

FIR Filtering Using the Lightweight Signal Processor on MPC5777M

by: Andrew Turner and Ian Howie

1 Introduction

The Lightweight Signal Processing (LSP) auxiliary processing unit (APU) is designed to accelerate processing applications normally suited to digital signal processing (DSP) operations. The LSP-APU is embedded in the e200z425n3 I/O processor found on the Freescale's MPC5777M device. The LSP-APU can be used to accelerate signal processing routines such as finite impulse response (FIR) and infinite impulse response (IIR) filters. This is accomplished using the standard 32-bit general purpose registers (GPRs), and supports vector multiply/multiply accumulate/ dot product operations using either a single or a pair of GPR registers as a source/destination accumulator or as a destination.

This application note details how to create an FIR filter using the LSP-APU and the benefits it provides. Examples are provided of the same FIR filter coded using standard C, LSP-APU assembly functions, and also C-level LSP intrinsic functions. This note also provides performance comparisons of each technique. A software package is provided with the application note that uses the MULTI integrated development environment tool provided by Green Hills software.

It is expected that the LSP-APU will be mainly utilized

Contents

1 Introduction.....	1
2 LSP comparison with SPE and SPE2.....	2
3 LSP –APU Programming basics	2
3.1 GPR Registers.....	2
3.2 GPR Register pairings	3
3.3 Instruction Overview	3
3.4 Example instructions	3
3.5 LSP-APU Data formats	4
3.6 Fractional arithmetic and its benefits to signal processing.....	5
4 FIR Filter Introduction.....	6
4.1 Creating a 10 tap filter with LSP instructions	7
4.2 LSP Intrinsic instructions.....	11
4.3 LSP Intrinsic data Types.....	12
4.4 FIR Filter using Ininsics	14
5 Performance comparisons	14
6 Summary.....	15
Appendix A.....	16
A.1 10 tap FIR filter using LSP-APU assembly	16
Appendix B.....	19
B.1 FIR filter written with LSP intrinsics.....	19

to perform signal processing on the 16-bit data output from the ADC modules, so the examples detailed in the application note and software operate on 16-bit data.

2 LSP comparison with SPE and SPE2

Freescall offers different embedded signal processing solutions for different devices depending on the target application. While the MPC5777M incorporates the LSP, other MPC5XXX Automotive microcontrollers may use the Signal Processing Engine (SPE) or SPE2. The LSP uses a 32-bit register model whereas the SPE/SPE2 uses a 64-bit register model. This gives the LSP less computational performance than the SPE/SPE2, but offers benefits in current consumption. Additionally the LSP does not contain a Floating Point unit which is present on SPE/SPE2. Existing SPE code can be ported to the LSP but some degradation in performance should be expected.

3 LSP-APU programming basics

The LSP APU utilizes a general purpose register file with thirty-two 32-bit registers. LSP instructions generally take elements from a first source register and operate on them with the corresponding elements of a second source register (and/or a third source register or register pair acting as an accumulator) to produce results. Results are placed in the destination register (32-bit results) or register pair (64-bit results). Instructions that are vector in nature (that is, produce results of more than one element) provide results for each element that are independent of the computation of the other elements.

3.1 GPR registers

The LSP APU requires a GPR register file with thirty-two 32-bit registers. Certain LSP APU instructions view the 32-bit register as being composed of a vector of elements, each of which is 16 bits wide. The most significant 16 bits are called halfword 0 (H0), the upper halfword, high halfword or even halfword. The least significant 16 bits are called halfword 1 (H1), the lower halfword, low halfword or odd halfword. LSP instructions write to all 32 bits of the destination register. [Figure 1](#) illustrates this nomenclature.

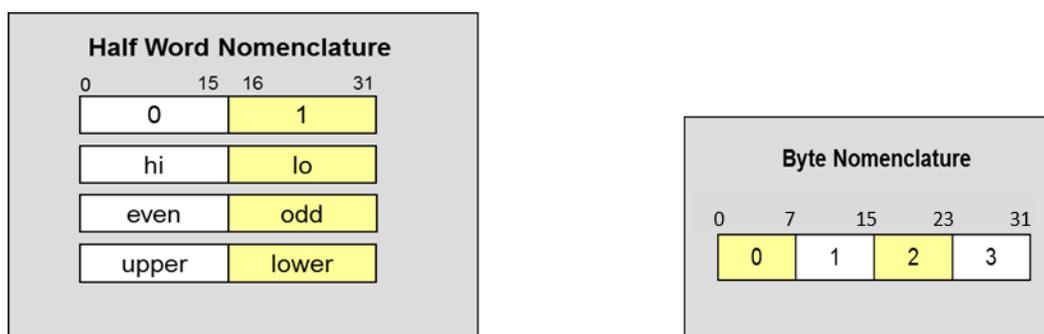


Figure 1. LSP register file nomenclature

3.2 GPR register pairings

Certain LSP instructions require a 64-bit source or destination. For these operands, a pair of general purpose registers (register pair) are used. Pairs are always defined as an adjacent even/odd pair, such as r10/r11, r12/r13, etc. Instruction encodings indicate the even register of the pair in the rD field.

3.3 Instruction overview

The following instruction types feature in the LSP-APU

- Loads and stores
 - load and store 64-bit values (via register pairs)
 - load and store 32-bit values
 - load and store 16-bit values
 - 16-bit values can be in upper and lower half of 32-bit word
 - Sign extension or not
 - Effective addresses must be naturally aligned for the length of the values loaded or stored
- Operations
 - Arithmetic, logical, comparison, selection, movement
 - Vector and scalar
- Multiply and Multiply-Accumulate
 - 16 x 16 -> 16, with or without accumulation
 - 16 x 16 -> 32, with or without accumulation
 - 16 x 16 or 32 x 32 -> 64, with accumulation
 - When accumulating, result is also written to destination register

3.4 Example instructions

LSP supports several different computational capabilities. Both modulo and saturation results can be performed. Modulo results produce truncation of the overflow bits in a calculation. Saturation provides a maximum or minimum representable value (for the data type) for positive or negative overflow respectively. Instructions are provided for a wide range of computational capability. The operation types can be divided into several basic categories:

- Simple instructions. These instructions use the corresponding elements of the operands to produce a result that is placed in the destination register.
- Simple vector instructions. These instructions use the corresponding elements of the operands to produce a vector result that is placed in the destination register.
 - Arithmetic, shift of vector elements
 - Rounding and saturation, operations
 - Vector packing and unpacking operations
 - Vector merge and extraction operations

- Complex instructions. The divide fractional instruction uses the operands to perform a fractional divide result that is placed in the destination register.
- Multiply and accumulate instructions. These instructions perform multiply operations, optionally add the result to the accumulator value in the destination register, and place the result into the destination register. These instructions are composed of different multiply forms, data formats, and data accumulate options.
- Dot product instructions. These instructions perform multiple multiply operations, optionally add the result to the accumulator value in the destination register, and place the result into the destination register. These instructions are composed of different forms, data formats and data accumulate options.
- Load and store instructions. These instructions provide load and store capabilities for moving data to and from memory. A variety of forms are provided that position data for efficient computation.
- Compare instructions.
- Miscellaneous instructions. These instructions perform miscellaneous functions such as field manipulation, bit-reversed and circular incrementing, count leading sign/zero bits, etc.

Full details of all instructions can be found in the LSP-APU reference manual.

An example of a 16-bit Vector addition LSP-APU instruction is shown in [Figure 2](#), illustrating the GPR register usage differences between a traditional classic Power PC add instruction and the LSP-APU add half-word instruction. Mnemonics for LSP instructions generally begin with the letter(s) “z” or “zv”.

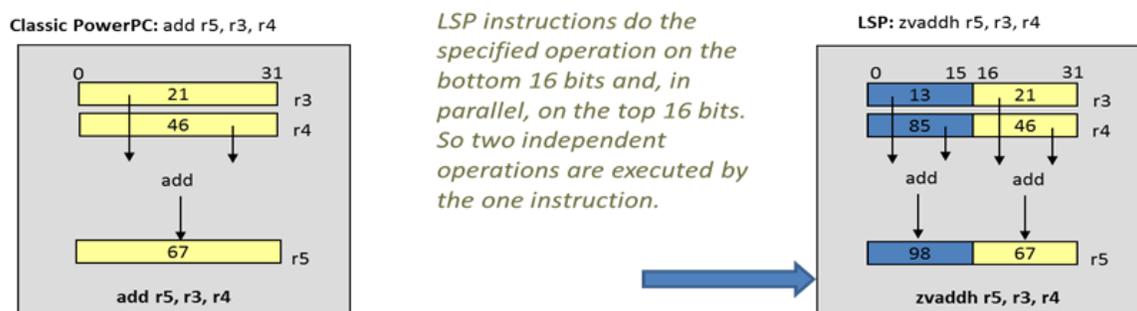


Figure 2. 16-bit vector addition example using LSP-APU

3.5 LSP-APU data formats

The LSP-APU provides two data formats. These are integer format – as traditionally used in computing, and fractional data format which is conventionally used for DSP fractional arithmetic and is useful for representing data converted from analog devices. In the LSP-APU, Integer formats can be signed or unsigned, whereas the Fractional data format is treated as signed.

Unsigned integer arithmetic operations produce identical bit results as unsigned fractional values would, therefore unsigned fractional instruction forms are not defined for LSP.

3.6 Fractional arithmetic and its benefits to signal processing

When multiplying two 16-bit integers the results can often require 32 bits for the result. Rather than using integers, DSPs often use fractional arithmetic as it removes the need to deal with problematic overflows.

Signed fractions consist of 16- or 32-bit binary fractional values in twos complement form the range from -1 to less than 1 . Signed fractions in 1.15, or 1.31 format place the decimal point immediately to the right of the most significant bit. The largest representable value is $1 - 2^{-(n-1)}$ where n represents the number of bits in the value. The smallest representable value is -1 . Certain computations that produce values larger than $1 - 2^{-(n-1)}$ or smaller than -1 set OV in the SPEFSCR.

While multiplication of two 16-bit fixed point fractional numbers can also require 32 bits to display the result, the 32-bit result can be rounded to 16 bits introducing an error of only 2^{-16} . This is the method used by the LSP-APU. The format used in the examples in this application note is 1.15 (Frac16) and can represent numbers between -1 and 1 (2^{-15}) (1 cannot be represented exactly). Unfortunately the debugger and assemblers cannot represent the fractional values so the decimal equivalent is displayed. To convert from a 16-bit integer to a fractional number simply divide by 32768. [Figure 3](#) shows a comparison of Signed integer format and Frac 16 format.

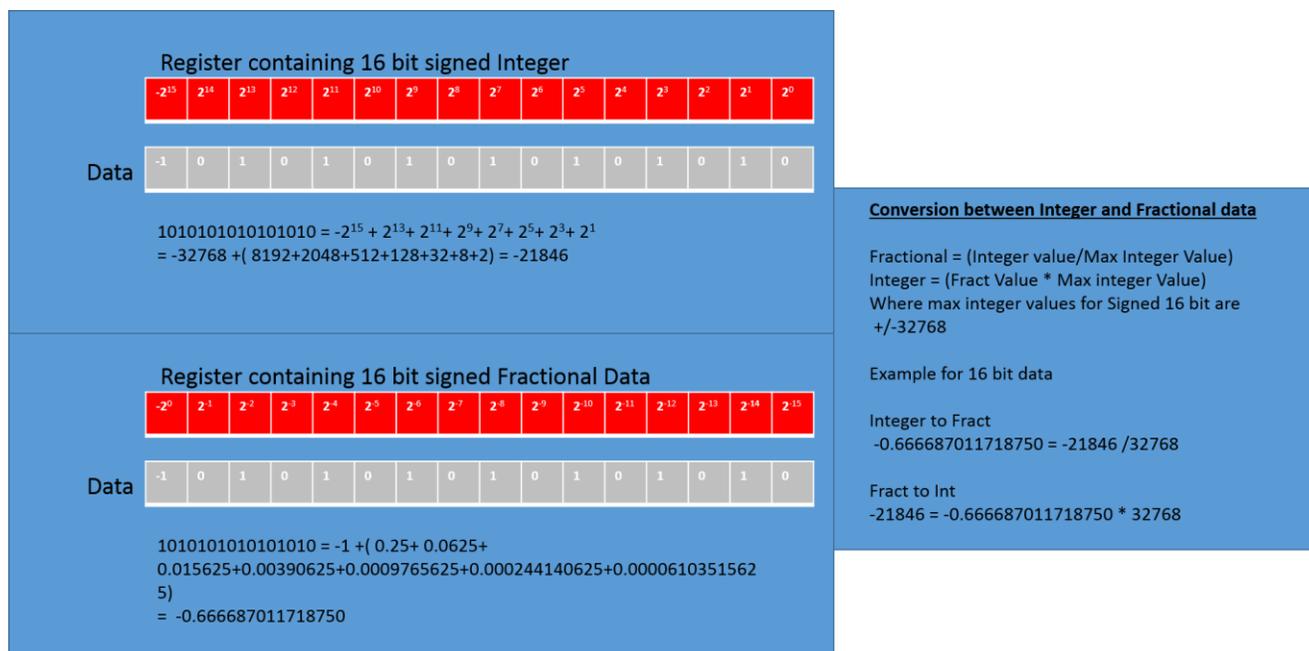


Figure 3. Example Multiplication of a Fractional number using LSP-APU Fractional arithmetic instructions

Multiplication of two signed fractional value causes the result to be shifted left one bit to remove the resultant redundant sign bit in the product. In this case, a 0 bit is concatenated as the least significant bit of the shifted result. This can be explained using the example shown below.

- Example operation to be carried out:
- $0.75 * 0.75 = 0.5625$

Multiplication example using Fractional Arithmetic instruction

- Fractional format 0.75 converted to 16-bit Integer = $(0.75 * 32768) = 24576 = 0x6000$
- Carry out multiplication $0x6000 * 0x6000 = 0x24000000$
- Result Rounded to Upper 16 bits = $0x2400$
- Convert back to fractional: $0x2400 / 32768 = 0.28125$ //Note this is half the correct result
- To obtain the correct result the LSP-APU left shifts the result one location
- $0x2400 = 0010\ 0100\ 0000\ 0000$ shift left 1 = $0100\ 1000\ 0000\ 0000 = 0x4800$
- Convert back to fractional: $0x4800 / 32768 = 0.5625$

4 FIR filter introduction

A finite impulse response (FIR) filter is a digital signal processing procedure used to obtain a particular impulse response of finite duration to an input through the digital suppression of unwanted components. The FIR filter of length N multiplies N delayed signal inputs by defined filter co-efficients and sums the results to produce a weighted average of the most recent input samples. The co-efficients values determine which signal components are suppressed by the filter. The filter co-efficients are most easily generated using a filter design software tool.

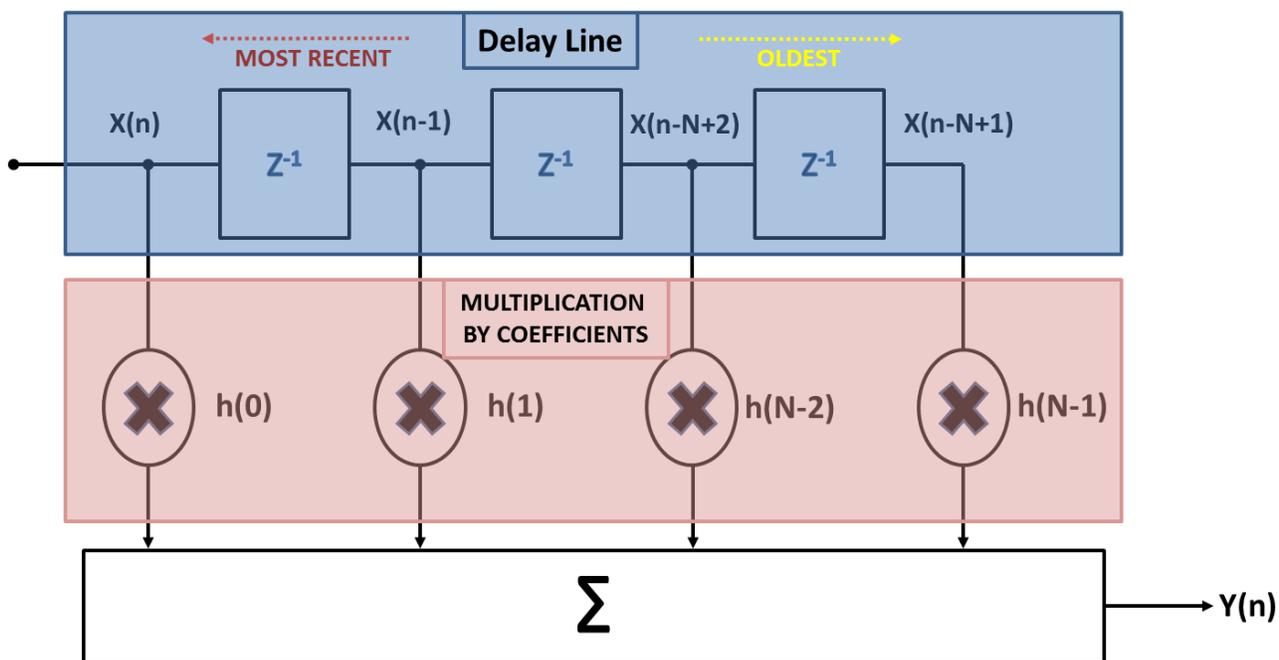


Figure 4. FIR Filter operation diagram

The FIR filter performs the following convolution equation

$$y(n) = h(n) * x(n)$$

Convolution is a multiplication of one sample by the other and then integrating the product. For discretely sampled time signals the integration is simply a sum, so the FIR filter can be represented by this form:

$$y(n) = \sum_{k=0}^{N-1} h(k) * (n - k)$$

An example of an N-tap filter is shown in [Figure 4](#) where N is the number of taps, h(k) is a vector of coefficients and x(n-k) is the input data array.

For example, the 20th element of the output vector of a 10-TAP filter can be represented as:

$$y(20) = h(0) \cdot x(20) + h(1) \cdot x(19) + h(2) \cdot x(18) \dots\dots\dots + h(9) \cdot x(11) \text{ Eqn. 1}$$

Thus, each entry in the output vector is the accumulated product of N multiplications of a coefficient by the corresponding delayed data sample. Note that the initial outputs y(0), y(1)y(N-2) are not valid until the delay line has been filled with sample x(N-1).

4.1 Creating a 10 tap filter with LSP instructions

The FIR filter can be efficiently implemented using the LSP. This section details how the FIR filter process can be split into LSP code sections, and then how the data is then manipulated within the 32-bit general-purpose registers (GPRs) by the LSP instructions. The diagrams can be used to aid understanding of the corresponding code sequences in the code appendix A. The software package associated with this application note also contains this code and the reader may wish to step through this code sequence to gain an insight into the operation of the LSP instructions.

Stage 1:

The first step of the LSP FIR filter is to populate the general purpose registers with the filter coefficients and the initial data. [Figure 1](#) and [Figure 2](#) illustrated show how the LSP allows the ordinary 32-bit General purpose registers to be used as two discrete 16-bit registers. [Figure 5](#) then shows how to use this principle to vector load the co-efficients and initial data into the GPR registers. The coefficients are loaded using a combination of zlhhsplat and zlhhsplatu instruction, and the data is loaded using a combination of zladd(u) and zlwwu instructions. The code for this sequence can be viewed in stage 1 of the LSP filter code shown in appendix A – FIR Filter written in LSP assembly.

Once the co-efficients and data have been loaded, the code enters the FIR filtering loop. The first stage in the filter loop is to load the new data consisting of 2 x 16-bit ADC results into a GPR register.

Stage 2:

The input data is multiplied by the odd co-efficients. Execution of the first MAC (Multiply Accumulate operation) instruction results in a vector containing two elements that reside in the 32-bit accumulator: $h[9].x[0]$ and $h[9].x[1]$.

The next MAC instruction multiplies the next vectors of data elements and odd coefficients and accumulates them with the previous results. Thus, with a single LSP-APU LPU-APU instruction, two MACs are executed per clock cycle. The operation is shown visually in [Figure 5](#) and the code sequence is located in stage 2 of Appendix A.

Stage 3:

To shift the data through the delay line, a series of vector merge instructions are executed. This is shown in [Figure 6](#), where the resultant vector is the offset of the original vector. The corresponding code sequence is shown at stage 3 of appendix A.

Stage 4:

Now that the data has been shifted one place through the delay line, a series of MAC instructions are executed except this time multiplying the even coefficients with the input data elements. The results are accumulated with those from stage 2. In this manner, two elements of the output vector are calculated in parallel as illustrated in [Figure 7](#).

Stage 5:

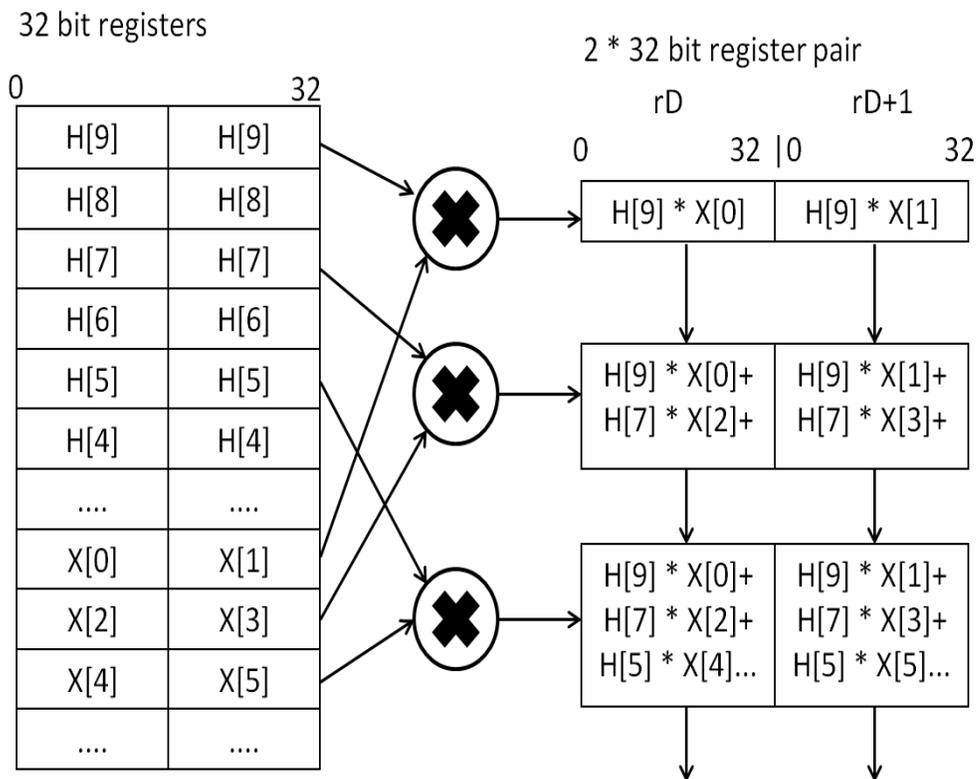
The calculation of the first two outputs of the filter are now stored in the output array using the `zstwhedu` instruction which stores the word as halfwords from the even elements in the register pair $Rs:RS+1$.

Stage 6:

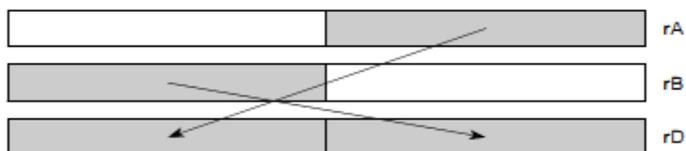
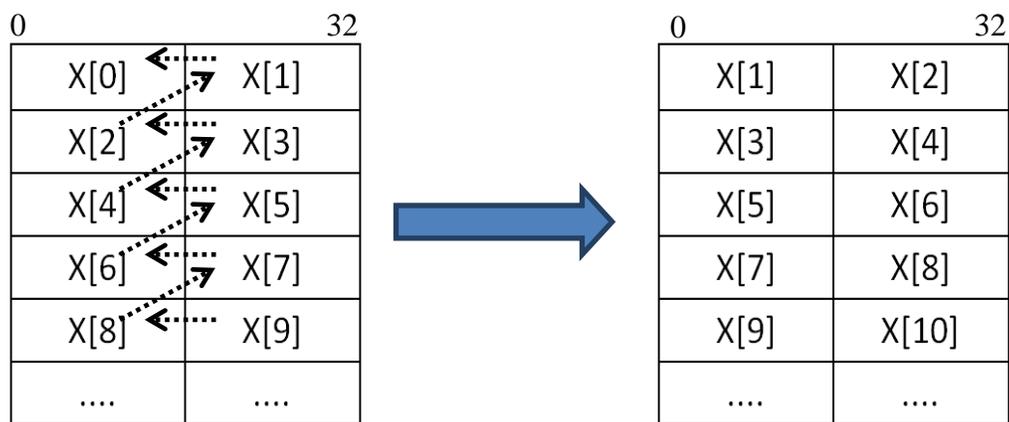
The data is shifted one position through the delay line, as per stage 3. This prepares the data for the MAC instructions at the start of the filter loop.

Stage 7:

The Counter address for the filter is incremented and compared with the intended number of input samples. If the filter has not completed it returns to the start of the main loop.



The delayed input data is multiplied and accumulated using the instruction “zvmhulfaas” - (vector multiply halfwords, upper/lower, signed fractional and accumulate with saturate)



The code example uses Vector merge low/high halfwords (zvmergelohih)

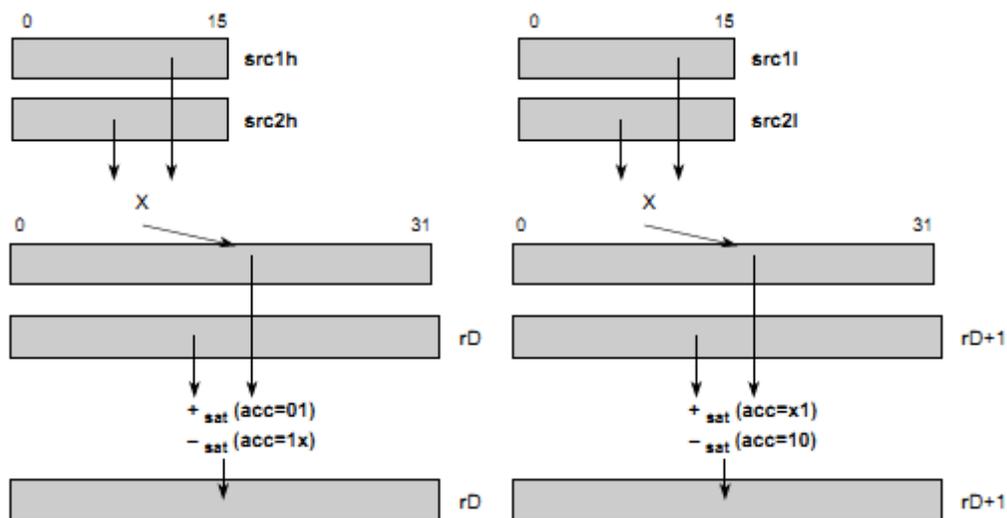


Figure 6. Vector merge instructions to delay data

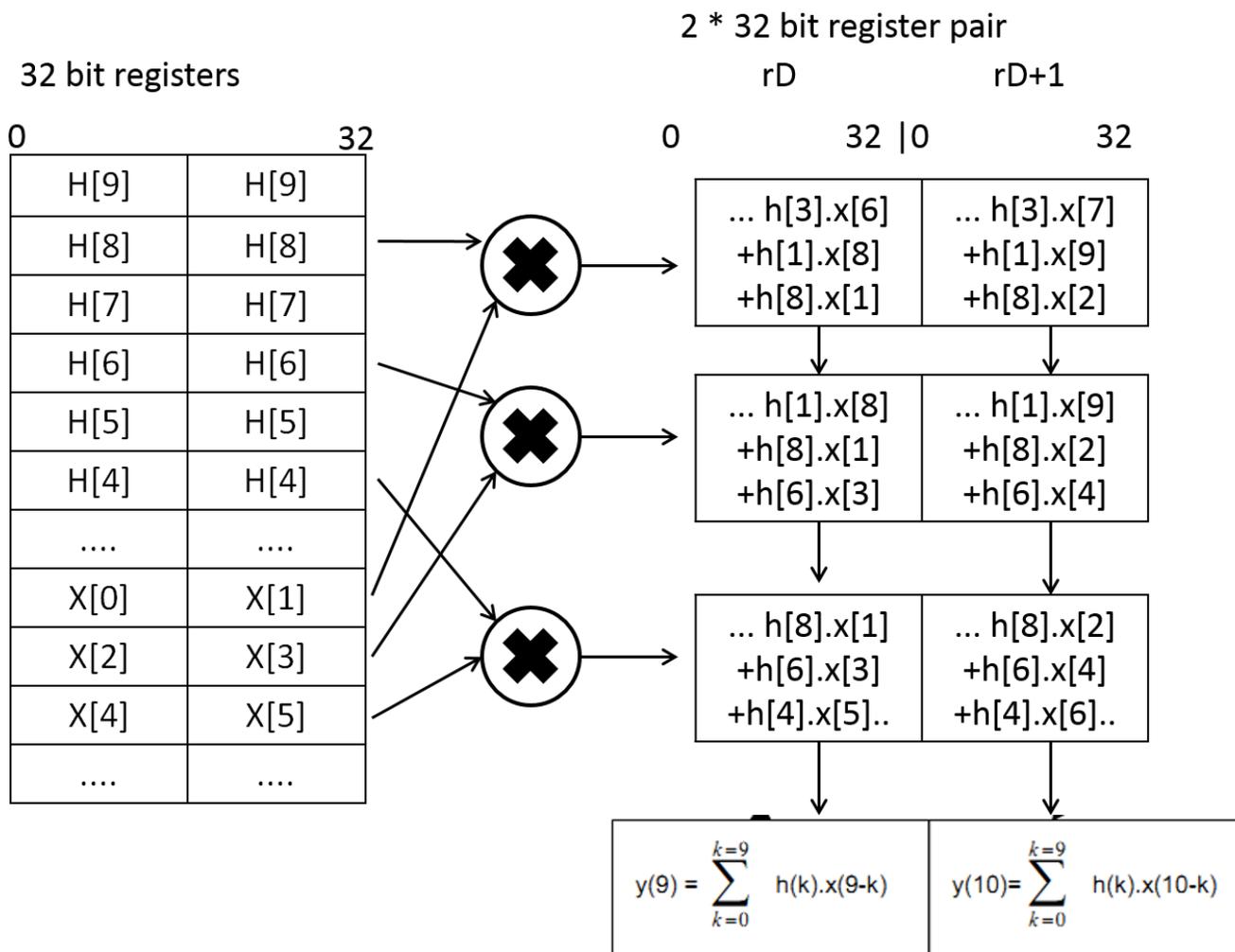


Figure 7. Multiplying and accumulating the delayed input data

4.2 LSP intrinsic instructions

There are two methods available to develop functions for the LSP APU. This can be directly using the LSP assembly instructions as detailed in the LSP APU Reference manual or using a C programming interface model, also known as compiler intrinsic functions. The compiler intrinsics operate as a function call that when compiled generates the LSP instruction that it represents. Coding with the intrinsic functions provides some benefits over using the assembly as the actual register allocation is decided by the compiler and the code generated is Power PC Embedded Application Binary Interface (EABI) compliant. The intrinsics coding environment is also more familiar to the user and abstracts some of the complexity from using the LSP. However because the compiler allocates register usage for each intrinsic used, this can lead to some inefficiencies. An intrinsic call does not strictly result in generation of a single assembly instruction when compiled. This can have a detrimental performance effect when compared with coding that utilizes only assembly instructions.

4.3 LSP intrinsic data types

The following section lists the supported LSP data types. The compiler treats LSP data types as integer types and uses the standard PowerPC ABI calling conventions. There are four kinds of data types:

- * Opaque data types are generic types accepted in most LSP intrinsic functions.
- * Transparent data types are accepted in place of opaque types, but are more specific and provide better display and debugging ability in MULTI.
- * Immediate data types are accepted for intrinsics that have immediates for arguments.
- * The Condition data type is accepted in vector comparison intrinsics.

Opaque Types

Name	Vector Size	Element Size (Bits)
__lsp16_16__	16	16
__lsp32_32__	32	32
__lsp32_16__	32	16
__lsp64_64__	64	64
__lsp64_32__	64	32

Transparent Types

Name	Opaque Type	Signedness	Format
__lsp16_s16__	__lsp16_16__	signed	integer
__lsp32_s32__	__lsp32_32__	signed	integer
__lsp32_s16__	__lsp32_16__	signed	integer
__lsp64_s64__	__lsp64_64__	signed	integer
__lsp64_s32__	__lsp64_32__	signed	integer
__lsp16_u16__	__lsp16_16__	unsigned	integer
__lsp32_u32__	__lsp32_32__	unsigned	integer
__lsp32_u16__	__lsp32_16__	unsigned	integer
__lsp64_u64__	__lsp64_64__	unsigned	integer
__lsp64_u32__	__lsp64_32__	unsigned	integer
__lsp16_uf16__	__lsp16_16__	unsigned	fraction 0.16
__lsp32_uf32__	__lsp32_32__	unsigned	fraction 0.32
__lsp32_uf16__	__lsp32_16__	unsigned	fraction 0.16

<code>__lsp16_sf16__</code>	<code>__lsp16_16__</code>	signed fraction 0.16
<code>__lsp32_sf32__</code>	<code>__lsp32_32__</code>	signed fraction 1.31
<code>__lsp32_sf16__</code>	<code>__lsp32_16__</code>	signed fraction 1.15
<code>__lsp32_9_23__</code>	<code>__lsp32_32__</code>	signed fraction 9.23
<code>__lsp64_9_23__</code>	<code>__lsp64_32__</code>	signed fraction 9.23
<code>__lsp64_33_31__</code>	<code>__lsp64_64__</code>	signed fraction 33.31
<code>__lsp64_17_47__</code>	<code>__lsp64_64__</code>	signed fraction 17.47

Immediate Types

Name	Bits	Signedness
<code>__lsp5_uimm__</code>	5	unsigned
<code>__lsp4_uimm__</code>	4	unsigned
<code>__lsp5_simm__</code>	5	signed
<code>__lsp2_offset__</code>	2	unsigned

Condition Type

Name	Bits
<code>__lsp4_crd__</code>	4

In addition to intrinsics explicitly defining individual LSP instructions, there exists a set of data manipulation intrinsics. These intrinsics act like functions with parameters that are passed by value. These intrinsics are classified into:

- Create intrinsics
 - These intrinsics create new generic 64-bit opaque data types from the given inputs passed by value. In the FIR example, the 64-bit opaque variable are initialized as:
 - `Coefficients1 = __ev_create_u32(h_ptr[0], h_ptr[0]);`
- Get intrinsics
 - These intrinsics allow the user to access data from within a specified location of the generic 64-bit opaque data type. In the FIR example, the upper and lower results within the 64-bit opaque variable `Accumulating_Product` are extracted as follows:
 - `y_ptr[n] = __ev_get_upper_u32(Accumulating_Product)`
 - `y_ptr[n+1] = __ev_get_lower_u32(Accumulating_Product)`
- Set intrinsics
 - These intrinsics provide the capability of setting values in a 64-bit opaque data type that

the intrinsic or the user specifies.

- Convert intrinsics
 - These intrinsics convert a generic 64-bit opaque data type to a specific signed or unsigned integer

4.4 FIR filter using intrinsics

The LSP 10 tap FIR filter written directly in assembly was re-written using LSP assembly intrinsics and is shown in Appendix B. A comparison between the original LSP assembly code and the intrinsics version should give the reader an insight into the usage of the intrinsics functions. The intrinsics FIR filter also has two different possible configurations. The first option, (A) simply reuses the intrinsic functions that are explicitly equivalent to the LSP assembly function to load the co-efficients and initial data into the registers. The second option (B) uses the Create intrinsics described in [above](#) section. The create intrinsics add readability to the code at the expense of execution time due to extra non-LSP instructions.

5 Performance comparisons

The performance of the different 10-tap FIR filters is shown in [Table 1](#).

Table 1. FIR filter performance comparison

FIR FILTER SOURCE TYPE	Compiler optimisations	Processor Cycle count	Compiled functions size (bytes)
C-Code	None	47620	89
C-Code	-Ospeed	10819	233
LSP Assembly	None (compiler optimization has no effect on LSP assembly instructions)	5639	174
LSP Intrinsics (Equivalent Intrinsics only)	None	6054	217
LSP Intrinsics (Equivalent Intrinsics only)	-Ospeed	4348	405
LSP Intrinsics (Using create intrinsics)	None	7946	541
LSP Intrinsics (Using create intrinsics)	-Ospeed	4669	539

The code provided in the associated software package for this application note was used to obtain these measurements. In the project all of the instructions and data for the three function types are located in the e200z425 local data and instruction memory, guaranteeing single cycle CPU read access for fastest code execution. The software was compiled using Green Hills software compiler version 201354. The optimization option (`-Ospeed`) enables optimizations that improve both size and performance, and additional optimizations that improve performance at the expense of size.

6 Summary

Using the LSP-APU to create digital filtering processing functions rather than C code can significantly reduce the execution time of the Filter functions. The examples given in this application note demonstrate how to create a simple 10 tap FIR filter using LSP assembly or Intrinsic functions. The performance benefits of the LSP-APU code versus the C-code are apparent using the 10 tap filter, and these benefits of using the LSP generally become even more significant as the complexity of the DSP function increases.

In addition to the FIR filtering examples in this application note, the LSP-APU can be also used to significantly improve the overall performance of the fixed-point arithmetic based applications, such as motor control tasks. As the motor control applications can be complicated to develop, Freescale provides the easy to use solution, the Automotive Math and Motor Control Library Set. This production ready library provides the customer with highly optimized building blocks to create the motor control application. It is available as precompiled object code for free at www.freescale.com/AutoMCLib. Moreover it is also available in licensed source code version, further utilizing the benefits of LSP-APU instruction set to increase the calculation speed.

Appendix A

A.1 10 tap FIR filter using LSP-APU assembly

```

# void fir_Signed_c(unsigned short N, short *x, short *y, short *h);
#fir_Signed16      (N, SDADC1_RESULTS, LSP_FIR_OUT, hr1);
.align 16
fir_Signed16:
#<# register definition
.equ N, r3
.equ x, r4
.equ y, r5
.equ h, r6

#co-efficients
.equ h0, r10
.equ h1, r11
.equ h2, r12
.equ h3, r13
.equ h4, r14
.equ h5, r15
.equ h6, r16
.equ h7, r17
.equ h8, r18
.equ h9, r19
#Data
.equ x0, r20
.equ x1, r21
.equ x2, r22
.equ x3, r23
.equ x4, r24
.equ x5, r25
.equ x6, r26
.equ x7, r27
.equ x8, r28
.equ x9, r29
.equ x10,r30
.equ temp, r8
.equ temp1,r9
.equ cnt, r7

```

```

#>#
#-----
#----- SAVING CONTEXT -----
# store nonvolatile registers
e_stwu    r1, -28(r1)    # stwu - store with update - r1 contains stack pointer. In this case local DMEM
# store r25 to r31 onto stack
e_stmw    r25, 0(r1)    # stmw - store multiple word - r 14 ro r31 are volatile and as such must be saved and restored if used in the
routine
#-----
#Initialize counter to zero
e_li cnt,0                # clear counter
#-----
#----- STAGE 1 : LOADING THE CO-EFFICIENTS -----
##load the co-efficients
zlhsplat h0,0(h)          # CO-EFFICIENT1   # zlhsplat = Vector Load halfword into halfwords and splat  rD,d(rA)
zlhsplatu h1, 2(h)        # CO-EFFICIENT2   # zlhsplatu = Vector Load halfword into halfwords and splat [with update] rD,d(rA)
zlhsplatu h2, 2(h)        # CO-EFFICIENT3   # In both of the zlhsplat[u] instructions the halfword addressed by rA is "splatted"
                        # across the destination word rD
zlhsplatu h3, 2(h)        # CO-EFFICIENT4   #such that rD = rArA. In the case of the zlhsplatu instruction the update immediate "u"
                        # means that rA is updated to
zlhsplatu h4, 2(h)        # CO-EFFICIENT5   #the effective address + 2 bytes prior to the "splat"execution.
zlhsplatu h5, 2(h)        # CO-EFFICIENT6
zlhsplatu h6, 2(h)        # CO-EFFICIENT7
zlhsplatu h7, 2(h)        # CO-EFFICIENT8
zlhsplatu h8, 2(h)        # CO-EFFICIENT9
zlhsplatu h9, 2(h)        # CO-EFFICIENT10
#-----
#----- :LOADING THE INITIAL DATA-----
se_subi   y, 4            # decrement y pointer - This is because zstwhedu instruction in the output loop has an
                        # immediate update so the pointer must be pre decremented by 1 word prior to reaching # loop
#Load the X-data
zldd x0, 0(x)              #Loads two 32-bit registers (x0 and x1) with the 16-bit results of the first 4 ADC conversions
zldd x2, 8(x)              #Loads two 32-bit registers (x2 and x3) with the 16-bit results of the next 4 ADC conversions
zlw w x4, 8(x)             #Loads one 32-bit register (x4) with the 16-bit results of the next 2 ADC conversions
#-----
#----- MAIN LOOP -----
loop:
zlw w x5, 4(x)             # load in the first new data - this is not multiplied yet but is merged in during stage 4
#----- STAGE 2 : MAC THE ODD COEFFICIENTS-----
# multiply and accumulate using odd coefficients

```

Summary

```

zvmhulsf temp, h9, x0      # InputVector_0 * Coefficients 10
zvmhulsfaas temp, h7, x1   # InputVector_1 * Coefficients 8
zvmhulsfaas temp, h5, x2   # InputVector_2 * Coefficients 6
zvmhulsfaas temp, h3, x3   # InputVector_3 * Coefficients 4
zvmhulsfaas temp, h1, x4   # InputVector_4 * Coefficients 2
#----- STAGE 3 : MERGE THE INPUT VECTORS-----
#----- merge to rotate input data vectors so that the even coefficients multiply the corresponding delayed data -----
zvmergelohih x0,x0,x1     # InputVector0 = Merge InputVector_0 & InputVector_1
zvmergelohih x1,x1,x2     # InputVector1 = Merge InputVector_1 & InputVector_2
zvmergelohih x2,x2,x3     # InputVector2 = Merge InputVector_2 & InputVector_3
zvmergelohih x3,x3,x4     # InputVector3 = Merge InputVector_3 & InputVector_4
zvmergelohih x4,x4,x5     # InputVector4 = Merge InputVector_4 & InputVector_5 ----- "New" vector comes in here
#----- STAGE 4 : MAC THE EVEN COEFFICIENTS-----
# multiply and accumulate using even coeffs
zvmhulsfaas temp, h8, x0      # InputVector_0 * Coefficients 9
zvmhulsfaas temp, h6, x1      # InputVector_1 * Coefficients 7
zvmhulsfaas temp, h4, x2      # InputVector_2 * Coefficients 5
zvmhulsfaas temp, h2, x3      # InputVector_3 * Coefficients 3
zvmhulsfaas temp, h0, x4      # InputVector_4 * Coefficients 1
#----- STAGE 5 : STORE OUTPUT TO ARRAY-----
zstwhedu temp,4(y)          # Update y address pointer by 4 immediately and then store two 16 bit results
#----- STAGE 6 : MERGE THE INPUT VECTORS-----
#merge to rotate input data vectors
zvmergelohih x0,x0,x1     # InputVector0 = Merge InputVector_0 & InputVector_1
zvmergelohih x1,x1,x2     # InputVector1 = Merge InputVector_1 & InputVector_2
zvmergelohih x2,x2,x3     # InputVector2 = Merge InputVector_2 & InputVector_3
zvmergelohih x3,x3,x4     # InputVector3 = Merge InputVector_3 & InputVector_4
zvmergehiloh x4,x5,x5     # InputVector4 = InputVector_5 -----"New" vector comes in here
#----- STAGE 7 : LOOP END TEST-----
e_addi cnt, cnt, 2
cmpw cnt, N
e_bne loop
#-----
#----- STAGE 8 RESTORE CONTEXT -----
# restore non-volatile regs
e_lmw r25, 0(r1)
# delete stack frame
e_addi r1, r1, 28
se_blr

```

Appendix B

B.1 FIR filter written with LSP intrinsics

```

void fir_frac16_LSP_trans_intrinsic(uint16_t N, int16_t *x_ptr, int16_t *y_ptr, int16_t *h_ptr)
{
    __lsp32_sf16__ InputVector0, InputVector1, InputVector2, InputVector3, InputVector4, InputVector5;

    __lsp32_sf16__ Coeff1, Coeff2, Coeff3, Coeff4, Coeff5, Coeff6, Coeff7, Coeff8, Coeff9, Coeff10;

    __lsp64_32__ Accumulating_Product;
    int n = 0;

    //-----
    //----- STAGE 1 : LOADING THE CO-EFFICIENTS -----
    // ----- Option A Load the coefficients in to registers using direct LSP equivalent instructions ---
    //
    #ifdef Full_Intrinsics

    Coeff1 = __zlhhsplat(h_ptr, 0);
    Coeff2 = __zlhhsplatu(h_ptr, 2);
    Coeff3 = __zlhhsplatu(h_ptr, 4);
    Coeff4 = __zlhhsplatu(h_ptr, 6);
    Coeff5 = __zlhhsplatu(h_ptr, 8);
    Coeff6 = __zlhhsplatu(h_ptr, 10);
    Coeff7 = __zlhhsplatu(h_ptr, 12);
    Coeff8 = __zlhhsplatu(h_ptr, 14);
    Coeff9 = __zlhhsplatu(h_ptr, 16);
    Coeff10 = __zlhhsplatu(h_ptr, 18);

    //----- Option B Load the co-efficients using LSP create intrinsics -----

    #else
    //example usage of create intrinsics
    //__lsp32_16__ __zlhhsplat __ARGS(( void * a, __lsp5_uimm__ b )) __ATTRIBUTE__((pure));

    Coeff1 = __lsp_create_32_16(h_ptr[0], h_ptr[0]);
    Coeff2 = __lsp_create_32_16(h_ptr[1], h_ptr[1]);
    Coeff3 = __lsp_create_32_16(h_ptr[2], h_ptr[2]);
    Coeff4 = __lsp_create_32_16(h_ptr[3], h_ptr[3]);
    Coeff5 = __lsp_create_32_16(h_ptr[4], h_ptr[4]);
    
```

Summary

```

Coeff6 = __lsp_create_32_16(h_ptr[5], h_ptr[5]);
Coeff7 = __lsp_create_32_16(h_ptr[6], h_ptr[6]);
Coeff8 = __lsp_create_32_16(h_ptr[7], h_ptr[7]);
Coeff9 = __lsp_create_32_16(h_ptr[8], h_ptr[8]);
Coeff10 = __lsp_create_32_16(h_ptr[9], h_ptr[9]);

#endif

//#-----
//#----- : LOADING THE INITIAL DATA-----
// Option A: Load the first set of data to be processed using direct LSP equivalent instructions
#ifdef Full_Intrinsics
    InputVector0 = __zlwu(x_ptr,0);
    InputVector1 = __zlwu(x_ptr,4);
    InputVector2 = __zlwu(x_ptr,8);
    InputVector3 = __zlwu(x_ptr,12);
    InputVector4 = __zlwu(x_ptr,16);
#else
//Option B: Load the first set of data to be processed using LSP equivalent instructions -----

// Load the first set of data to be processed
InputVector0 = __lsp_create_32_16(x_ptr[n+0], x_ptr[n+1]) ;
InputVector1 = __lsp_create_32_16(x_ptr[n+2], x_ptr[n+3]) ;
InputVector2 = __lsp_create_32_16(x_ptr[n+4], x_ptr[n+5]) ;
InputVector3 = __lsp_create_32_16(x_ptr[n+6], x_ptr[n+7]) ;
InputVector4 = __lsp_create_32_16(x_ptr[n+8], x_ptr[n+9]) ;

// //if defined LSP instructions
#endif

for (n=0; n<N; n+=2)
{

#ifdef Full_Intrinsics
InputVector5 = __zlwu(x_ptr,16);
x_ptr += 2;
#else
InputVector5 = __lsp_create_32_16(x_ptr[n+10],x_ptr[n+11]) ;
#endif

//----- STAGE 2 : MAC THE ODD COEFFICIENTS-----

```

```

Accumulating_Product = __zvmhulsf(InputVector0, Coeff10) ;
Accumulating_Product = __zvmhulsfaas(Accumulating_Product, InputVector1, Coeff8) ;
Accumulating_Product = __zvmhulsfaas(Accumulating_Product,InputVector2, Coeff6) ;
Accumulating_Product = __zvmhulsfaas(Accumulating_Product,InputVector3, Coeff4) ;
Accumulating_Product = __zvmhulsfaas(Accumulating_Product,InputVector4, Coeff2) ;

// ----- STAGE 3 : MERGE THE INPUT VECTORS-----
// ----- merge to rotate input data vectors so that the even coefficients -----
// ----- multiply the corresponding delayed data -----

InputVector0 = __zvmergelohih(InputVector0, InputVector1);
InputVector1 = __zvmergelohih(InputVector1, InputVector2);
InputVector2 = __zvmergelohih(InputVector2, InputVector3);
InputVector3 = __zvmergelohih(InputVector3, InputVector4);
InputVector4 = __zvmergelohih(InputVector4, InputVector5);

// ----- STAGE 4 : MAC THE EVEN COEFFICIENTS-----
Accumulating_Product = __zvmhulsfaas(Accumulating_Product,InputVector0, Coeff9);
Accumulating_Product = __zvmhulsfaas(Accumulating_Product,InputVector1, Coeff7);
Accumulating_Product = __zvmhulsfaas(Accumulating_Product,InputVector2, Coeff5);
Accumulating_Product = __zvmhulsfaas(Accumulating_Product,InputVector3, Coeff3);
Accumulating_Product = __zvmhulsfaas(Accumulating_Product,InputVector4, Coeff1);

// ----- STAGE 5 : STORE OUTPUT TO ARRAY-----

__zstwhed (Accumulating_Product, y_ptr,0);
y_ptr =y_ptr+2;

// ----- STAGE 6 : MERGE THE INPUT VECTORS-----

InputVector0 = __zvmergelohih(InputVector0, InputVector1);
InputVector1 = __zvmergelohih(InputVector1, InputVector2);
InputVector2 = __zvmergelohih(InputVector2, InputVector3);
InputVector3 = __zvmergelohih(InputVector3, InputVector4);
InputVector4 = __zvmergehiloh(InputVector5, InputVector5);
}
}
#pragma ghs section vletext=default

```

B.1.1 FIR Filter written in C

```
void fir_frac16_c(uint16_t N, int16_t *x, int16_t *y, int16_t *h)
{
    int32_t n, k;
    int32_t acc;

    for (n = 0; n < N; n++)
    {
        acc = 0;
        for (k = 0; k < 10; k++)
        {
            acc += h[k] * x[n+k];
        }
        y[n] = (int16_t) (acc >> 15);
    }
}
```


How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale and the Freescale logo are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. All other product or service names are the property of their respective owners.

© 2015 Freescale Semiconductor, Inc.

Document Number: AN5092
Rev. 1
05/2015