0>>15

   add    r6,  r6, r10           @ add x[j]

   @satuation

   cmp    r6,  r12 @ compare with 0x7fff
   movgt  r6,  r12
   mvn    r12, r12
   cmp    r6,  r12 @ compare with 0x8000
   movlt  r6,  r12
  strh   r6,  [r9,#-2] @ save state
```

Figure 6. IIR Filter Example, Assembly Code (Sheet 12 of 12)

```
   @ B section
   @ r9 pointed to w[N]

   mov    r7, r1                @ r7 =b
   ldr    r8, [sp,#48]          @ r8 =M
   sub    r5, r5, r5
   mar    acc0, r5, r5        @ acc0=0
   sub  r9, r9, r8, lsl #1    @ w[N-M]

loop452:   @ inner loop.
   ldr         r10, [r7], #+4
   ldr         r11, [r9], #+4
   ldr         r5, [r7], #+4
   ldr         r6, [r9], #+4
   subs        r8, r8, #4
   miaph   acc0, r11, r10
   miaph   acc0, r5, r6
   bne         loop452

   mra r5, r6, acc0    @ acc0=[r6 r5]
   mov r6, r6, asl #17
   orr r6, r6, r5, lsr #15 @ acc0>>15


  @satuation

   cmp r6, r12 @compare with 0x8000
   movlt r6, r12
   mvn r12, r12
   cmp r6, r12 @compare with 0x7fff
   movgt r6, r12

   strh   r6, [r4], #+2         @ store the result

   add    r2, r2, #2        @ advance the state pointer
updateState4:
   ldrh   r5, [sp,#54]          @ r5 = L
   sub   r9, r2, r5, asl #1 @ r2=r0-2L, point to the begining of x
   ldr    r8, [sp,#44]          @ r8 =N
   sub   r8, r8, #1 @ copy  N-1 samples
loop45:
   ldrsh  r10, [r2], #+2 @ load it
   strh   r10, [r9], #+2 @ save it
   subs   r8, r8, #1
   bne   loop45


doneNow4:
   ldmia sp!,{r4-r12,pc}        @ return



.align 4
L$SAT0x7fff:         .long   0x07fff
```

6.3 FFT Source Code Example

Figure 7. FFT Example, C Code (Sheet 1 of 20)

```
/**
*
* @author Intel Corporation
* @date 17 June 2004
*
*
* -- Intel Copyright Notice --
*
*
* Copyright 2004 Intel Corporation All Rights Reserved.
*
* The source code contained or described herein and all documents
* related to the source code ("Material") are owned by Intel Corporation
* or its suppliers or licensors. Title to the Material remains with
* Intel Corporation or its suppliers and licensors. The Material
* contains trade secrets and proprietary and confidential information of
* Intel or its suppliers and licensors. The Material is protected by
* worldwide copyright and trade secret laws and treaty provisions. Except
* for the licensing of the source code hereunder, no part of the Material may
* be used, copied, reproduced, modified, published, uploaded, posted,
* transmitted, distributed, or disclosed in any way without Intel's prior
* express written permission.
*
* Except for the licensing of the source code as provided hereunder, no
* license under any patent, copyright, trade secret or other intellectual
* property right is granted to or conferred upon you by disclosure or
* delivery of the Materials, either expressly, by implication, inducement,
* estoppel or otherwise and any license under such intellectual property
* rights must be express and approved by Intel in writing.
*
* For further details, please see the file README.TXT distributed with
* this software.
* -- End Intel Copyright Notice --
*/


#include "vxWorks.h"
#include "intLib.h"
#include "errnoLib.h"
#include "errno.h"
#include "stdio.h"
#include "math.h"
#include "memLib.h"
#include "stdlib.h"
```

Figure 7. FFT Example, C Code (Sheet 2 of 20)

```
extern int writePerfrmCtrl(int x);
extern int readCycleCounter();
long startClock, stopClock;
int numberOfRun=2;

void Profile_Split_Radix_FFT_C(short *xR, short *xI, int N);
void Profile_Split_Radix_FFT_asm(short *xR, short *xI, int N);
void Profile_Split_Radix_FFT_asm_DSP(short *xR, short *xI, int N);

/* All FFT and DFT outputs are scaled down by N */

int bitReverse(int b, int B);
void checkDifference(short *wR, short *wI, short *vR, short *vI, int N);

void DFT_FloatingPoint_N(short *xR, short *xI, float *yR, float *yI, int N);
void DFT_FixedPoint_N(short *xR, short *xI, short *yR, short *yI, int N);
void Split_Radix_FFT_C(short *xR, short *xI, int N);
void Split_Radix_ButterFly_Optimized(short *xR, short *xI, int M, int q);
void Split_Radix_ButterFly(short *xR, short *xI, int M, int q);
void Split_Radix_ButterFly_with_rouding(short *xR, short *xI, int M, int q);
void Split_Radix_FFT_asm(short *xR, short *xI, int N);
void Split_Radix_FFT_asm_DSP(short *xR, short *xI, int N);

extern void Split_Radix_FFT_ASM(short *xR, short *xI,short *cosRsinI, int N,
short *bitRevTable, int H);
extern void Split_Radix_FFT_ASM_DSP(short *xR, short *xI,short *cosRsinI, int
N, short *bitRevTable, int H);

float pi;
short *cosR, *sinI;

void DFT_FloatingPoint_N(short *xR, short *xI, float *yR, float *yI, int N)
{
     /* the outputs are scaled down by N */
     int k, n;

     for (k=0; k<N; k++)
     {
           for(yR[k]=0, yI[k]=0, n=0; n<N; n++)
           {
                 yR[k]+=xR[n]*cos(2*pi*k*n/N) + xI[n]*sin(2*pi*k*n/N);
                 yI[k]+=xI[n]*cos(2*pi*k*n/N) - xR[n]*sin(2*pi*k*n/N);
           }
           yR[k]=yR[k]/N;
           yI[k]=yI[k]/N;
     }
}

void DFT_FixedPoint_N(short *xR, short *xI, short *yR, short *yI, int N)
{ /* the result is divided by N */

     int k, n, B;
     long long tmpR, tmpI;
     cosR=(short *) malloc(N*sizeof(short));
     sinI=(short *) malloc(N*sizeof(short));
```

Figure 7. FFT Example, C Code (Sheet 3 of 20)

```
      B=(int)(log10(N)/log10(2));

      /* printf("B=%d \n",B); */


      for(n=0; n<N; n++)
      {
            cosR[n]=(short) (cos(2*pi*n/N)*32767);
            sinI[n]=(short) (sin(2*pi*n/N)*32767);
      }

      for (k=0; k<N; k++)
      {
             for(tmpR=0,tmpI=0, n=0; n<N; n++)
             {
                 tmpR+=xR[n]*((long)cosR[k*n%N]) + xI[n]*((long)sinI[k*n%N]);
                 tmpI+=xI[n]*((long)cosR[k*n%N]) - xR[n]*((long)sinI[k*n%N]);
             }

            yR[k]=(short) (tmpR>>(B+15));
            yI[k]=(short) (tmpI>>(B+15)); /* 15 bit for fixed scaling */
      }
}

int bitReverse(int b, int B)
{
      int i, tmp;

      for (tmp=0, i=0; i<B; i++)
      {
            tmp=tmp<<1;
            tmp|=(b&1);
            b=b>>1;
      }
      return tmp;
}

void Split_Radix_FFT_asm(short *xR, short *xI, int N)
{
      int n, B, h, H;
      short *bitRevTable, tmp, *ptr;
      int *cosSinTable; /* cosSinTable[0]={bit31~16...bit15~0}= { sinI[n],
      cosR[n] }
      /* cosSinTable[1]={bit31~16...bit15~0}= { cosR[n], -sinI[n]}......*/

      B=(int)(log10(N)/log10(2));

      /* create the cos& sin table */
      cosSinTable=(int *) malloc(4*N*sizeof(short));
      ptr=(short *)cosSinTable;
      for(n=0; n<N; n++)
      {
            /* swap for big endian  */
            ptr[4*n+1]=(short) (cos(2*pi*n/N)*32767);
            ptr[4*n+0]=(short) (sin(2*pi*n/N)*32767);
```

Figure 7. FFT Example, C Code (Sheet 4 of 20)

```
            ptr[4*n+3]=-ptr[4*n+0];
            ptr[4*n+2]=ptr[4*n+1];

            /*
            printf("cosSinTable[%d]=%x \n", 2*n, cosSinTable[2*n]);
            printf("cosSinTable[%d]=%x \n", 2*n+1, cosSinTable[2*n+1]);
            printf(" cosR=%x, -sinI=%x, sinI=%x, cos=%x \n",
            ptr[4*n+2],ptr[4*n+3],ptr[4*n+0],ptr[4*n+1]);
            */
    }

    /* create bit reverse table */

    bitRevTable=(short *) malloc(N*sizeof(short));

    for(h=0, H=0, n=0; n<N; n++)
    {
           tmp=bitReverse(n,B);
           if(n<tmp)
           {
                bitRevTable[h]=2*n; /* 2 for word addressing */
                bitRevTable[h+1]=2*tmp;
                h+=2;
                H+=1;
           }

    }
     printf("bitRevTable size H=%d \n",H);

    /* code above this line in this function should be put into a initializa-
    tion section */


    /* FFT */
    Split_Radix_FFT_ASM(xR, xI, (short *)cosSinTable, N, bitRevTable,H);
}

void Split_Radix_FFT_asm_DSP(short *xR, short *xI, int N)
{
    int n, B, h, H;
    short *bitRevTable, tmp, *ptr;
    int *cosSinTable;  /* cosSinTable[0]={bit31~16...bit15~0}= { sinI[n],
    cosR[n] }   /* cosSinTable[1]={bit31~16...bit15~0}= { cosR[n], -si-
    nI[n]}......*/

    B=(int)(log10(N)/log10(2));

    /* create the cos& sin table */
    cosSinTable=(int *) malloc(4*N*sizeof(short));
    ptr=(short *)cosSinTable;
```

Figure 7. FFT Example, C Code (Sheet 5 of 20)

```
      for(n=0; n<N; n++)
      {
            /* swap for big endian  */

            ptr[4*n+1]=(short) (cos(2*pi*n/N)*32767);
            ptr[4*n+0]=(short) (sin(2*pi*n/N)*32767);
            ptr[4*n+3]=-ptr[4*n+0];
            ptr[4*n+2]=ptr[4*n+1];

            /*
            printf("cosSinTable[%d]=%x \n", 2*n, cosSinTable[2*n]);
            printf("cosSinTable[%d]=%x \n", 2*n+1, cosSinTable[2*n+1]);
            printf(" cosR=%x, -sinI=%x, sinI=%x, cos=%x \n",
            ptr[4*n+2],ptr[4*n+3],ptr[4*n+0],ptr[4*n+1]);
            */
      }

            /* create bit reverse table */

            bitRevTable=(short *) malloc(N*sizeof(short));
            for(h=0, H=0, n=0; n<N; n++)
            {
                  tmp=bitReverse(n,B);

                 if(n<tmp)
                  {
                        bitRevTable[h]=2*n; /* 2 for word addressing */
                        bitRevTable[h+1]=2*tmp;
                        h+=2;
                        H+=1;
                  }

            }
           printf("bitRevTable size H=%d \n", H);


            /* code above this line in this function should be put into a ini-
            tialization section */


      /* FFT */

      Split_Radix_FFT_ASM_DSP(xR, xI, (short *)cosSinTable, N, bi-
      tRevTable,H);
}
```

Figure 7. FFT Example, C Code (Sheet 6 of 20)

```
void Split_Radix_FFT_C(short *xR, short *xI, int N)
{
     int n, B, h, H;
     short *bitRevTable, tmp;

     B=(int)(log10(N)/log10(2));

     /* create the cos& sin table */

     cosR=(short *) malloc(N*sizeof(short));
     sinI=(short *) malloc(N*sizeof(short));

     for(n=0; n<N; n++)
     {
           cosR[n]=(short) (cos(2*pi*n/N)*32767);
           sinI[n]=(short) (sin(2*pi*n/N)*32767);
     }

     /* create bit reverse table */
     bitRevTable=(short *) malloc(N*sizeof(short));
     for(h=0, H=0, n=0; n<N; n++)
     {
            tmp=bitReverse(n,B);

            if(n<tmp)
            {
                 bitRevTable[h]=n;
                 bitRevTable[h+1]=tmp;
                 h+=2;
                 H+=1;
            }
      }


     /* FFT */

     /* Split_Radix_ButterFly(xR, xI, N, 1); */

     Split_Radix_ButterFly_Optimized(xR, xI, N, 1);
     /* Split_Radix_ButterFly_with_rouding(xR, xI, N, 1); */

     /* reorder the output, optimized way */
     for(n=0; n<H; n++)
     {
           tmp=xR[bitRevTable[2*n]];
           xR[bitRevTable[2*n]]=xR[bitRevTable[2*n+1]];
           xR[bitRevTable[2*n+1]]=tmp;

           tmp=xI[bitRevTable[2*n]];
           xI[bitRevTable[2*n]]=xI[bitRevTable[2*n+1]];
           xI[bitRevTable[2*n+1]]=tmp;
     }

}
```

Figure 7. FFT Example, C Code (Sheet 7 of 20)

```
void Split_Radix_ButterFly_Optimized(short *xR, short *xI, int M, int q)
{
      /* the result is divided by N */
      /* q=N/M */

      int tmpR, tmpI, tmpRR, tmpII;
      short n, L, N;

      if(M==4)
      {
            L=M>>1;
            for(n=0; n<L; n++)
            {
                  tmpR=((int)xR[n])+xR[n+L];
                  tmpI=((int)xI[n])+xI[n+L];

                  tmpRR=((int)xR[n])-xR[n+L];
                  tmpII=((int)xI[n])-xI[n+L];

                  xR[n]=(short) (tmpR>>1);
                  xI[n]=(short) (tmpI>>1);
                  xR[n+L]=(short) (tmpRR>>1);
                  xI[n+L]=(short) (tmpII>>1);
            }

            tmpR=((int)xR[2]+xI[3])>>1;
            tmpI=((int)xI[2]-xR[3])>>1;

            tmpRR=((int)xR[2]-xI[3])>>1;
            tmpII=((int)xI[2]+xR[3])>>1;

            xR[2]=(short)tmpR;
            xI[2]=(short)tmpI;

            xR[3]=(short)tmpRR;
            xI[3]=(short)tmpII;

            /* top 2 points */

            tmpR=((int)xR[0])+ xR[1];
            tmpI=((int)xI[0])+ xI[1];

            tmpRR=((int)xR[0])- xR[1];
            tmpII=((int)xI[0])- xI[1];

            xR[0]=(short) (tmpR>>1);
            xI[0]=(short) (tmpI>>1);
            xR[1]=(short) (tmpRR>>1);
            xI[1]=(short) (tmpII>>1);
      }

      else if(M==2)
      {
            tmpR=((int)xR[0])+ xR[1];
            tmpI=((int)xI[0])+ xI[1];
```

Figure 7. FFT Example, C Code (Sheet 8 of 20)

```
        xR[0]=(short) (tmpR>>1);
        xI[0]=(short) (tmpI>>1);
        xR[1]=(short) (tmpRR>>1);
        xI[1]=(short) (tmpII>>1);
    }

    else
    {
        L=M>>1;

        for(n=0; n<L; n++)
        {
            tmpR=((int)xR[n])+xR[n+L];
            tmpI=((int)xI[n])+xI[n+L];

            tmpRR=((int)xR[n])-xR[n+L];
            tmpII=((int)xI[n])-xI[n+L];

            xR[n]=(short) (tmpR>>1);
            xI[n]=(short) (tmpI>>1);
            xR[n+L]=(short) (tmpRR>>1);
            xI[n+L]=(short) (tmpII>>1);
        }

        N=M*q;
        M=M>>1;
        L=L>>1;

        n=0; /* is treated especially */
        tmpR=((int)xR[n+M]+xI[n+M+L])>>1;
        tmpI=((int)xI[n+M]-xR[n+M+L])>>1;

        tmpRR=((int)xR[n+M]-xI[n+M+L])>>1;
        tmpII=((int)xI[n+M]+xR[n+M+L])>>1;

        xR[n+M]=(short) tmpR;
        xI[n+M]=(short) tmpI;

        xR[n+M+L]=(short) tmpRR;
        xI[n+M+L]=(short) tmpII;

        for(n=1; n<L; n++)
        {
            tmpR=((int)xR[n+M]+xI[n+M+L])>>1;
            tmpI=((int)xI[n+M]-xR[n+M+L])>>1;
            tmpRR=((int)xR[n+M]-xI[n+M+L])>>1;
            tmpII=((int)xI[n+M]+xR[n+M+L])>>1;

            xR[n+M]=(short) ((tmpR*cosR[(n*q)%N]+tmpI*si-
            nI[(n*q)%N])>>15);

            xI[n+M]=(short) ((tmpI*cosR[(n*q)%N]-tmpR*si-
            nI[(n*q)%N])>>15);
```

Figure 7. FFT Example, C Code (Sheet 9 of 20)

```
                  xR[n+M+L]=(short) ((tmpRR*cosR[(3*n*q)%N]+tmpII*si-
                  nI[(3*n*q)%N])>>15);

                  xI[n+M+L]=(short) ((tmpII*cosR[(3*n*q)%N]-tmpRR*si-
                  nI[(3*n*q)%N])>>15);
            }


            q=q*2;
            Split_Radix_ButterFly_Optimized(xR, xI, M, q);
            Split_Radix_ButterFly_Optimized(xR+M, xI+M, L, q*2);
            Split_Radix_ButterFly_Optimized(xR+M+L, xI+M+L, L, q*2);

      }
}

void Split_Radix_ButterFly_with_rouding(short *xR, short *xI, int M, int q)
{
      /* the result is divided by N */
      /* more accurate because we are using rounding */

      /* q=N/M */
      int tmpR, tmpI, tmpRR, tmpII;
      short n, L, N;

      if(M==1)
            return;

      else if(M==2)
      {
            tmpR=((int)xR[0])+ xR[1];
            tmpI=((int)xI[0])+ xI[1];

            tmpRR=((int)xR[0])- xR[1];
            tmpII=((int)xI[0])- xI[1];


            xR[0]=(short) (tmpR>>1);
            xI[0]=(short) (tmpI>>1);
            xR[1]=(short) (tmpRR>>1);
            xI[1]=(short) (tmpII>>1);
      }

      else
      {
            L=M>>1;

            for(n=0; n<L; n++)
            {
                  tmpR=((int)xR[n])+xR[n+L];
                  tmpI=((int)xI[n])+xI[n+L];

                  tmpRR=((int)xR[n])-xR[n+L];
                  tmpII=((int)xI[n])-xI[n+L];
```

Figure 7. FFT Example, C Code (Sheet 10 of 20)

```
                xR[n]=(short) (tmpR>>1);
                xI[n]=(short) (tmpI>>1);
                xR[n+L]=(short) (tmpRR>>1);
                xI[n+L]=(short) (tmpII>>1);
          }

          N=M*q;
          M=M>>1;
          L=L>>1;

          for(n=0; n<L; n++)
          {
                tmpR=((int)xR[n+M]+xI[n+M+L])>>1;
                tmpI=((int)xI[n+M]-xR[n+M+L])>>1;

                tmpRR=((int)xR[n+M]-xI[n+M+L])>>1;
                tmpII=((int)xI[n+M]+xR[n+M+L])>>1;

                xR[n+M]=(short) (((tmpR*cosR[(n*q)%N]+tmpI*si-
                nI[(n*q)%N])>>14)+1)>>1;

                xI[n+M]=(short) (((tmpI*cosR[(n*q)%N]-tmpR*si-
                nI[(n*q)%N])>>14)+1)>>1;

                xR[n+M+L]=(short) (((tmpRR*cosR[(3*n*q)%N]+tmpII*si-
                nI[(3*n*q)%N])>>14)+1)>>1;

                xI[n+M+L]=(short) (((tmpII*cosR[(3*n*q)%N]-tmpRR*si-
                nI[(3*n*q)%N])>>14)+1)>>1;
          }

          q=q*2;
          Split_Radix_ButterFly_with_rouding(xR, xI, M, q);
          Split_Radix_ButterFly_with_rouding(xR+M, xI+M, L, q*2);
          Split_Radix_ButterFly_with_rouding(xR+M+L, xI+M+L, L, q*2);
     )
)

void Split_Radix_ButterFly(short *xR, short *xI, int M, int q)
{
     /* the result is divided by N */

     /* q=N/M */

     int tmpR, tmpI, tmpRR, tmpII;
     short n, L, N;

     if(M==1)
          return;
```

Figure 7. FFT Example, C Code (Sheet 11 of 20)

```
      else if(M==2)
      {
            tmpR=((int)xR[0])+ xR[1];
            tmpI=((int)xI[0])+ xI[1];
            tmpRR=((int)xR[0])- xR[1];
            tmpII=((int)xI[0])- xI[1];

            xR[0]=(short) (tmpR>>1);
            xI[0]=(short) (tmpI>>1);
            xR[1]=(short) (tmpRR>>1);
            xI[1]=(short) (tmpII>>1);
      }

      else
      {
            L=M>>1;
            for(n=0; n<L; n++)
            {
                  tmpR=((int)xR[n])+xR[n+L];
                  tmpI=((int)xI[n])+xI[n+L];
                  tmpRR=((int)xR[n])-xR[n+L];
                  tmpII=((int)xI[n])-xI[n+L];

                  xR[n]=(short) (tmpR>>1);
                  xI[n]=(short) (tmpI>>1);
                  xR[n+L]=(short) (tmpRR>>1);
                  xI[n+L]=(short) (tmpII>>1);
            }

            N=M*q;
            M=M>>1;
            L=L>>1;
            for(n=0; n<L; n++)
            {
                  tmpR=((int)xR[n+M]+xI[n+M+L])>>1;
                  tmpI=((int)xI[n+M]-xR[n+M+L])>>1;
                  tmpRR=((int)xR[n+M]-xI[n+M+L])>>1;
                  tmpII=((int)xI[n+M]+xR[n+M+L])>>1;

                  xR[n+M]=(short) ((tmpR*cosR[(n*q)%N]+tmpI*si-
                  nI[(n*q)%N])>>15);

                  xI[n+M]=(short) ((tmpI*cosR[(n*q)%N]-tmpR*si-
                  nI[(n*q)%N])>>15);

                  xR[n+M+L]=(short) ((tmpRR*cosR[(3*n*q)%N]+tmpII*si-
                  nI[(3*n*q)%N])>>15);

                  xI[n+M+L]=(short) ((tmpII*cosR[(3*n*q)%N]-tmpRR*si-
                  nI[(3*n*q)%N])>>15);
            }
            q=q*2;
            Split_Radix_ButterFly(xR, xI, M, q);
            Split_Radix_ButterFly(xR+M, xI+M, L, q*2);
            Split_Radix_ButterFly(xR+M+L, xI+M+L, L, q*2);
      }
)
```

Figure 7. FFT Example, C Code (Sheet 12 of 20)

```
int getNum(char *str)
{
    int c;
    int i = 0;
    char input[100];

    if(str && *str) printf("%s", str);
    do
    {
        c = getc(stdin);

        if (c == 0x08)
        {
            if(i) i--;
        }
        else
        {
            input[i++] = c;
        }

    } while(i<100 && c!='
    input[i] = '\0';

    return atoi(input);
}


void testFFT()
{
     short *xR, *xI;
     short *yR, *yI;
     short *uR, *uI;
     short *vR, *vI;
     short *wR, *wI;
     short *pR, *pI;
     short *qR, *qI;

     float *zR, *zI, *A;

     int k,n,i, N;
     float f0, fs;

     N=128;
     f0=343.75;    /* 625=8000/128*10,    343.75=8000/128*5.5  */
     fs=8000;

     N=getNum("Number of Point=");

     numberOfRun=getNum("Number of run to profile=");

     printf("%d-point FFT \n”,N);
```

Figure 7. FFT Example, C Code (Sheet 13 of 20)

```
      xR=(short *) malloc(N*sizeof(short));
      yR=(short *) malloc(N*sizeof(short));
      uR=(short *) malloc(N*sizeof(short));
      vR=(short *) malloc(N*sizeof(short));
      wR=(short *) malloc(N*sizeof(short));
      pR=(short *) malloc(N*sizeof(short));
      qR=(short *) malloc(N*sizeof(short));

      xI=(short *) malloc(N*sizeof(short));
      yI=(short *) malloc(N*sizeof(short));
      uI=(short *) malloc(N*sizeof(short));
      vI=(short *) malloc(N*sizeof(short));
      wI=(short *) malloc(N*sizeof(short));
      pI=(short *) malloc(N*sizeof(short));
      qI=(short *) malloc(N*sizeof(short));

      zR=(float *) malloc(N*sizeof(float));
      zI=(float *) malloc(N*sizeof(float));
       A=(float *) malloc(N*sizeof(float));

      pi=asin(1)*2;
      printf("pi=%f \n", pi);


      printf("a %f Hz real sin waveform sampled at %fkHz, scaled for 16-bit
      fixed point \n", f0, fs);

      for(n=0; n<N; n++)
      {
            xR[n]=(short) (cos(2*pi*f0*n/fs)*32767);
            xI[n]=0;
       }

#if 1
      printf("input samples \n");

      for(n=0; n<N; n++)
      /* printf("xR[%d]=%d \n",n,xR[n]); */

      printf("%d \n", xR[n]);

#endif


DFT_FloatingPoint_N(xR, xI, zR, zI, N);

#if 0
      printf("floating point DFT output \n");

      for(n=0; n<N; n++)
      printf("zR[%d]=%f, zI[%d]=%f \n", n, zR[n], n, zI[n]);

#endif
```

Figure 7. FFT Example, C Code (Sheet 14 of 20)

```
    printf("floating point DFT output amplitude \n");

    for(n=0; n<N; n++)
    {
           A[n]=sqrt(zR[n]*zR[n]+zI[n]*zI[n]);
           /* printf("Amplitude[%d]=%f \n", n, A[n]); */

           /* printf("%f \n", A[n]); */
    }

    printf("floating point DFT output converted into 16 bit fixed-point \n");

    for(n=0; n<N; n++)
    {
          uR[n]=(short) (zR[n]);
          uI[n]=(short) (zI[n]);
           /* printf("uR[%d]=%d, uI[%d]=%d \n", n, uR[n], n, uI[n]);  */

    }

    printf("C fixed-point DFT \n");

    DFT_FixedPoint_N(xR, xI, vR, vI, N);

    printf("C fixed-point FFT \n");

    for(n=0; n<N; n++)
    {
          wR[n]=xR[n];
          wI[n]=xI[n];
    }

    Split_Radix_FFT_C(wR, wI, N);

    printf("asm fixed-point FFT \n");

    for(n=0; n<N; n++)
    {
          pR[n]=xR[n];
          pI[n]=xI[n];
    }
     Split_Radix_FFT_asm(pR, pI, N);

    printf("asm+DSP-copro  fixed-point FFT \n");

    for(n=0; n<N; n++)
    {
          qR[n]=xR[n];
          qI[n]=xI[n];
    }


    Split_Radix_FFT_asm_DSP(qR, qI, N);

    printf("fixed point DFT output \n");
```

Figure 7. FFT Example, C Code (Sheet 15 of 20)

```
      for(n=0; n<N; n++)
      {
       printf("floating point DFT:         zR[%d]=%f, zI[%d]=%f \n", n, zR[n],
        n, zI[n]);

         printf("rounding floating point DFT: uR[%d]=%d, uI[%d]=%d \n",
      n, uR[n], n, uI[n]);

       printf("fixed point DFT:             vR[%d]=%d, vI[%d]=%d \n",
      n, vR[n], n, vI[n]);

       printf("C fixed point FFT:          wR[%d]=%d, wI[%d]=%d \n", n, wR[n],
        n, wI[n]);

       printf("ASM fixed point FFT:         pR[%d]=%d, pI[%d]=%d \n \n", n,
        pR[n], n, pI[n]);

       printf("ASM+DSP-copro fixed point FFT:  qR[%d]=%d, qI[%d]=%d \n \n", n,
        qR[n], n, qI[n]);

      }

      printf("difference between ASM and ASM+DSP_Copro Split_Radix_FFT \n");
      checkDifference(qR,qI,pR,pI,N);
      printf("difference between C and ASM for Split_Radix_FFT \n");
      checkDifference(wR,wI,pR,pI,N);
      printf("difference between C fixed DFT and ASM Split_Radix_FFT \n");
      checkDifference(vR,vI,pR,pI,N);
      printf("difference between Rounding floating point DFT and ASM
      Split_Radix_FFT \n");

      checkDifference(uR,uI,pR,pI,N);
      printf(" \n \n @@@@@@@@FFT profiling.....@@@@@@@@@@ \n");

      printf("C fixed-point FFT \n");

      for(n=0; n<N; n++)
      {
            wR[n]=xR[n];
            wI[n]=xI[n];
      }
      Profile_Split_Radix_FFT_C(wR, wI, N);
      printf("asm fixed-point FFT \n");

      for(n=0; n<N; n++)
      {
            pR[n]=xR[n];
            pI[n]=xI[n];
      }
      Profile_Split_Radix_FFT_asm(pR, pI, N);
      printf("asm_DSP-copro fixed-point FFT \n");
      for(n=0; n<N; n++)
      {
           pR[n]=xR[n];
           pI[n]=xI[n];
      }
       Profile_Split_Radix_FFT_asm_DSP(pR, pI, N);
)
```

Figure 7. FFT Example, C Code (Sheet 16 of 20)

```
void checkDifference(short *wR, short *wI, short *vR, short *vI, int N)
{
     int n, tmp,  maxError, indMaxError;

     printf("    checking the difference... \n");

     for(indMaxError=0, maxError=0, n=0; n<N; n++)
     {
            tmp=abs(wR[n]-vR[n]);
            if(maxError<tmp)
            {
                  maxError=tmp;
                  indMaxError=n;
            }

            tmp=abs(wI[n]-vI[n]);
            if(maxError<tmp)
            {
                  maxError=tmp;
                  indMaxError=n;
            }
     }
printf("        max difference maxError=%d, indMaxError=%d \n",
     maxError, indMaxError);
}

void bitReverseOnArry(short *xR, short *xI, int N)
{
     int n, B;
     short *bitRevTable, tmp;
     B=(int)(log10(N)/log10(2));

     /* create bit reverse table */
     bitRevTable=(short *) malloc(N*sizeof(short));

     for(n=0; n<N; n++)
     {
           bitRevTable[n]=bitReverse(n,B);
     }

     /* reorder the output, simple way */

     for(n=0; n<N; n++)
     {
            if (n>bitRevTable[n])
                  continue;
                  tmp=xR[n];
                  xR[n]=xR[bitRevTable[n]];
                  xR[bitRevTable[n]]=tmp;

                  tmp=xI[n];
                  xI[n]=xI[bitRevTable[n]];
                  xI[bitRevTable[n]]=tmp;
     }
}
```

Figure 7. FFT Example, C Code (Sheet 17 of 20)

```
void Profile_Split_Radix_FFT_asm(short *xR, short *xI, int N)
{
      int n, B, h, H;
      short *bitRevTable, tmp, *ptr;
      int *cosSinTable;  /* cosSinTable[0]={bit31~16...bit15~0}= { sinI[n],
      cosR[n]} /* cosSinTable[1]={bit31~16...bit15~0}= { cosR[n], -si-
      nI[n]}......*/

      B=(int)(log10(N)/log10(2));

      /* create the cos& sin table */
      cosSinTable=(int *) malloc(4*N*sizeof(short));
      ptr=(short *)cosSinTable;
      for(n=0; n<N; n++)
      {
            /* swap for big endian  */
            ptr[4*n+1]=(short) (cos(2*pi*n/N)*32767);
             ptr[4*n+0]=(short) (sin(2*pi*n/N)*32767);
            ptr[4*n+3]=-ptr[4*n+0];
            ptr[4*n+2]=ptr[4*n+1];

            /* printf("cosSinTable[%d]=%x \n", 2*n, cosSinTable[2*n]);
            printf("cosSinTable[%d]=%x \n", 2*n+1, cosSinTable[2*n+1]);
             printf("cosR=%x,-sinI=%x,sinI=%x,cos=%x\n",
            ptr[4*n+2],ptr[4*n+3],ptr[4*n+0],ptr[4*n+1]);
            */
       }
      /* create bit reverse table */
      bitRevTable=(short *) malloc(N*sizeof(short));
      for(h=0, H=0, n=0; n<N; n++)
      {
             tmp=bitReverse(n,B);
             if(n<tmp)
             {
                  bitRevTable[h]=2*n; /* 2 for word addressing */
                  bitRevTable[h+1]=2*tmp;
                  h+=2;
                  H+=1;
             }
      }
      printf("bitRevTable size H=%d \n", H);
      /* the following is to profile the code, should be removed */
      writePerfrmCtrl(0x07);  /* start all the counters*/
      startClock=readCycleCounter();
            for(n=0;n<numberOfRun; n++)   /* run 10 times for measurement*/
            {/* FFT */
                  Split_Radix_FFT_ASM(xR, xI, (short *)cosSinTable, N, bi-
                  tRevTable,H);
            }
      stopClock=readCycleCounter();
      printf("%d point FFT using Split_Radix_FFT_ASM \n",N);
      printf("total cycles =%d \n", stopClock-startClock);
      printf("number of Run =%d \n", numberOfRun);
      printf("average cycle per point =%f \n\n",
      (stopClock-startClock)*1.0/numberOfRun/N);
}
```

Figure 7. FFT Example, C Code (Sheet 18 of 20)

```
void Profile_Split_Radix_FFT_asm_DSP(short *xR, short *xI, int N)
{
     int n, B, h, H;
     short *bitRevTable, tmp, *ptr;
     int *cosSinTable;  /* cosSinTable[0]={bit31~16...bit15~0}= { sinI[n],
     cosR[n] }/* cosSinTable[1]={bit31~16...bit15~0}= { cosR[n], -si-
     nI[n]}......*/

    B=(int)(log10(N)/log10(2));
    /* create the cos& sin table */
    cosSinTable=(int *) malloc(4*N*sizeof(short));
    ptr=(short *)cosSinTable;

    for(n=0; n<N; n++)
    {
          /* swap for big endian  */
          ptr[4*n+1]=(short) (cos(2*pi*n/N)*32767);
           ptr[4*n+0]=(short) (sin(2*pi*n/N)*32767);
          ptr[4*n+3]=-ptr[4*n+0];
          ptr[4*n+2]=ptr[4*n+1];

           /* printf("cosSinTable[%d]=%x \n", 2*n, cosSinTable[2*n]);
           printf("cosSinTable[%d]=%x \n", 2*n+1, cosSinTable[2*n+1]);
           printf("cosR=%x,-sinI=%x,sinI=%x,cos=%x\n",
           ptr[4*n+2],ptr[4*n+3],ptr[4*n+0],ptr[4*n+1]);
           */
    }

     /* create bit reverse table */
     bitRevTable=(short *) malloc(N*sizeof(short));
     for(h=0, H=0, n=0; n<N; n++)
     {
           tmp=bitReverse(n,B);
           if(n<tmp)
           {
                bitRevTable[h]=2*n; /* 2 for word addressing */
                bitRevTable[h+1]=2*tmp;
                h+=2;
                H+=1;
          }
     }
     printf("bitRevTable size H=%d
     /* the following is to profile the code, should be removed */
     writePerfrmCtrl(0x07);  /* start all the counters*/
     startClock=readCycleCounter();
     for(n=0;n<numberOfRun; n++)   /* run 10 times for measurement*/
     {/* FFT */
           Split_Radix_FFT_ASM_DSP(xR, xI, (short *)cosSinTable, N, bi-
           tRevTable,H);
     }

     stopClock=readCycleCounter();
```

Figure 7. FFT Example, C Code (Sheet 19 of 20)

```
      printf("%d point FFT using Split_Radix_FFT_ASM \n", N);
      printf("total cycles =%d \n", stopClock-startClock);
      printf("number of Run =%d \n", numberOfRun);
       printf("average  cycle  per  point  =%f  \n  \n",  stopClock-start-
      Clock)*1.0/numberOfRun/N);

}

void Profile_Split_Radix_FFT_C(short *xR, short *xI, int N)
{
      int m, n, B, h, H;
      short *bitRevTable, tmp;
      B=(int)(log10(N)/log10(2));

      /* create the cos& sin table */

      cosR=(short *) malloc(N*sizeof(short));
      sinI=(short *) malloc(N*sizeof(short));

      for(n=0; n<N; n++)
      {
            cosR[n]=(short) (cos(2*pi*n/N)*32767);
            sinI[n]=(short) (sin(2*pi*n/N)*32767);
      }
      /* create bit reverse table */

      bitRevTable=(short *) malloc(N*sizeof(short));
      for(h=0, H=0, n=0; n<N; n++)
      {
             tmp=bitReverse(n,B);
             if(n<tmp)
             {
                  bitRevTable[h]=n;
                  bitRevTable[h+1]=tmp;
                  h+=2;
                  H+=1;
             }
      }
      /* the following is to profile the code, should be removed */

      writePerfrmCtrl(0x07);  /* start all the counters*/
      startClock=readCycleCounter();
      for(m=0;m<numberOfRun; m++)   /* run 10 times for measurement*/
      {
            /* FFT */
            Split_Radix_ButterFly_Optimized(xR, xI, N, 1);
            /* reorder the output, optimized way */
            for(n=0; n<H; n++)
            {
                  tmp=xR[bitRevTable[2*n]];
                  xR[bitRevTable[2*n]]=xR[bitRevTable[2*n+1]];
                  xR[bitRevTable[2*n+1]]=tmp;
                  tmp=xI[bitRevTable[2*n]];
                  xI[bitRevTable[2*n]]=xI[bitRevTable[2*n+1]];
                  xI[bitRevTable[2*n+1]]=tmp;
            }
      }
```

Figure 7. FFT Example, C Code (Sheet 20 of 20)

```
      stopClock=readCycleCounter();
      printf("%d point FFT using Split_Radix_FFT_C \n", N);
      printf("total cycles =%d \n", stopClock-startClock);
      printf("number of Run =%d \n", numberOfRun);
      printf("average cycle per point =%f \n \n", stopClock-start-
      Clock)*1.0/numberOfRun/N);
}
```

Figure 8. FFT Example, Assembly Code (Sheet 1 of 11)

```
@@/*************************************************************************
@@*
@@* @author Intel Corporation
@@* @date 17 June 2004
@@*
@@*
@@* -- Intel Copyright Notice --
@@*
@@*
@@*
@@*
@@* Copyright 2004 Intel Corporation All Rights Reserved.
@@*
@@*
@@* The source code contained or described herein and all documents
@@* related to the source code ("Material") are owned by Intel
@@* Corporation or its suppliers or licensors. Title to the Material
@@* remains with Intel Corporation or its suppliers and licensors.
@@* The Material contains trade secrets and proprietary and confidential
@@* information of Intel or its suppliers and licensors. The Material
@@* is protected by worldwide copyright and trade secret laws and treaty
@@* provisions. Except for the licensing of the source code hereunder,
@@* no part of the Material may be used, copied, reproduced, modified,
@@* published, uploaded, posted, transmitted, distributed, or disclosed
@@* in any way without Intel's prior express written permission.
@@*
@@*
@@* Except for the licensing of the source code as provided hereunder,
@@* no license under any patent, copyright, trade secret or other
@@* intellectual property right is granted to or conferred upon you by
@@* disclosure or delivery of the Materials, either expressly, by
@@* implication, inducement, estoppel or otherwise and any license under
@@* such intellectual property rights must be express and approved by
@@* Intel in writing.
@@*
@@*
@@* For further details, please see the file README.TXT distributed with
@@* this software.
@@* -- End Intel Copyright Notice --
@@*************************************************************************/

@ 2 point, 4 point, and first pass in the second loop are treated special
@ because no multiplication required
```

Figure 8. FFT Example, Assembly Code (Sheet 2 of 11)

```
#include <arch/arm/arm.h>

@ Split_Radix_FFT_ASM(short *xR, short *xI,short *cosRsinI, int N, short @*bi-
tRevTable, int H);
@ Split_Radix_FFT_ASM_DSP(short *xR, short *xI,short *cosRsinI, int N, short
@*bitRevTable, int H);

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@ split radix FFT
@@ function parameter:
@@
@@ r0 = xR: input/output real part pointer
@@ r1 = xI: input/output image part pointer
@@ r2 = cos -sin sin cos: cos sin table pointer
@@ r3 = N: length of FFT
@@ [sp,#40] = bitRevTable:bit reverse table
@@ [sp,#44] = Hbit reverse table size
@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


.global Split_Radix_FFT_ASM
.global _Split_Radix_FFT_ASM
.global Split_Radix_FFT_ASM__FPsN20iT0i

.global Split_Radix_FFT_ASM_DSP
.global _Split_Radix_FFT_ASM_DSP
.global Split_Radix_FFT_ASM_DSP__FPsN20iT0i

.balign 4

Split_Radix_FFT_ASM_DSP:
Split_Radix_FFT_ASM_DSP:
Split_Radix_FFT_ASM_DSP__FPsN20iT0i:

   stmdb  sp!,{r4-r12,lr}
   mov r4, #8 @ q=1 (8 bytes because of 4 shorts for the cos sin table)
   bl Split_Radix_ButterFly_asm_DSP

   @ bit reverse
   b reorderOutput
.balign 4
Split_Radix_FFT_ASM:
_Split_Radix_FFT_ASM:
Split_Radix_FFT_ASM__FPsN20iT0i:

   stmdb  sp!,{r4-r12,lr}
   mov r4, #8@ q=1 (8 bytes because of 4 shorts for the cos sin table)
   bl Split_Radix_ButterFly_asm

   @ bit reverse
reorderOutput:
   ldr r4, [sp,#40] @ bitRevTable
   ldr r5, [sp,#44] @ bitRevTable size
   cmp r5, #0
   beq doneNow
```

Figure 8. FFT Example, Assembly Code (Sheet 3 of 11)

```
loopR:
   ldrsh r6, [r4], #2     @ bitRevTable
   ldrsh r8, [r0,+r6]     @ xR[n]
   ldrsh r9, [r1,+r6]     @ xI[n]

   ldrsh r7, [r4], #2  @ bitRevTable
   ldrsh r10, [r0,+r7]    @ xR[bitReverse[n]]
   ldrsh r11, [r1,+r7]    @ xI[bitReverse[n]]

@ exchange

   strh    r8, [r0,+r7]
   strh    r9, [r1,+r7]
   strh    r10, [r0,+r6]
   strh    r11, [r1,+r6]

   subs r5, r5, #1
   bne loopR

doneNow:
   ldmia sp!,{r4-r12,pc}        @ return

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@ void Split_Radix_ButterFly_asm(short *xR, short *xI,short *cosR, short
@@ *sinI, int M, int q)
@@ r0 = xR: input/output real part pointer
@@ r1 = xI: input/output image part pointer
@@ r2 = cos -sin sin cosR: cos sin table pointer
@@ r3 = M:FFT
@@ r4 = q:FFT
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

.align 4
Split_Radix_ButterFly_asm:
_Split_Radix_ButterFly_asm:

       stmdb       sp!,{r0-r12,lr}         @ push registers

       cmp r3, #4
       bne checkNext

fourPointFFT:
       mov r7,   r3@ L
loop4:
       ldrsh r10, [r0]               @ xR[n]
       ldrsh r11, [r0,r3]            @ xR[n+L]
       sub     r14, r10, r11           @ tmpRR=((int)xR[n])- xR[n+L];
       add     r12, r10, r11           @ tmpR=((int)xR[n])+ xR[n+L]
       mov r14, r14, asr #1
       mov r12, r12, asr #1
       strh     r14, [r0,r3]            @ xR[n+L]=(short) (tmpRR>>1);
       strh     r12, [r0], #+2          @ xR[n]=(short) (tmpR>>1);  n=n+1

       ldrsh r10, [r1]               @ xI[n]
```

Figure 8. FFT Example, Assembly Code (Sheet 4 of 11)

```
ldrsh r11, [r1,r3]            @ xI[n+L]
       sub     r14, r10, r11          @ tmpII=((int)xI[n])- xI[n+L];
       add     r12, r10, r11          @ tmpI=((int)xI[n])+ xI[n+L]
       mov r14, r14, asr #1
       mov r12, r12, asr #1
       strh     r14, [r1,r3]           @ xI[n+L]=(short) (tmpII>>1);
       strh     r12, [r1], #+2         @ xI[n]=(short) (tmpI>>1); n=n+1

       subs r7, r7, #2
       bne loop4

       @ at this point, r0 exatly point to xR[L], r1 exactly point to xI[L]

       mov r3,   r3, asr #1 @ L=M/2

       ldrsh r10, [r0]               @ xR[n]
       ldrsh r11, [r0,r3]            @ xR[n+L]

       ldrsh r8,  [r1]               @ xI[n]
       ldrsh r9,  [r1,r3]            @ xI[n+L]

       add     r12, r10, r9            @ tmpR=((int)xR[n])+ xI[n+L]
       sub     r14, r8, r11            @ tmpI=((int)xI[n])- xR[n+L];

       sub     r10,  r10, r9           @ tmpRR=((int)xR[n])- xI[n+L]
       add     r11,  r8, r11           @ tmpII=((int)xI[n])+ xR[n+L];

       mov r12, r12, asr #1        @ tmpR
       mov r14, r14, asr #1        @ tmpI
       mov r10, r10, asr #1        @ tmpRR
       mov r11, r11, asr #1        @ tmpII

       strh r11, [r1,r3]            @ xI[n+L]
       strh r10, [r0, r3]           @ xR[n+L]
       strh r14, [r1], #-4          @ xI[n], point back to the begining
       strh r12, [r0], #-4          @ xR[n], point back to the begining
       b twoPointsDFT

checkNext:
       cmp r3, #2
       bne oneButterFly

twoPointsDFT:
       ldrsh r10, [r0], #+2          @ xR[0]
       ldrsh r11, [r0], #-2          @ xR[1]
       add     r12, r10, r11           @ tmpR=((int)xR[0])+ xR[1]
       sub     r14, r10, r11           @ tmpRR=((int)xR[0])- xR[1];
       mov r12, r12, asr #1
       mov r14, r14, asr #1
       strh     r12, [r0], #+2          @ xR[0]=(short) (tmpR>>1);
       strh     r14, [r0], #-2          @ xR[1]=(short) (tmpRR>>1);

       ldrsh r10, [r1], #+2          @ xI[0]
       ldrsh r11, [r1], #-2          @ xI[1]
       add     r12, r10, r11           @ tmpI=((int)xI[0])+ xI[1]
       sub     r14, r10, r11           @ tmpII=((int)xI[0])- xI[1];
       mov r12, r12, asr #1
```

Figure 8. FFT Example, Assembly Code (Sheet 5 of 11)

```
       mov r14, r14, asr #1
       strh     r12, [r1], #+2          @ xI[0]=(short) (tmpI>>1);
       strh     r14, [r1], #-2          @ xI[1]=(short) (tmpII>>1);

       ldmia    sp!,{r0-r12,pc}         @ return

oneButterFly:
       @ save variables for iterations
       stmdb   sp!,{r0-r4}             @ push registers

       @ section A of the butterfly
       mov r7,   r3@ L
loop1:
       ldrsh r10, [r0]               @ xR[n]
       ldrsh r11, [r0,r3]            @ xR[n+L]
       sub     r14, r10, r11           @ tmpRR=((int)xR[n])- xR[n+L];
       add     r12, r10, r11           @ tmpR=((int)xR[n])+ xR[n+L]
       mov r14, r14, asr #1
       mov r12, r12, asr #1
       strh     r14, [r0,r3]            @ xR[n+L]=(short) (tmpRR>>1);
       strh     r12, [r0], #+2          @ xR[n]=(short) (tmpR>>1);  n=n+1

       ldrsh r10, [r1]               @ xI[n]
       ldrsh r11, [r1,r3]            @ xI[n+L]
       sub     r14, r10, r11           @ tmpII=((int)xI[n])- xI[n+L];
       add     r12, r10, r11           @ tmpI=((int)xI[n])+ xI[n+L]
       mov r14, r14, asr #1
       mov r12, r12, asr #1
       strh     r14, [r1,r3]            @ xI[n+L]=(short) (tmpII>>1);
       strh     r12, [r1], #+2          @ xI[n]=(short) (tmpI>>1); n=n+1

       subs r7, r7, #2
       bne loop1

       @ section B of the butterfly
        @ at this point, r0 exactly point to xR[L], r1 exactly point to xI[L]

       mov r3,   r3, asr #1@ L=M/2
       sub r7,   r3, #2 @ treat the first loop specially

       addr6,   r4, r4, lsl #1@ r7=3*q
       add r5,   r2, r6
       @ n+=3Q to skip first loop, point to cos&sin with 3q offset
       addr2,   r2, r4        @ n+=q to skip first loop,

       @ first loop treated specially because there is no need to do
        @ multiplication for n=0

       ldrsh r10, [r0]               @ xR[n]
       ldrsh r11, [r0,r3]            @ xR[n+L]

       ldrsh r8,  [r1]               @ xI[n]
       ldrsh r9,  [r1,r3]            @ xI[n+L]
       add     r12, r10, r9            @ tmpR=((int)xR[n])+ xI[n+L]
       sub     r14, r8, r11            @ tmpI=((int)xI[n])- xR[n+L];

       sub     r10,  r10, r9           @ tmpRR=((int)xR[n])- xI[n+L]
       add     r11,  r8, r11           @ tmpII=((int)xI[n])+ xR[n+L];
```

Figure 8. FFT Example, Assembly Code (Sheet 6 of 11)

```
        mov r10, r10, asr #1        @ tmpRR
        mov r11, r11, asr #1        @ tmpII
        mov r12, r12, asr #1        @ tmpR
        mov r14, r14, asr #1        @ tmpI

        @ r8 & r9 are free now

        strh r11, [r1,r3]            @ xI[n+L]
        strh r10, [r0, r3]           @ xR[n+L]
        strh r14, [r1], #+2          @ xI[n]
        strh r12, [r0], #+2          @ xR[n], n+=2

        @ multiplication required for n>1
loop2:
        ldrsh r10, [r0]               @ xR[n]
        ldrsh r11, [r0,r3]            @ xR[n+L]

        ldrsh r8,  [r1]               @ xI[n]
        ldrsh r9,  [r1,r3]            @ xI[n+L]

        add     r12, r10, r9            @ tmpR=((int)xR[n])+ xI[n+L]
        sub     r14, r8, r11            @ tmpI=((int)xI[n])- xR[n+L];

        sub     r10,  r10, r9           @ tmpRR=((int)xR[n])- xI[n+L]
        add     r11,  r8, r11           @ tmpII=((int)xI[n])+ xR[n+L];

        mov r12, r12, asr #1        @ tmpR
        mov r14, r14, asr #1        @ tmpI
        mov r10, r10, asr #1        @ tmpRR
        mov r11, r11, asr #1        @ tmpII

        @ r8 & r9 are free now

        ldr r8, [r5,#4]         @ {cosR[n], -sinI[n]}
        smulbt r9, r11,  r8@ tmpII*cosR
        smlabb r9, r10,  r8, r9 @ -tmpRR*sinI
        mov r9, r9,   asr #15       @
        strh r9, [r1,r3]             @ xI[n+L]

        ldrr8, [r5], r6         @ {sinI[n], cosR[n]}, n+=3*q
        smulbt r9, r11,  r8@ tmpII*sinI
        smlabb r9, r10,  r8, r9 @ tmpRR*cosR
        mov r9, r9,   asr #15       @
        strh r9, [r0, r3]            @ xR[n+L]

        ldrr8, [r2,#4]              @ {cosR[n], -sinI[n]}
        smulbt r9, r14,  r8@ tmpI*cosR
        smlabb r9, r12,  r8, r9 @ -tmpR*sinI
        mov r9, r9,   asr #15       @
        strh r9, [r1], #+2           @ xI[n]

        ldrr8, [r2], r4         @ {sinI[n], cosR[n]}, n+=q
        smulbt r9, r14,  r8@ tmpI*sinI
        smlabb r9, r12,  r8, r9 @ tmpR*cosR
        mov r9, r9,   asr #15       @
        strh r9, [r0], #+2           @ xR[n], n+=2
```

Figure 8. FFT Example, Assembly Code (Sheet 7 of 11)

```
        subs r7, r7, #2
        bne loop2

        @ restore variables for iterations

        ldmia    sp!,{r0-r4}             @ pop

        cmp r3, #4
        beq twoPointsDFT @ do 2 points DFT


iterating:
        mov r3, r3, lsr #1 @ M=M/2
        mov r4, r4, lsl #1 @ q=q*2
        bl Split_Radix_ButterFly_asm

        add r0, r0, r3, lsl #1 @ xR+M
        add r1, r1, r3, lsl #1 @ xI+M
        mov r3, r3, lsr #1 @ M=M/2
        mov r4, r4, lsl #1 @ q=q*2
        bl Split_Radix_ButterFly_asm

        add r0, r0, r3, lsl #1 @ xR+M+M/2
        add r1, r1, r3, lsl #1 @ xI+M+M/2
        bl Split_Radix_ButterFly_asm

endNow:
        ldmia    sp!,{r0-r12,pc}         @ return



@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@ void Split_Radix_ButterFly_asm_DSP(short *xR, short *xI,short *cosR, short
@@ *sinI, int M, int q)
@@ r0 = xR: input/output real part pointer
@@ r1 = xI: input/output image part pointer
@@ r2 = cos -sin sin cosR: cos sin table pointer
@@ r3 = M:FFT
@@ r4 = q:FFT
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

.align 4

Split_Radix_ButterFly_asm_DSP:
_Split_Radix_ButterFly_asm_DSP:

   stmdb       sp!,{r0-r12,lr}         @ push registers

   cmp r3, #4
   bne checkNext_DSP

fourPointFFT_DSP:
        mov r7,   r3@ L
loop4_DSP:
        ldrsh r10, [r0]               @ xR[n]
        ldrsh r11, [r0,r3]            @ xR[n+L]
```

Figure 8. FFT Example, Assembly Code (Sheet 8 of 11)

```
        sub     r14, r10, r11           @ tmpRR=((int)xR[n])- xR[n+L];
        add     r12, r10, r11           @ tmpR=((int)xR[n])+ xR[n+L]
        mov r14, r14, asr #1
        mov r12, r12, asr #1
        strh    r14, [r0,r3]             @ xR[n+L]=(short) (tmpRR>>1);
        strh    r12, [r0], #+2           @ xR[n]=(short) (tmpR>>1);  n=n+1

        ldrsh r10, [r1]               @ xI[n]
        ldrsh r11, [r1,r3]            @ xI[n+L]
        sub     r14, r10, r11           @ tmpII=((int)xI[n])- xI[n+L];
        add     r12, r10, r11           @ tmpI=((int)xI[n])+ xI[n+L]
        mov r14, r14, asr #1
        mov r12, r12, asr #1
        strh    r14, [r1,r3]             @ xI[n+L]=(short) (tmpII>>1);
        strh    r12, [r1], #+2           @ xI[n]=(short) (tmpI>>1); n=n+1

        subs r7, r7, #2
        bne loop4_DSP

        @ at this point, r0 exactly point to xR[L], r1 exactly point to xI[L]

        mov r3,   r3, asr #1@ L=M/2

        ldrsh r10, [r0]               @ xR[n]
        ldrsh r11, [r0,r3]            @ xR[n+L]

        ldrsh r8,  [r1]               @ xI[n]
        ldrsh r9,  [r1,r3]            @ xI[n+L]

        add     r12, r10, r9            @ tmpR=((int)xR[n])+ xI[n+L]
        sub     r14, r8, r11            @ tmpI=((int)xI[n])- xR[n+L];

        sub     r10,  r10, r9           @ tmpRR=((int)xR[n])- xI[n+L]
        add     r11,  r8, r11           @ tmpII=((int)xI[n])+ xR[n+L];

        mov r12, r12, asr #1        @ tmpR
        mov r14, r14, asr #1        @ tmpI
        mov r10, r10, asr #1        @ tmpRR
        mov r11, r11, asr #1        @ tmpII

        strh r11, [r1,r3]            @ xI[n+L]
        strh r10, [r0, r3]           @ xR[n+L]
        strh r14, [r1], #-4          @ xI[n], point back to the begining
        strh r12, [r0], #-4          @ xR[n], point back to the begining

        b twoPointsDFT_DSP

checkNext_DSP:

        cmp r3, #2
        bne oneButterFly_DSP

twoPointsDFT_DSP:
        ldrsh r10, [r0], #+2          @ xR[0]
        ldrsh r11, [r0], #-2          @ xR[1]
        add     r12, r10, r11           @ tmpR=((int)xR[0])+ xR[1]
        sub     r14, r10, r11           @ tmpRR=((int)xR[0])- xR[1];
        mov r12, r12, asr #1
```

Figure 8. FFT Example, Assembly Code (Sheet 9 of 11)

```
        mov r14, r14, asr #1         @
        strh     r12, [r0], #+2           @ xR[0]=(short) (tmpR>>1);
        strh     r14, [r0], #-2           @ xR[1]=(short) (tmpRR>>1);

        ldrsh r10, [r1], #+2           @ xI[0]
        ldrsh r11, [r1], #-2           @ xI[1]
        add     r12, r10, r11            @ tmpI=((int)xI[0])+ xI[1]
        sub     r14, r10, r11            @ tmpII=((int)xI[0])- xI[1];
        mov r12, r12, asr #1
        mov r14, r14, asr #1
        strh     r12, [r1], #+2           @ xI[0]=(short) (tmpI>>1);
        strh     r14, [r1], #-2           @ xI[1]=(short) (tmpII>>1);

        ldmia    sp!,{r0-r12,pc}          @ return

oneButterFly_DSP:

        @ save variables for iterations
        stmdb   sp!,{r0-r4}              @ push registers

        @ section A of the butterfly
        mov r7,   r3@ L

loop1_DSP:
        ldrsh r10, [r0]                @ xR[n]
        ldrsh r11, [r0,r3]             @ xR[n+L]
        sub     r14, r10, r11            @ tmpRR=((int)xR[n])- xR[n+L];
        add     r12, r10, r11            @ tmpR=((int)xR[n])+ xR[n+L]
        mov r14, r14, asr #1
        mov r12, r12, asr #1
        strh     r14, [r0,r3]             @ xR[n+L]=(short) (tmpRR>>1);
        strh     r12, [r0], #+2           @ xR[n]=(short) (tmpR>>1);  n=n+1

        ldrsh r10, [r1]                @ xI[n]
        ldrsh r11, [r1,r3]             @ xI[n+L]
        sub     r14, r10, r11            @ tmpII=((int)xI[n])- xI[n+L];
        add     r12, r10, r11            @ tmpI=((int)xI[n])+ xI[n+L]
        mov r14, r14, asr #1
        mov r12, r12, asr #1
        strh     r14, [r1,r3]             @ xI[n+L]=(short) (tmpII>>1);
        strh     r12, [r1], #+2           @ xI[n]=(short) (tmpI>>1); n=n+1

        subs r7, r7, #2
        bne loop1_DSP

        @ setion B of the butterfly
        @ at this point, r0 exactly point to xR[L], r1 exactly point to xI[L]
        mov r3,   r3, asr #1 @ L=M/2
        sub r7,   r3, #2 @ treat the first loop specially
        add r6,   r4, r4, lsl #1 @ r7=3*q
        add r5,   r2, r6
         @ n+=3Q to skip first loop, point to cos&sin with 3q offset
        addr2,   r2, r4        @ n+=q to skip first loop,

        @ first loop treated specially because there is no need to do
         @ multiplication for n=0
        ldrsh r10, [r0]                @ xR[n]
        ldrsh r11, [r0,r3]             @ xR[n+L]
```

Figure 8. FFT Example, Assembly Code (Sheet 10 of 11)

```
ldrshr8,  [r1]                @ xI[n]
       ldrsh r9,  [r1,r3]             @ xI[n+L]

       add     r12, r10, r9             @ tmpR=((int)xR[n])+ xI[n+L]
       sub     r14, r8, r11             @ tmpI=((int)xI[n])- xR[n+L];

       sub     r10,  r10, r9            @ tmpRR=((int)xR[n])- xI[n+L]
       add     r11,  r8, r11            @ tmpII=((int)xI[n])+ xR[n+L];

       mov r10, r10, asr #1         @ tmpRR
       mov r11, r11, asr #1         @ tmpII
       mov r12, r12, asr #1         @ tmpR
       mov r14, r14, asr #1         @ tmpI

       @ r8 & r9 are free now
       strh r11, [r1,r3]             @ xI[n+L]
       strh r10, [r0, r3]            @ xR[n+L]
       strh r14, [r1], #+2           @ xI[n]
       strh r12, [r0], #+2           @ xR[n], n+=2

       @ multiplication required for n>1

loop2_DSP:
       ldrsh r10, [r0]               @ xR[n]
       ldrsh r11, [r0,r3]            @ xR[n+L]

       ldrsh r8,  [r1]               @ xI[n]
       ldrsh r9,  [r1,r3]            @ xI[n+L]

       add     r12, r10, r9             @ tmpR=((int)xR[n])+ xI[n+L]
       sub     r14, r8, r11             @ tmpI=((int)xI[n])- xR[n+L];

       sub     r10,  r10, r9            @ tmpRR=((int)xR[n])- xI[n+L]
       add     r11,  r8, r11            @ tmpII=((int)xI[n])+ xR[n+L];

       @ r8 & r9 are free now
       @ scale and pack the data

       mov r12, r12, lsl #15        @ tmpR
       mov r12, r12, lsr #16        @ tmpR >>1
       mov r14, r14, asr #1         @ tmpI >>1
       orr r12, r12, r14, lsl #16   @ r12={tmpI tmpR}

       mov r10, r10, lsl #15        @ tmpRR
       mov r10, r10, lsr #16        @ tmpRR >>1
       mov r11, r11, asr #1         @ tmpII >>1
       orr r10, r10, r11, lsl #16   @ r10={tmpII tmpRR}


       sub r8,  r8, r8
       ldr r9, [r5,#4]         @ {cosR[n], -sinI[n]}
       mar acc0, r8, r8 @ r10={tmpII tmpRR}
       miaph    acc0, r10, r9 @ tmpII*cosR -tmpRR*sinI
       mra r11,  r14, acc0  @ acc0=[r14 r11]
       mov r11,  r11, asr #15
       strh r11, [r1,r3]             @ xI[n+L]
       ldr r9, [r5], r6         @ {sinI[n], cosR[n]}, n+=3*q
```

Figure 8. FFT Example, Assembly Code (Sheet 11 of 11)

```
       mar acc0, r8, r8 @ r10={tmpII tmpRR}
       miaph    acc0, r10, r9 @ tmpRR*cosR + tmpII*sinI
       mra r11,  r14, acc0  @ acc0=[r14 r11]
       mov r11,  r11, asr #15
       strh r11, [r0, r3]           @ xR[n+L]

       ldr r9, [r2,#4]             @ {cosR[n], -sinI[n]}
       mar acc0, r8, r8 @ r12={tmpI tmpR}
       miaph    acc0, r12, r9 @ tmpI*cosR -tmpR*sinI
       mra r11,  r14, acc0  @ acc0=[r14 r11]
       mov r11,  r11, asr #15
       strh r11, [r1], #+2          @ xI[n]

       ldr r9, [r2], r4        @ {sinI[n], cosR[n]}, n+=q
       mar acc0, r8, r8 @ r12={tmpI tmpR}
       miaph    acc0, r12, r9 @ tmpR*cosR + tmpI*sinI
       mra r11,  r14, acc0  @ acc0=[r14 r11]
       mov r11,  r11, asr #15
       strh r11, [r0], #+2          @ xR[n], n+=2

       subs r7, r7, #2
       bne loop2_DSP

       @ restore variables for iterations

       ldmia    sp!,{r0-r4}             @ pop

iterating_DSP:
       mov r3, r3, lsr #1 @ M=M/2
       mov r4, r4, lsl #1 @ q=q*2
       bl Split_Radix_ButterFly_asm_DSP

       add r0, r0, r3, lsl #1 @ xR+M
       add r1, r1, r3, lsl #1 @ xI+M
       mov r3, r3, lsr #1 @ M=M/2
       mov r4, r4, lsl #1 @ q=q*2
       bl Split_Radix_ButterFly_asm_DSP

       add r0, r0, r3, lsl #1 @ xR+M+M/2
       add r1, r1, r3, lsl #1 @ xI+M+M/2
       bl Split_Radix_ButterFly_asm_DSP

endNow_DSP:
       ldmia    sp!,{r0-r12,pc}         @ return
```