Freescale Semiconductor
Application Note

AN3733
Rev. 0, 07/2008

Improving DSP Performance Using Efficient SPE Coding

by: Robert Moran
Applications Engineering
Microcontroller Solutions Group

The signal processing engine (SPE) on the MPC5500 family of devices allows multiple data operations to be performed by a single instruction. This feature is aimed at DSP operations, where the use of the SPE will increase system performance. The fundamentals of the SPE are explained within this application note, along with the benefits to system performance that this module offers.

To achieve optimal performance from the SPE, consideration must be given to implementation of the software. This application note details techniques that should be used to gain this optimal performance. There are also guidelines and examples to aid the user in developing an SPE function for a DSP algorithm. With these guidelines, the user should be able to develop DSP functions using the SPE that will improve overall system performance.

Contents

1 Introduction 1
2 SPE Instruction Set 2
2.1 Instruction Overview 2
2.2 Breaking Down Mnemonics 3
3 SPE Performance Improvements Over Book E 6
3.1 SPE Versus Book E — Simple Example 6
3.2 Instruction Timing 9
3.3 SPE versus Book E Timing Comparison 12
4 Techniques for Writing Efficient SPE 13
4.1 Controlling Instructions Using SPE Assembly 13
4.2 Scheduling Instructions to Reduce Stalls 13
4.3 Rolling Out Small Loops 16
4.4 Aligning SPE Instructions 16
4.5 Register Usage 17
5 Guide to Writing SPE Functions 17
6 Summary 18
Appendix A Multiplication of Two Matrices 19
Appendix B SPE Function Examples 20

1 Introduction

It is possible to improve system performance for DSP operations on the MPC5500 family by using the signal

processing engine (SPE) auxiliary processing unit (APU). The SPE APU is designed to accelerate signal-processing applications normally suited to DSP operation. This acceleration is accomplished using short (two-element) vectors within 64-bit general purpose registers (GPRs) and using single-instruction multiple-data (SIMD) operations to perform the requisite computations. The SPE also architects an accumulator register to allow for back-to-back operations without loop unrolling.

The SPE has its own set of dedicated instructions, which generally take elements from each source register and operate on those elements with the corresponding elements of a second source register (and/or the accumulator). The results of these operations are placed in the destination register and/or the accumulator. Instructions that are vector in nature (that is, they produce results of more than one element) provide results for each element that are independent of the computation of the other elements. For example, the SPE allows the addition of two 32-bit values in parallel.

The SPE APU uses the GPRs within the e200z3 and e200z6 core. The GPRs are implemented as 64-bit registers, although Book E instructions use only the lower 32 bits. When SPE instructions are executed they use the full range of the 64-bit GPRs. The SPE APU instructions view the 64-bit register as being composed of a vector of two elements, each of which is 32 bits wide. (Some instructions read or write 16-bit elements.) The most significant 32 bits are called the upper word, high word, or even word. The least significant 32 bits are called the lower word, low word, or odd word. Unless otherwise specified, SPE instructions write all 64 bits of the destination register.

The SPE APU is not a coprocessor, and therefore does not perform SPE instructions in parallel to the e200z3/6 core. It uses the same pipeline and is subject to the same restrictions in bandwidth as the standard Book E instruction set. The advantage that the SPE offers is that it can perform multiple data operations for a single instruction, whereas Book E is limited to one data operation per instruction.

The SPE also implements floating-point instructions, which allows the use of vector and non-vector single precision floating-point operations. This application note will be focusing only on vector operations — more information on floating-point instructions can be found in the signal-processing engine chapter of the e200z3/6 core manuals.

2 SPE Instruction Set

2.1 Instruction Overview

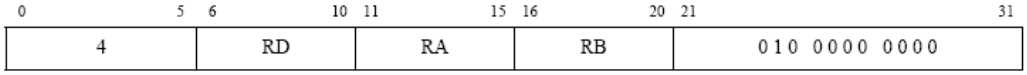
SPE instructions that perform vector operations begin with the mnemonic “ev” — this represents a 64-bit vector instruction. The syntax of SPE instructions follows the same form as Book E instructions. For data operations the instruction is represented by “instruction mnemonic, destination register, source register A, source register B.” The syntax of load and store instructions also uses the same form as Book E instructions.

The syntax of a simple SPE instruction is shown in Figure 1.

evaddw **evaddw**

Vector Add Word

evaddw **rD,rA,rB**




```
RD[0:31] = RA[0:31] + RB[0:31]                // Modulo sum
RD[32:63] = RA[32:63] + RB[32:63]             // Modulo sum
```

Adds each element of **rA** to the corresponding element of **rB** and places the results into the corresponding elements of **rD**. The sum is a modulo sum.

Figure 1. SPE Add Instruction

The instruction above will perform an add operation between the lower 32 bits of RA and RB, while also performing an add operation between the upper 32 bits of RA and RB. This is visually represented in Figure 2. The “evaddw” instruction takes one clock cycle to execute.

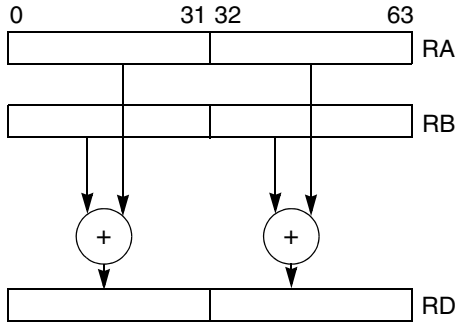


Figure 2. Addition of Two 32-bit Values Using “evaddw” Instruction

The corresponding instruction in Book E is “add RD, RA, RB.” However, this instruction performs only a 32-bit add and therefore requires an additional add instruction to replicate the “evaddw” instruction.

The clear difference here is that two add instructions, and therefore two clock cycles, are required to perform the same operation as one “evaddw” instruction, carried out in one clock cycle.

In the context of a DSP function that consists of a small loop processing a large quantity of data, a significant number of clock cycles can be saved per loop by using the SPE vector instructions, and hence significantly increase the system performance.

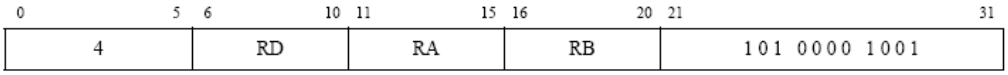
2.2 Breaking Down Mnemonics

There are several SPE instructions that at first sight look very complex. These instructions can be broken down and simplified to make them easier to understand. This is best illustrated by using an example of the “evmhesmiaaw” instruction illustrated in Figure 3.

evmhesmiaaw **evmhesmiaaw**

Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate into Words

evmhesmiaaw **rD,rA,rB** (M=1, O=0, F=0, S=1)




```
temp1[0:31] = rA[0:15] *si rB[0:15]
temp2[0:31] = rA[32:47] *si rB[32:47]

temp3[0:32] = ACC[0:31] + temp1[0:31]
temp4[0:32] = ACC[32:63] + temp2[0:31]

ACC[0:31] = rD[0:31] = temp3[1:32]
ACC[32:63] = rD[32:63] = temp4[1:32]
```

Figure 3. Example of the "evmhesmiaaw" Instruction

The "evmhesmiaaw" mnemonic can be broken into several elements.

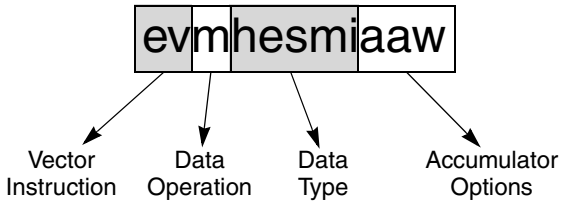


Figure 4. Elements of an SPE instruction

As shown in Figure 4, in SPE instructions there are generally three elements that differ:

- Data Operation: The operation that will be performed on the data. In this example it is a "multiply" operation (represented by the "m").
- Data Type: This relates to the properties of the data that are expected by the instruction. In this example the instruction expects signed modulo integer data (represented as "smi") that is half-word in size (represented by the "h"), of which each half-word is stored in the even side (represented by the "e") of each 32-bit element of the 64-bit register.
- Accumulator Options: Allows the accumulator to be set to the result of the instruction, or, as in this example, adds the result of the instructions to the contents of the accumulator.

The operation of this instruction is illustrated in Figure 5.

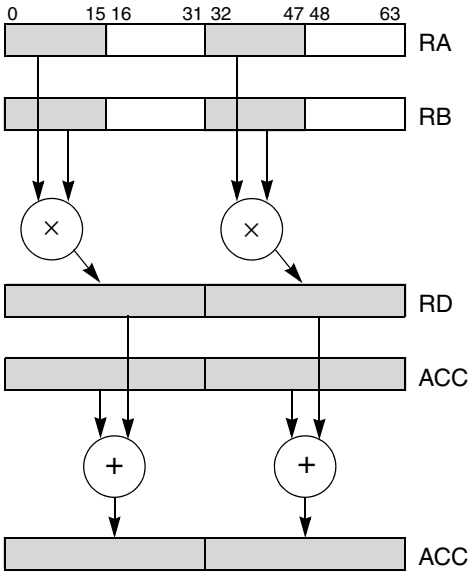


Figure 5. Multiply and Accumulate of Two 16-bit Values Using "evmhesmiaaw" Instruction

As shown above, the complex mnemonics can be broken down into a form that is easier to understand. This can make it much simpler to select the exact instruction needed for an operation. For example, there are approximately 80 "multiply" instructions, all of which are subtly different, but each operation can be described in a similar fashion. The difference between the many variations is generally the type of data used and the location of this data in the 64-bit vector. A programmer is allowed a great deal of flexibility to find the correct instruction for a specific function.

At a high level the different instructions can be broken down into several fields:

- Simple integer instructions
  - — Basic logical/mathematical operations — add, subtract, and, or, etc.
  - — Data manipulation — merge, splat, shift, compare, etc.
- Load and store instructions
- Complex integer instructions
  - — Multiply and/or accumulate
  - — Add/subtract and/or accumulate
  - — Divide

It is beyond the scope of this application note to detail the specific instructions listed above. However, a full description can be found in the document *EREF: A Programmer's Reference Manual for Freescale Embedded Processors (Including the e200 and e500 Families)* available from the Freescale website.

3 SPE Performance Improvements Over Book E

3.1 SPE Versus Book E — Simple Example

As mentioned previously, the SPE provides a performance enhancement because it allows multiple data operations in a single cycle. The example below shows a comparison of SPE assembly versus Book E assembly for a very simple multiply-and-accumulate operation. The SPE instructions have not been optimized in this example, so that it is easier to see the initial performance enhancement gained by using the SPE over Book E. Optimization techniques for the SPE instructions will be detailed later in this application note.

Example 1.

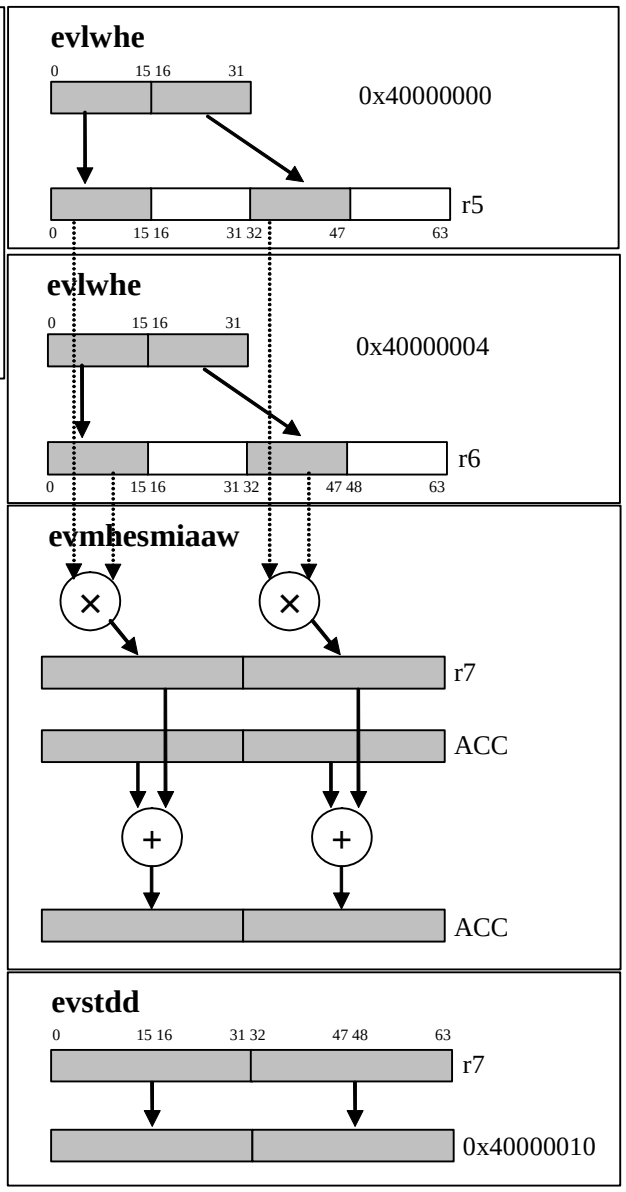
Perform a multiply-and-accumulate of two 16-bit values with two other 16-bit values, which are stored side by side at start location 0x40000000 (assume the address is pre-loaded into r4) and store to location 0x40000010 (assume the address is pre-loaded into r3).

SPE Implementation:

SPE Instructions

```
evlwhe          r5, 0x0 (r4)
evlwhe          r6, 0x4 (r4)
evmhesmiaaw     r7, r5, r6
evstdd          r7, 0x0 (r3)
```

Corresponding Data Operations

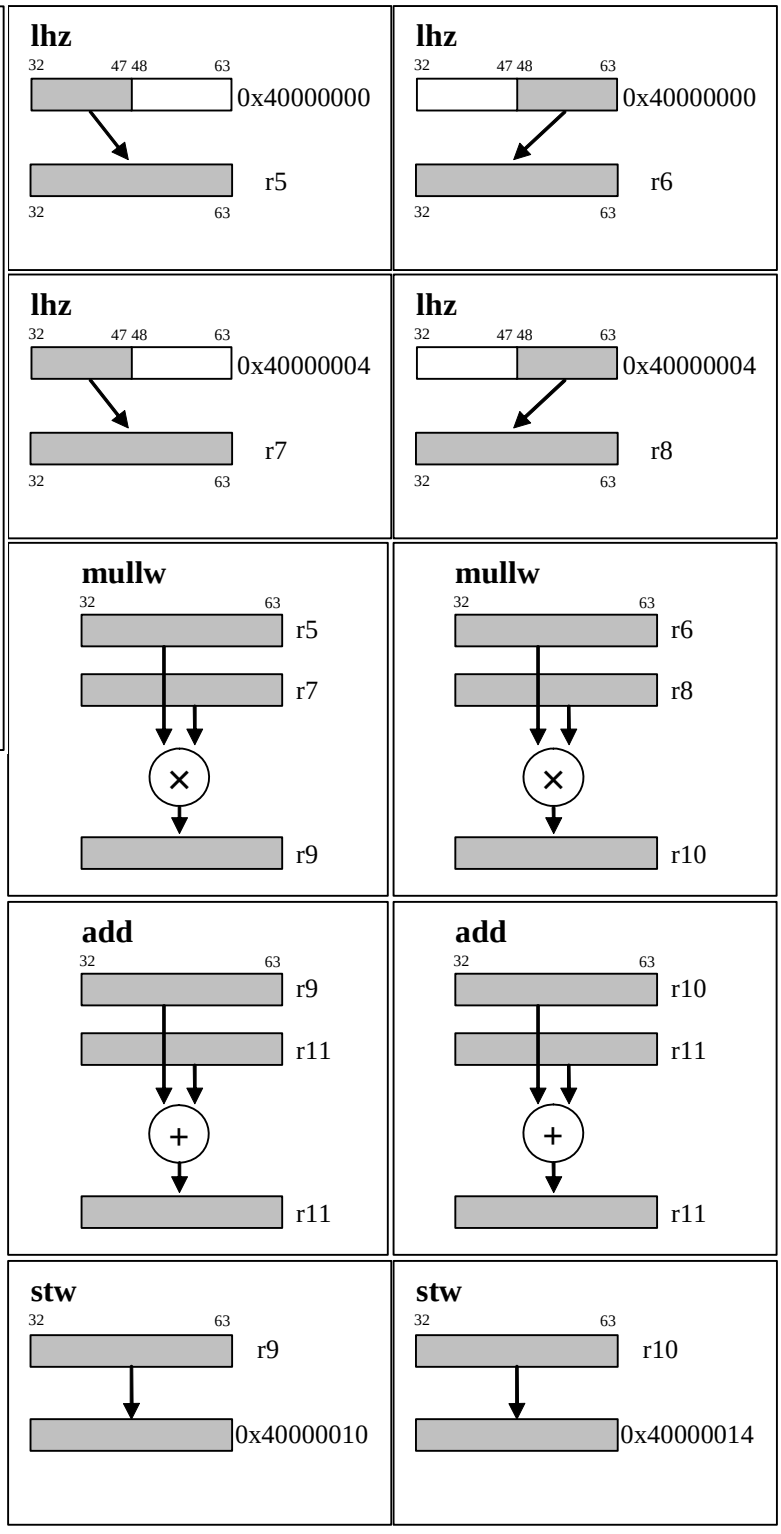


Book E Implementation:

Book E Instructions

```
lhz         r5, 0x0 (r4)
lhz         r7, 0x4 (r4)
mullw       r9, r5, r7
add         r11, r9, r11
stw         r9, 0x0 (r3)
lhz         r6, 0x2 (r4)
lhz         r8, 0x6 (r4)
mullw       r10, r6, r8
add         r11, r10, r11
stw         r10, 0x4 (r3)
```

Corresponding Data Operations



Comparing the two different implementations, it can clearly be seen that the SPE coding uses fewer instructions and fewer operations than the equivalent Book E coding. This indicates the performance impact that the SPE has over Book E. This performance impact is further illustrated by looking at the instruction execution time. To perform a theoretical timing comparison, the execution of the instructions through the core pipeline must be understood in detail.

3.2 Instruction Timing

Instructions are processed in the core using the instruction pipeline. MPC55xx devices that implement the e200z6 core use a seven-stage pipeline, whereas devices with the e200z3 core use a four-stage pipeline. This chapter will focus on the seven-stage pipeline. However, the principles discussed can also be applied to the four-stage pipeline.

The seven-stage pipeline is split into seven stages, consisting of four different functions. A diagram of the seven-stage pipeline is shown in Figure 6:

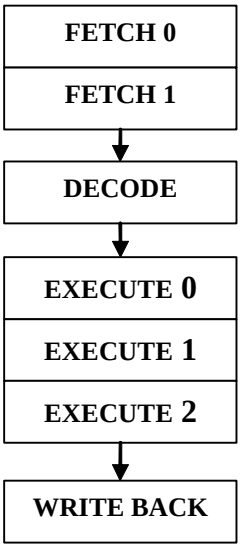


Figure 6. Seven Stages of e200z6 Pipeline

The details of the pipeline are as follows:

- Two instruction fetch stages

  As many as two instructions are fetched per clock cycle and placed in a seven-entry instruction buffer.
- Instruction decode stage

  Each instruction takes one cycle to decode and is dispatched at the end of the decode stage.

- Three execution stages

  A three-stage execution pipeline includes feed-forwarding, which allows dependent instructions to continue through the pipeline. All instructions, including branch instructions, pass through all three stages of the execute pipeline, in order and in single file. The execution units within the core are used at this stage to complete the specific task of the instruction. These execution units include the Integer unit, the Load/Store unit, the SPE APU unit, and the Branch unit.
- Result write-back stage

  This is where the results are committed to architected registers (such as GPRs) and instructions are deallocated from the instruction pipeline.

Instructions are fed into the pipeline one at a time, each stage containing only one instruction at a time. Furthermore, instructions cannot bypass any stage of the pipeline and must flow in a linear manner. This is illustrated in Figure 7.

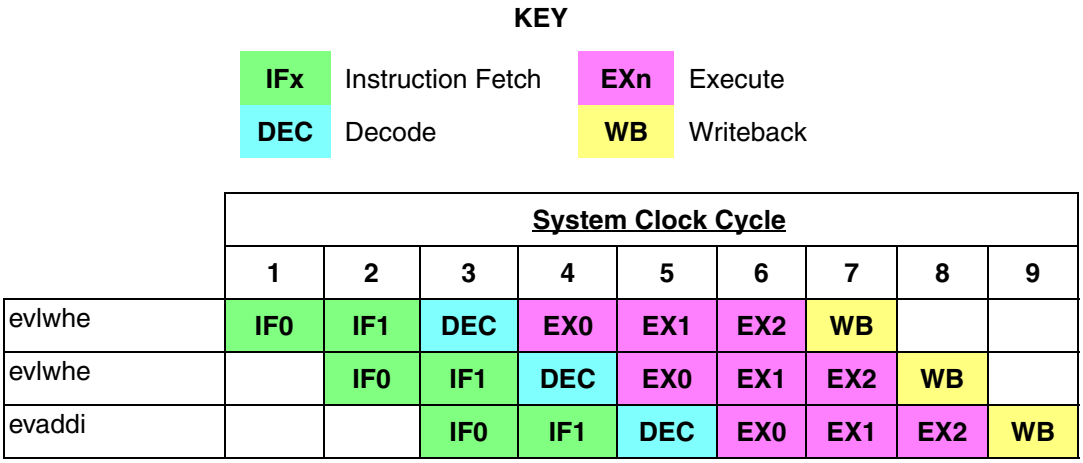


Figure 7. Illustration of Instructions Progressing Through Pipeline

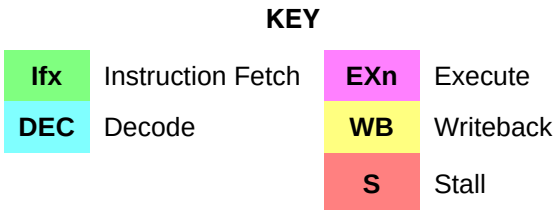
Figure 7 also illustrates that instructions are fed into the pipeline on every new cycle. This means that, presuming no delays (discussed later), an instruction will complete on every clock cycle, thereby achieving single-cycle throughput.

All instructions have an execution time associated with them; this execution time can be broken down to throughput and latency. Throughput of an instruction is the time taken by an instruction before the next instruction can be executed.

There is one condition that is of the highest importance: if the next instruction depends on an output from the previous instruction then it will have to wait additional cycles before it can begin execution. The overall time this would take is referred to as latency.

If the core has to wait before it can begin the execution of the next instruction due to the latency of the previous instruction, then this is referred to as stalling the pipeline. Stalls in the pipeline have a performance impact, as no new instructions are being executed when the pipeline has stalled. This problem can be alleviated by rescheduling instructions so as to reduce the effects of instruction latency. Details of this technique are described later in Section 4, “Techniques for Writing Efficient SPE.” Further details of the pipeline can be found in the “Instruction Pipeline and Execution Timing” chapter of the respective e200z3/6 core manuals.

An example of an instruction stall is shown in Figure 8. This example uses a multiply instruction followed by a store instruction. The store instruction is saving the result of the previous multiply, and therefore needs to wait for the multiply instruction to provide its result (in other words, complete the execution stage) before the store can begin its execution.



	System Clock Cycle									
	1	2	3	4	5	6	7	8	9	10
evmhesmiaaw	IF0	IF1	DEC	EX0	EX1	EX2	WB			
evstdd		IF0	IF1	DEC	S	S	EX0	EX1	EX2	WB

Figure 8. Latency Example

Figure 8 shows that the store instruction cannot begin its execution until the multiply instruction has completed its execution in cycle seven. Therefore the store instruction does not complete until three cycles after the multiply instruction has completed. This means that the stall introduced a latency of three cycles.

The throughput and latency of each instruction can be found in the “Instruction Timings” section of the core manual for either the e200z3 or the e200z6.

Writing efficient code for the SPE involves looking at the execution timing of instructions in great detail. The essential element of this timing is the latency. The timing analysis above is more complex than is needed and can be summarized by looking only at the latency of the instructions. Figure 9 shows a simpler representation of instruction timing, looking only at latency, and provides the user with a quick method to identify stalls in the pipeline.

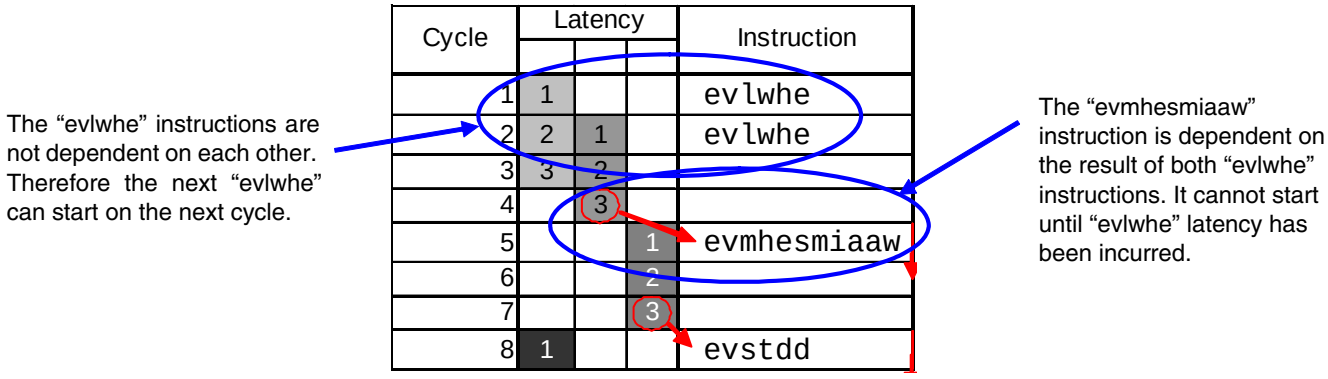


Figure 9. Performance of SPE versus Book E

Figure 9 details how the latency of instructions affects the overall instruction timing. Instructions that are dependent on the latency of a previous instruction are highlighted by the red circle and arrow. Multiply, load, and store SPE instructions have a throughput of one cycle and a latency of three cycles. This means

that any non-dependent instruction which follows a multiply instruction can begin on the next clock cycle, that is, the throughput time. However, any instruction dependent on the output of the multiply instruction cannot begin until three cycles after the multiply instruction, in other words, the latency.

This can be further illustrated by looking at Figure 9. On the second cycle the SPE instruction being executed is “evlwhe.” This has a throughput of one cycle and a latency of three cycles. The next instruction is “evmhesmiaaw,” which uses the output of the two previous “evlwhe” instructions. This means that it has to wait until both “evlwhe” instructions have propagated fully through the pipeline. As the second “evlwhe” occurs on the second cycle and has a latency of three cycles, its output will not be valid until the fifth cycle. This is when the “evmhesmiaaw” instruction can begin execution. This means, though, that no new instructions are being executed on the third or fourth cycle. Therefore two stalls were introduced to the pipeline.

This timing analysis of the pipeline is not fully accurate, as it does not take into account any potential delays introduced by memory wait states or crossbar contention. However, in the context of writing efficient SPE instructions, this method provides an effective and quick tool for writing optimal code.

3.3 SPE versus Book E Timing Comparison

Referring back to the example shown in Section 3.1, “SPE Versus Book E — Simple Example,” the benefit that the SPE offers can be illustrated further by performing a timing comparison between the two implementations (this is illustrated in Figure 10). The SPE column details which clock cycle the SPE instructions are executed on, while the Book E column details the execution of the Book E instructions.

Cycle	Latency			SPE	Latency			BookE
	A	B	C		A	B	C	
1	1			evlwhe	1			lhz
2	2	1		evlwhe	2	1		lhz
3	3	2			3	2		
4		3				3		
5			1	evmhesmiaaw			1	mullw
6			2				2	
7			3				3	
8	1			evstdd	1			add
9	2					1		stw
10	3					2	1	lhz
11					1	3	2	lhz
12					2		3	
13					3			
14						1		mullw
15						2		
16						3		
17							1	add
18					1			stw
19					2			
20					3			

Figure 10. Performance of SPE versus Book E

As we see in Figure 10, the number of clock cycles required to complete the SPE instructions is 10, whereas the number of clock cycles required to complete the equivalent Book E instructions is 20. This small example illustrates that the SPE has an improvement in system performance over Book E. This system performance can be further improved in larger DSP operations where the effects of instruction latency are reduced.

4 Techniques for Writing Efficient SPE

There are techniques that can be used when writing SPE functions to improve the efficiency of the code.

- Controlling instructions using SPE assembly
- Scheduling instructions to reduce stalls
- Rolling out small loops
- Aligning SPE instructions
- Register usage

4.1 Controlling Instructions Using SPE Assembly

Currently there are two different ways that SPE instructions can be implemented. They can be coded using assembly or by using the C intrinsics defined in the *Signal Processing Engine Auxiliary Processing Unit Programming Interface Manual* (SPEPIM) available at the Freescale.com website.

The SPE-PIM defines a set of intrinsics with approximately one intrinsic for each available SPE instruction (with some additional intrinsics for data and register manipulation). Each intrinsic acts as a function call that, when compiled, will generate the SPE instruction that it represents.

However, the actual register allocation is delegated to the compiler, and the code generated using the SPE intrinsics will be PowerPC EABI compliant. Thus use of the SPE-PIM to produce SPE code is considerably easier than programming directly in assembly.

The one drawback of the SPE intrinsics is that there is less control over the instruction sequencing compared to using assembly. Because the compiler defines the register usage for the intrinsics, this can lead to inefficiencies in timing that would not be present when using assembly.

Choosing which implementation to use is a decision for the system designer. The SPE intrinsics are much easier to integrate into a C environment, and they offer performance improvements over the equivalent C implementation. However, for the best possible performance, assembly allows the flexibility to fully optimize the instruction timings and achieve the greatest efficiency. For this reason, the examples used in this application note will be referring to the assembly implementation of SPE instructions.

4.2 Scheduling Instructions to Reduce Stalls

One of the key elements in writing efficient SPE functions is to minimize the amount of stalls that will occur in the pipeline. Section 3.2, “Instruction Timing,” details how the throughput and latency of SPE instructions can introduce stalls into the pipeline. These can be minimized by scheduling the instructions in a particular manner so that the stalls are padded out with nondependent instructions.

Rescheduling the instructions to reduce the number of stalls can save several clock cycles on a small piece of code. This will particularly benefit applications that use a loop of code to process large quantities of data. For example, take a sixth-order FIR that is stored in a loop and used to process pressure data over two engine revolutions (for example, 720 data samples). If four instructions can be saved per loop due to re-scheduling, then potentially 2880 clock cycles could be saved during the processing of all of the data samples.

There are two stages to reducing the number of stalls in the assembly language code:

1. Identifying where stalls will occur
2. Re-scheduling the instructions

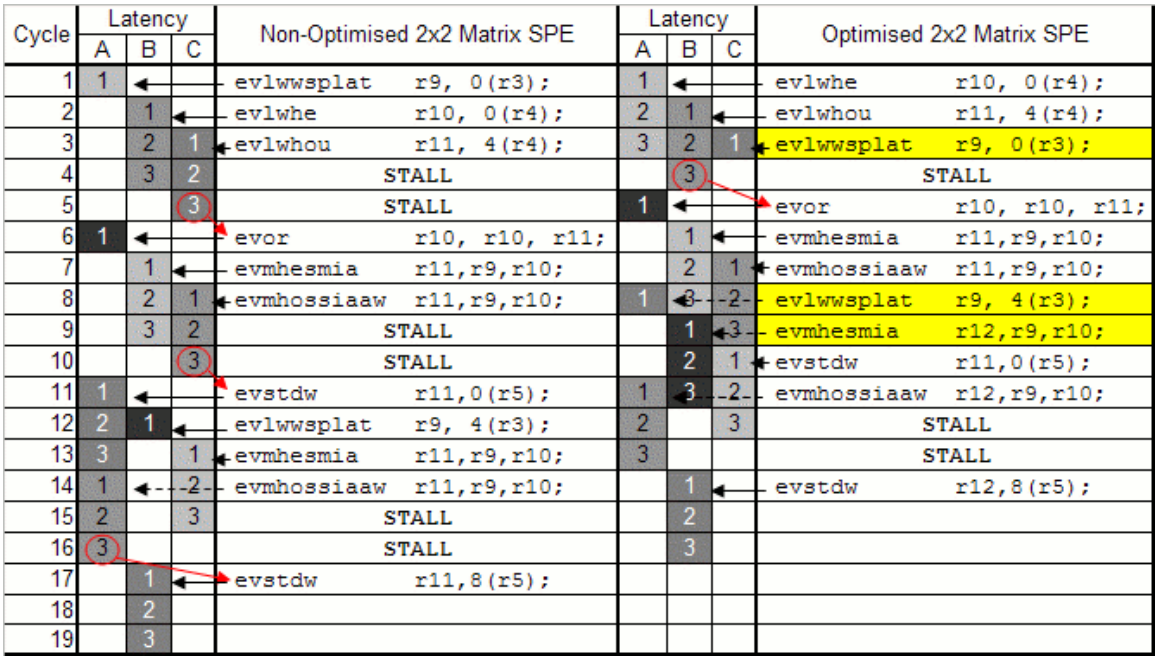
4.2.1 Identifying Stalls

Stalls will generally occur after instructions that have a latency of more than one cycle. When using non-floating-point SPE, this refers mainly to the multiply (and accumulate), load, and store instructions, which all have a latency of three cycles. All floating-point (vector or non-vector) instructions, with the exception of the divide instruction, have a latency of three cycles.

A common pattern in DSP algorithms is to perform multiply-and-accumulate (MAC) operations. The results of the MAC are then stored to a memory location. This is often captured in a loop, with a MAC instruction and a store instruction occurring on each loop iteration.

The implementation of this technique when using SPE assembly is often a multiply instruction followed by a store instruction. The store instruction depends on the output of the multiply instruction — therefore the pipeline will be stalled between the multiply and the store.

An example to illustrate this can be taken from Appendix A, Example A-1. This is a multiplication of two matrices. The left side of Figure 11 shows the timing of this example when the instruction scheduling has not been optimized. It can be seen that there are three places where the pipeline stalls.



Cycle	Latency A	Latency B	Latency C	Non-Optimised 2x2 Matrix SPE	Latency A	Latency B	Latency C	Optimised 2x2 Matrix SPE
1	1			evlwwsplat r9, 0(r3);	1			evlwhe r10, 0(r4);
2		1		evlwhe r10, 0(r4);	2	1		evlwhou r11, 4(r4);
3		2	1	evlwhou r11, 4(r4);	3	2	1	evlwwsplat r9, 0(r3);
4		3	2	STALL		3		STALL
5			3	STALL	1			evor r10, r10, r11;
6	1			evor r10, r10, r11;		1		evmhesmia r11,r9,r10;
7		1		evmhesmia r11,r9,r10;		2	1	evmhossiaaw r11,r9,r10;
8		2	1	evmhossiaaw r11,r9,r10;	1	3	2	evlwwsplat r9, 4(r3);
9		3	2	STALL		1	3	evmhesmia r12,r9,r10;
10			3	STALL		2	1	evstdw r11,0(r5);
11	1			evstdw r11,0(r5);	1	3	2	evmhossiaaw r12,r9,r10;
12	2	1		evlwwsplat r9, 4(r3);	2		3	STALL
13	3		1	evmhesmia r11,r9,r10;	3			STALL
14	1		2	evmhossiaaw r11,r9,r10;		1		evstdw r12,8(r5);
15	2		3	STALL		2		
16	3			STALL		3		
17		1		evstdw r11,8(r5);				
18		2						
19		3						

Figure 11. Timing Comparison of Non-Optimized versus Optimized Instruction Scheduling

4.2.2 How to Reschedule the Instructions

After the stalls have been identified, the next task is to reschedule the instructions to reduce the number of stalls.

The most common way to reschedule instructions is to move load instructions between any multiply instructions and their dependent store instructions. Because DSP operations often involve several MAC and store operations one after the other, the stalls in the pipeline between these two instructions can be used to load values for the next operation.

For example, the right side of Figure 11 illustrates how the matrix operations in Appendix A, "Multiplication of Two Matrices," could be optimized to reduce the number of stalls. The two stalls before the "evor" instruction exist because the instruction depends on the completion of the previous "evlwhe" and "evlwhou" instructions. However, it does not depend on the "evlwwsplat" instruction. Therefore this instruction can be moved into the area where one stall existed and will save a clock cycle.

The second set of stalls exist between a multiply instruction ("evmhossiaaw") and a store instruction ("evstdw"). This is where the values for the next MAC operations can be loaded. In this case the "evlwwsplat" and "evmhesmia" instructions are rescheduled to the areas where both stalls existed. One point to make here is that the destination register for the second set of MAC operations has changed from r11 to r12. This allows the values to be loaded for the second set of MAC operations before the first set of operations have completed. If this had not been done, and the "evmhesmia" instruction on cycle 9 used r11, then the result being saved to memory on the following store instruction would be incorrect.

In this example it was not possible to fill the final two stalls. However, if this routine was implemented in a loop then this would be the ideal place to reschedule the first two load instructions, thereby eliminating the two stalls.

4.3 Rolling Out Small Loops

A very simple but effective method for increasing performance when using SPE assembly instructions is to roll out small loops. Some DSP functions require a small loop of code to process a large quantity of data. A second-order FIR is a good example of this; it consists of two loads, two MAC and two stores for each data point. This loop could be run several hundred times in a typical application.

Each loop takes roughly 10 cycles to complete. However, there is a branch instruction at the end of every loop that needs to be considered. This branch instruction adds an additional three cycles per loop. In this instance the branch instruction adds a roughly 25% performance hit on every loop. The way to reduce this impact is to make the loops more linear, which can be done by increasing the number of times the operations of the loop are repeated before a branch instruction is taken.

Using the second-order FIR as an example, if the load, MAC, and store operations were performed four times each loop (as shown in Figure 12), then the overall number of clock cycles per loop would be roughly 40. This means that the impact of a three-cycle branch is reduced to 7.5%, thereby reducing the overall impact.

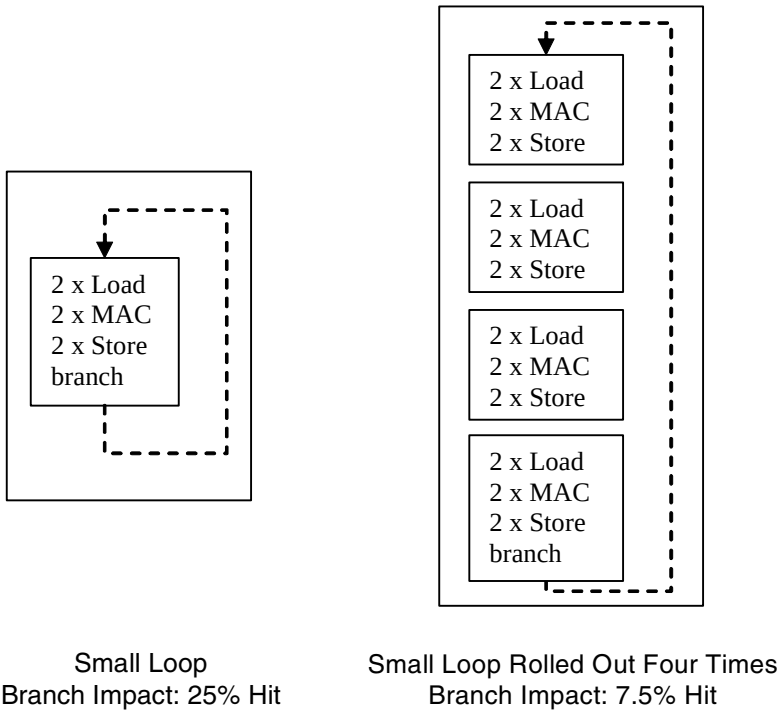


Figure 12. Rolling Out a Small Loop to Reduce the Impact of a Branch

4.4 Aligning SPE Instructions

The alignment of data that is used by SPE instructions is important for improving system performance.

On the MPC5554 and MPC5553 devices, all double-word accesses to data must be 64-bit aligned. An alignment exception will occur if any double-word load or store access is made to a non-64-bit aligned memory address.

On all other current MPC55xx family devices, a double-word access to data must be at least 32-bit aligned. It is recommended that the data be aligned on a 64-bit access for the most efficient throughput. If the data is 32-bit aligned but not 64-bit aligned, then two fetches to the memory would be required to gather the double-word data. This obviously has an impact on performance.

Another strong argument for aligning double-word data accesses to 64-bit boundaries is for compatibility. If software was written for MPC5566, which did not use 64-bit alignment, then it would not compatible with the MPC5554 and MPC5553 devices, on which an alignment exception would occur.

4.5 Register Usage

SPE instructions abide by the embedded application binary interface (EABI) for the use of general-purpose registers. This is also true for Book E instructions. In addition, it is common to save and restore the GPRs in accordance with the EABI when entering and exiting an SPE assembly function. For a large SPE function taking several thousand clock cycles to complete, the time required to save and restore registers has little impact on the overall system time. When a smaller task is performed, which may take tens of clock cycles, then the time taken for the save and restore routine becomes more significant.

One aspect of the EABI definitions is the difference between volatile (GPR3–GPR12) and nonvolatile registers (GPR14–GPR31). The contents of a nonvolatile register need to be saved if one of these registers is used in an SPE assembly function. This is the opposite of the volatile registers, which do not need to be saved between sub-routines.

For this reason, volatile registers can be used when developing SPE assembly functions. If they are used, then there is no need to save or restore them when entering and exiting the function. This saves several clock cycles.

For example, consider an SPE function that requires eight GPRs. If the nonvolatile registers GPR14–GPR22 were used for this function, then the save-and-restore routine would amount to an overhead of roughly 16 clock cycles. If the volatile registers GPR4–GPR12 were used, then the save-and-restore routine would not be required. Therefore there would be no overhead for using these registers.

5 Guide to Writing SPE Functions

For a required DSP operation, the procedure given here offers a high-level guide to writing an efficient SPE assembly function to implement the DSP operation.

1. Determine whether the DSP operation is suited to the SPE.

   SPE takes advantage of vector operations aligned on 64-bit boundaries. If the DSP operation required allows a great deal of parallelism, then it is most likely suited to the SPE. On the other hand, if the function requires a lot of data operations on non-64-bit aligned data, then the overhead of realigning data may outweigh the advantage of the vector operations. In this case, the SPE may not offer a distinct advantage.

   Floating-point operations are the exception to this situation, and would benefit greatly from the use of the SPE.

2. Derive a formula for the DSP operation and graphically map the flow of data through the SPE function.

   Look to see where data is suited for parallel operations, and take into account how data needs to be loaded (in other words, odd/even) to allow this technique.

   Ensure that data operations are suited to being 64-bit aligned to maximize the effectiveness of the SPE.

3. Implement the graphical representation as SPE instructions.

   Write the function in accordance with the flow of the equation. The main aim at this stage is to write the function in SPE assembly and verify that it performs the desired task, while also keeping it as simple as possible to understand. Do not try to reschedule the instructions at this stage, as it may add unnecessary complexity at this stage in the code development..

4. After the SPE assembly function has been written and tested, apply the techniques for improving efficiency.

   Plot the theoretical timing for the function. Identify where there are stalls in the code and try to reschedule other instructions to reduce the number of stalls.

   If the function has a short repetitive loop, evaluate if it is suited to being rolled out, which will reduce the effect of a branch instruction.

There are two examples in Appendix B, "SPE Function Examples," which detail how a fourth-order FIR SPE function (Example B-1, "Fourth-Order FIR Filter") and a simple two-by-two matrix multiplication SPE function (Example B-2, "Two-by-Two Matrix Multiplication") have been created by using these guidelines.

6 Summary

There are significant performance benefits to be gained by using the SPE for DSP operations. This application note has detailed the techniques that can be used to gain optimal performance with SPE instructions, the benefits of which, compared to standard C and non-optimal SPE encoding, are highlighted. These techniques will provide an efficient set of guidelines for the user to develop SPE functions that achieve optimal DSP performance.

Appendix A Multiplication of Two Matrices

Example A-1.

Multiply 16-bit matrices A x B when:

```
A = [00][01]    B = [00][01]
    [10][11]        [10][11]
```

r3 = input address of matrix A

r4 = input address of matrix B

r5 = output address

SPE assembly flow:

```
#/* load r9  = A[0][0] | A[0][1] | A[0][0] | A[0][1]     */
#/* load r10 = B[0][0] | 0x0000  | B[0][1] | 0x0000     */
#/* load r11 = 0x0000  | B[1][0] | 0x0000  | B[1][1]    */
#/* load r10 = B[0][0] | B[1][0] | B[0][1] | B[1][1]    */
#/*      A[0][0]*B[0][0] ||     A[0][0]*B[0][1]          */
#/* ACC_HI + A[0][1]*B[1][0] || ACC_LO + A[0][1]*B[1][1] */
#/* store to C[0][0] and C[0][1]                         */
#/* load r9  = A[1][0] | A[1][1] | A[1][0] | A[1][1]    */
#/*      A[1][0]*B[0][0] ||     A[1][0]*B[0][1]          */
#/* ACC_HI + A[1][1]*B[1][0] || ACC_LO + A[1][1]*B[1][1] */
#/* store to C[1][0] and C[1][1]                         */
```

Non-Optimized SPE assembly code:

```
evlwwsplat     r9, 0(r3);
evlwhe         r10, 0(r4);
evlwhou        r11, 4(r4);
evor           r10, r10, r11;
evmhesmia      r11, r9, r10;
evmhossiaaw    r11, r9, r10;
evstdw         r11, 0(r5);
evlwwsplat     r9, 4(r3);
evmhesmia      r12, r9, r10;
evmhossiaaw    r12, r9, r10;
evstdw         r12, 8(r5);
```

Optimized SPE assembly code:

```
evlwhe         r10, 0(r4);
evlwhou        r11, 4(r4);
evlwwsplat     r9, 0(r3);
evor           r10, r10, r11;
evmhesmia      r11, r9, r10;
evmhossiaaw    r11, r9, r10;
```

```
evlwwsplat      r9, 4(r3);
evmhesmia       r12, r9, r10;
evstdw          r11, 0(r5);
evmhossiaaw     r12, r9, r10;
evstdw          r12, 8(r5);
```

Appendix B SPE Function Examples

Example B-1. Fourth-Order FIR Filter

1. **Determine if the DSP operation is suited to the SPE.**

   The majority of operations in an FIR are multiply-and-accumulate.

   There are no restrictions on the data that do not allow 64-bit alignment for the data.

   Therefore this DSP operation is suited to the SPE.

2. **Derive a formula for the DSP operation and graphically map the flow of data through the SPE function.**

   The equation for a fourth-order FIR is:

$$y(n) = \sum_{k=0}^{3} h(k) \times x(n-k)$$

   For 2 sample points, y(8) and y(9), the equation can be broken out to:

   y(8) = h(0) × x(8) + h(1) × x(7) + h(2) × x(6) + h(3) × x(5)

   y(9) = h(0) × x(9) + h(1) × x(8) + h(2) × x(7) + h(3) × x(6)

   This can be represented graphically:

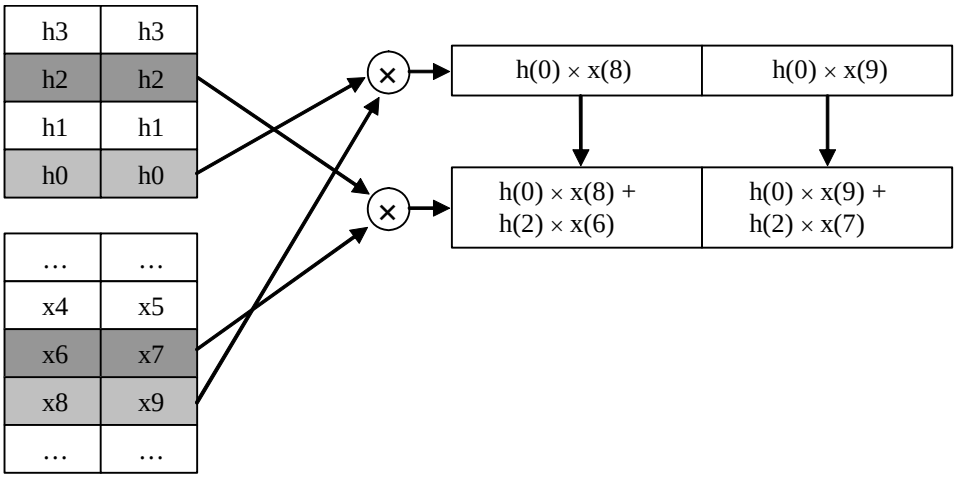


Figure 13. Multiply and Accumulate Two Coefficients

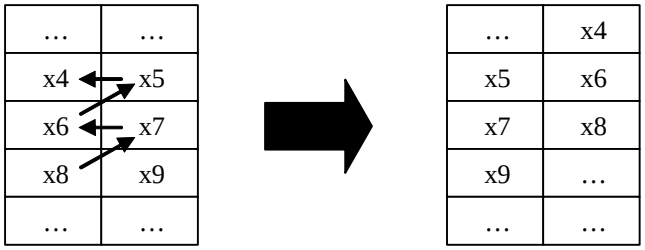


Figure 14. Shift Input Data by 32 Bits (Can Use a Merge Instruction to Implement)

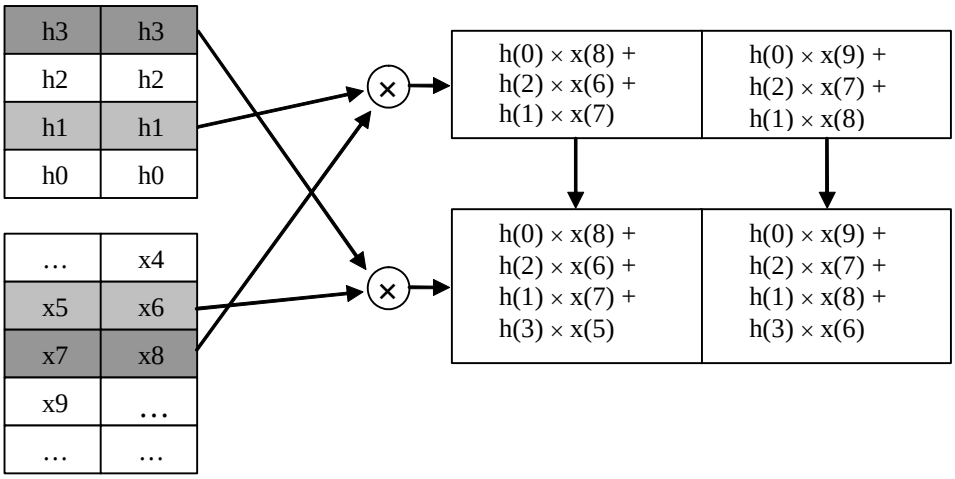


Figure 15. Multiply and Accumulate Remaining Two Coefficients

3. Implement the graphical representation as SPE instructions.

Assume:

r3 = Address of first element in output array “y.” Aligned to 64-bit boundary.

r4 = Address of first element in input array “x.” Aligned to 64-bit boundary.

r5 = Address of first element of coefficients “h.” Aligned to 64-bit boundary.

Input data is 16-bit signed integer.

Output data is 32-bit signed integer.

```
order4_FIR:

stwu        r1, -0x8(r1);         #/* Save Non-Volatile r16 */
evstdd      r16, 0x0(r1);         #/* Save Non-Volatile r16 */

addi        r16, r0, 75           #/* Loop to return 150 outputs */
addi        r3, r3, 0x4;          #/* Point yptr to 1st output */
subi        r4, r4, 0x4;          #/* xptr set before 1st input */
```

```
evlhhousplat r9,  0x0(r5);          #/* Load h(0) */
evlhhousplat r10, 0x2(r5);          #/* Load h(1) */
evlhhousplat r11, 0x4(r5);          #/* Load h(2) */
evlhhousplat r12, 0x6(r5);          #/* Load h(3) */

#------------------------------

Loop_begin:

evlwhou       r6, 0x0(r4);          #/* Load x(4/5) */
evlwhou       r7, 0x4(r4);          #/* Load x(6/7) */
evlwhou       r8, 0x8(r4);          #/* Load x(8/9) */

evmhosmia     r5, r8, r9;           #/* (x(8/9) x h0), store to ACC */
evmhosmiaaw   r5, r7, r11;          #/* (x(6/7) x h2) + ACC */

evmergelohi   r8, r7, r8;           #/* Use reg for x(7/8)*/
evmergelohi   r7, r6, r7;           #/* Use reg for x(5/6)*/

evmhosmiaaw   r5, r7, r12;          #/* (x(5/6) x h3) + ACC */
evmhosmiaaw   r5, r8, r10;          #/* (x(7/8) x h1) + ACC */

evstwho       r5, 0x0(r3);          #/* Store to y(8) & y(9) */


#------------------------------

addi          r3, r3, 0x4;          #/* update xptr */
addi          r4, r4, 0x4;          #/* update yptr */
cmpwi         r16, 0x0              #/* Check if loop should end */
subi          r16, r16, 0x1         #/* Decrement loop end */

bne           Loop_begin            #/* Branch to start of loop */

evldd         r16, 0x0(r1);         #/* Restore Non-Volatile r16 */
addi          r1, r1, 0x8;          #/* Save stack */

blr                                 #/* Return to function call */
```

4. **After the SPE assembly function has been written and tested, apply the techniques for improving the efficiency.**
   - Loop is rolled out three times to create a larger loop that outputs 6 “y” values.
   - Instructions are rescheduled to reduce number of stalls in pipeline.
   - Align load and store instructions to 64-bit boundaries to avoid non-aligned performance hits.
   - Use all possible volatile registers before using any nonvolatile registers.

   Timing comparison from non-optimized FIR loop to the optimized FIR loop is shown in Figure 16.

Cycle	Latency			Non-Optimised 4th Order FIR Loop		Latency			Optimised 4th Order FIR Loop	
	A	B	C			A	B	C		
1		1		evlwhou	r6, 0x0(r4);	3	1		evmergelohi	r18, r6, r7;
2		2	1	evlwhou	r7, 0x4(r4);			1	evmhosmia	r5, r7, r11;
3	1	3	2	evlwhou	r8, 0x8(r4);	1		2	evmhosmiaaw	r5, r8, r9;
4	2		3	**STALL**		2	1	3	evmergelohi	r17, r7, r8;
5	3			**STALL**		3		1	evmhosmiaaw	r5, r18, r12;
6		1		evmhosmia	r5, r8, r9;	1		2	evmhosmiaaw	r5, r17, r10;
7		2	1	evmhosmiaaw	r5, r7, r11;	2	1	3	evlwhou	r6, 0xC(r4);
8	1	3	2	evmergelohi	r8, r7, r8;	3	2		**STALL**	
9		1	3	evmergelohi	r7, r6, r7;		3		evstwho	r5, 0x0(r3);
10			1	evmhosmiaaw	r5, r7, r12;			1	evmhosmia	r5, r8, r11;
11	1		2	evmhosmiaaw	r5, r8, r10;	1		2	evmhosmiaaw	r5, r6, r9;
12	2		3	**STALL**		2	1	3	evmergelohi	r18, r8, r6;
13	3			**STALL**		3		1	evmhosmiaaw	r5, r17, r12;
14		1		evstwho	r5, 0x0(r3);	1		2	evmhosmiaaw	r5, r18, r10;
15		2	1	addi	r3, r3, 0x4;	2	1	3	evlwhou	r7, 0x10(r4);
16	1	3		addi	r4, r4, 0x4;	3	2		**STALL**	
17		1		cmpwi	r16, 0x0		3		evstwho	r5, 0x4(r3);
18			1	subi	r16, r16, 0x1			1	evmhosmia	r5, r6, r11;
19	1			bne	Loop_begin	1		2	evmhosmiaaw	r5, r7, r9;
20	2			**STALL**		2	1	3	evmergelohi	r17, r6, r7;
21	3			**STALL**		3		1	evmhosmiaaw	r5, r18, r12;
22						1		2	evmhosmiaaw	r5, r17, r10;
23						2	1	3	evlwhou	r8, 0x14(r4);
24				**Non-Optimised Loop**		3	2	1	evaddiw	
25				2 Outputs per loop			3		evstwho	r5, 0x8(r3);
26				6 Stalls per loop				1	addi	r3, r3, 0xC;
27						1			addi	r4, r4, 0xC;
28				**Optimised Loop**			1		cmpwi	r16, 0x0
29				6 Outputs per loop				1	subi	r16, r16, 0x1
30				4 Stalls per loop		1			bne	Loop_begin
31						2			**STALL**	
32						3			**STALL**	

Figure 16. Timing Comparison for Rescheduled Optimized FIR Loop

The timing comparison is then used to form the SPE assembly code below.

Assume:

r3 = Address of first element in output array “y.” Aligned to 64-bit boundary.

r4 = Address of first element in input array “x.” Aligned to 64-bit boundary.

r5 = Address of first element of coefficients “h.” Aligned to 64-bit boundary.

Input data is 16-bit signed integer.

Output data is 32-bit signed integer.

```
order4_FIR:

stwu          r1, -0x18(r1);        #/* Save Non-Volatile r16 */
evstdd        r16, 0x0(r1);         #/* Save Non-Volatile r16 */
evstdd        r17, 0x8(r1);         #/* Save Non-Volatile r16 */
```

```
evstdd        r18, 0x10(r1);       #/* Save Non-Volatile r16 */


addi          r16, r0, 25          #/* Loop to return 150 outputs */
addi          r3, r3, 0x4;         #/* Point yptr to 1st output */
subi          r4, r4, 0x4;         #/* xptr set before 1st input */

evlhhousplat  r9,  0x0(r5);        #/* Load h(0) */
evlhhousplat  r10, 0x2(r5);        #/* Load h(1) */
evlhhousplat  r11, 0x4(r5);        #/* Load h(2) */
evlhhousplat  r12, 0x6(r5);        #/* Load h(3) */

evlwhou       r6, 0x0(r4);         #/* Load x(0/1) */
evlwhou       r7, 0x4(r4);         #/* Load x(2/3) */
evlwhou       r8, 0x8(r4);         #/* Load x(4/5) */

evmergelohi   r18, r6, r7;         #/* Use reg for x(1/2)*/

#------------------------------

Loop_begin:

evmhosmia     r5, r7, r11;         #/* (x(2/3) x h2), store to ACC */
evmhosmiaaw   r5, r8, r9;          #/* (x(4/5) x h0) + ACC */

evmergelohi   r17, r7, r8;         #/* Use reg for x(3/4)*/

evmhosmiaaw   r5, r18, r12;        #/* (x(1/2) x h3) + ACC */
evmhosmiaaw   r5, r17, r10;        #/* (x(3/4) x h1) + ACC */

evlwhou       r6, 0xC(r4);         #/* Load x(6/7) */

evstwho       r5, 0x0(r3);         #/* Store to y(2) & y(3) */

#------------------------------

evmhosmia     r5, r8, r11;         #/* (x(4/5) x h2), store to ACC */
evmhosmiaaw   r5, r6, r9;          #/* (x(6/7) x h0) + ACC */

evmergelohi   r18, r8, r6;         #/* Use reg for x(5/6)*/

evmhosmiaaw   r5, r17, r12;        #/* (x(3/4) x h3) + ACC */
evmhosmiaaw   r5, r18, r10;        #/* (x(5/6) x h1) + ACC */

evlwhou       r7, 0x10(r4);        #/* Load x(8/9) */

evstwho       r5, 0x4(r3);         #/* Store to y(4) & y(5) */
```

```
#------------------------------

evmhosmia      r5, r6, r11;          #/* (x(6/7) x h2), store to ACC */
evmhosmiaaw    r5, r7, r9;           #/* (x(8/9) x h0) + ACC */

evmergelohi    r17, r6, r7;           #/* Use reg for x(7/8)*/

evmhosmiaaw    r5, r18, r12;         #/* (x(5/6) x h3) + ACC */
evmhosmiaaw    r5, r17, r10;         #/* (x(7/8) x h1) + ACC */

evlwhou        r8, 0x14(r4);         #/* Load x(10/11) */
evaddiw        r18, r17, 0;          #/* Copy reg, required for loop */

evstwho        r5, 0x8(r3);          #/* Store to y(6) & y(7) */

#------------------------------

addi           r3, r3, 0xC;          #/* update xptr */
addi           r4, r4, 0xC;          #/* update yptr */
cmpwi          r16, 0x0              #/* Check if loop should end */
subi           r16, r16, 0x1         #/* Decrement loop end */
bne            Loop_begin            #/* Branch to start of loop */

evldd          r16, 0x0(r1);         #/* Restore Non-Volatile r16 */
evldd          r17, 0x8(r1);         #/* Restore Non-Volatile r16 */
evldd          r18, 0x10(r1);        #/* Restore Non-Volatile r16 */
addi           r1, r1, 0x18;         #/* Save stack */

blr                                  #/* Return to function call */
```

5. **Results**

   The optimized SPE function was compared to the non-optimized function, and also to the equivalent function implemented in C, to evaluate the impact on performance.

   The measurements were taken using a MPC5554 with this configuration:

   — Cache enabled for flash and RAM
   — Flash BIUCR optimal Settings: 0x00014BFD
   — System frequency set to 132 MHz
   — Branch target buffer enabled

   The results from these measurements are shown in Table 1.

Table 1. Performance Results of Optimized SPE Assembly, Non-Optimized SPE Assembly and C Implementation of a Fourth-Order FIR

Function	Samples (N)	First Call (Clock Cycles)	Later Calls (Clock Cycles)	Performance Improvement versus C
Optimized SPE	150	1097	1072	92%
Non-Optimized SPE	150	1954	1943	86%
C Function	150	14480	14470	N/A

NOTE

There is a performance difference for the first call of the function as the variables are loaded into cache for the first time. These variables will be stored in the cache; therefore, later function calls do not reload them into cache.

The results in Table 1 clearly illustrate the impact that optimization techniques can have on SPE assembly. The techniques used for this example had 44% better performance than the non-optimized SPE function, and a 92% improvement compared to an equivalent C implementation of the function.

Example B-2. Two-by-Two Matrix Multiplication

---

1. **Determine whether the DSP operation is suited to the SPE.**

   The main operations in matrix multiplications are multiply-and-accumulate.

   There are no restrictions on the data that do not allow 64-bit alignment for the data.

   Therefore this DSP operation is suited to the SPE.

2. **Derive a formula for the DSP operation and graphically map out the flow of data through the SPE function.**

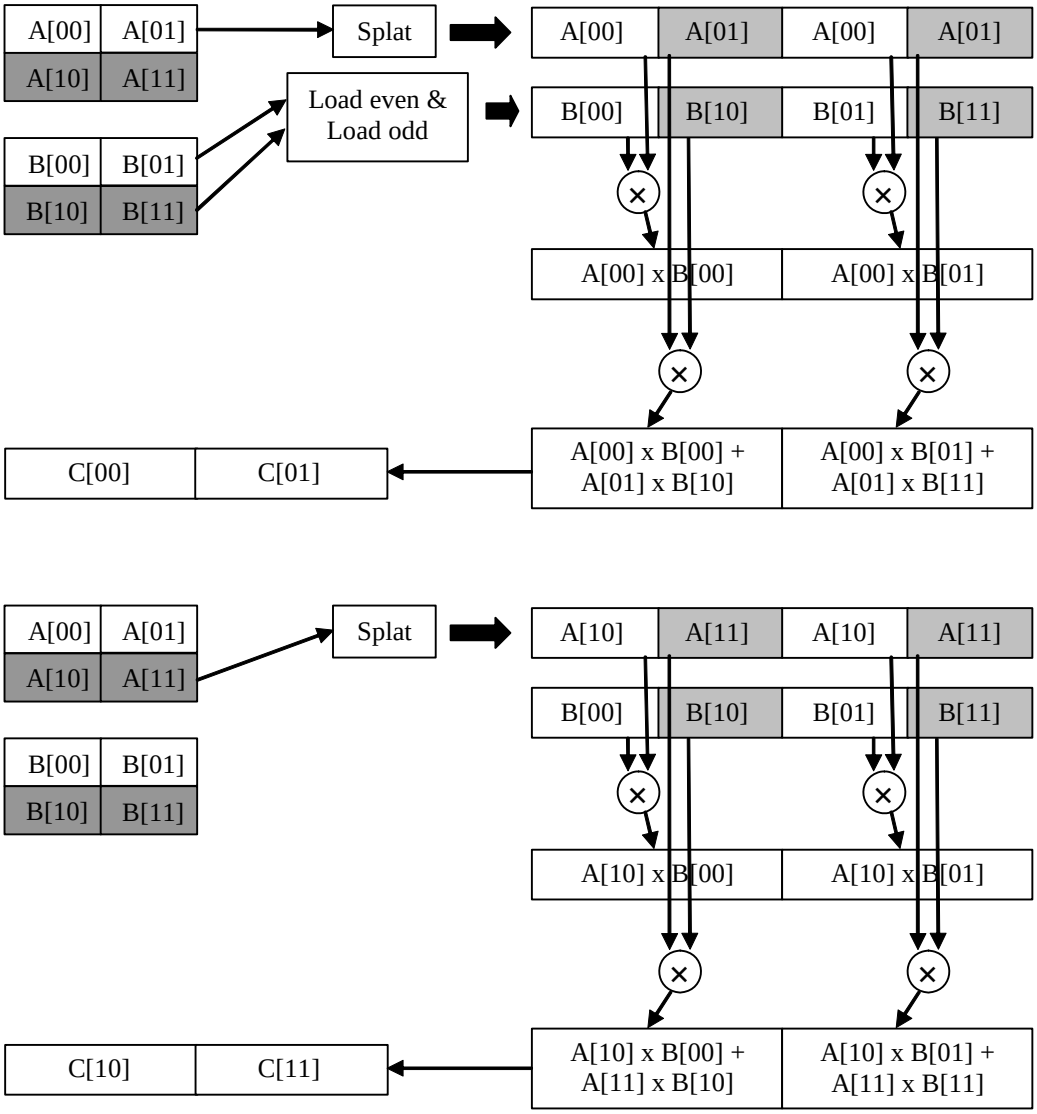
   Taking:

$$\begin{array}{ll} A = [00][01] & B = [00][01] \\ \quad\ [10][11] & \quad\ [10][11] \end{array}$$

$$C = A \times B = \begin{bmatrix} C[00] & C[01] \\ C[10] & C[11] \end{bmatrix}$$

$$= \begin{bmatrix} A[00].B[00] + A[01].B[10] & A[00].B[01] + A[01].B[11] \\ A[10].B[00] + A[11].B[10] & A[10].B[01] + A[11].B[11] \end{bmatrix}$$

This can be represented graphically:



3. Implement the graphical representation as SPE instructions.

Assume:

Require 100 matrix outputs.

r3 = Address of first row in "A." Aligned to 64-bit boundary.

r4 = Address of first row in "B." Aligned to 64-bit boundary.

r5 = Address of first row in "C." Aligned to 64-bit boundary.

Input data is 16-bit signed integer.

Output data is 32-bit signed integer.

Non-Optimized SPE assembly code:

```
Matrix_2_2_Multiply:

addi          r16, r0, 99          #/* Loop to return 100 outputs */

#------------------------------

#/* load r9  = A[0][0] | A[0][1] | A[0][0] | A[0][1]          */
#/* load r10 = B[0][0] | 0x0000  | B[0][1] | 0x0000          */
#/* load r11 = 0x0000  | B[1][0] | 0x0000  | B[1][1]          */
#/* Combine r10 & r11 = B[0][0] | B[1][0] | B[0][1] | B[1][1] */
#/*         A[0][0]*B[0][0]  ||          A[0][0]*B[0][1]      */
#/* ACC_HI + A[0][1]*B[1][0] || ACC_LO + A[0][1]*B[1][1]      */
#/* store to C[0][0] and C[0][1]                              */
#/* load r9  = A[1][0] | A[1][1] | A[1][0] | A[1][1]          */
#/*         A[1][0]*B[0][0]   ||         A[1][0]*B[0][1]      */
#/* ACC_HI + A[1][1]*B[1][0] || ACC_LO + A[1][1]*B[1][1]      */
#/* store to C[1][0] and C[1][1]                              */


Loop_begin:

evlwwsplat        r9, 0(r3);
evlwhe            r10, 0(r4);
evlwhou           r11, 4(r4);
evor              r10, r10, r11;
evmhesmia         r11,r9,r10;
evmhossiaaw       r11,r9,r10;
evstdw            r11,0(r5);
evlwwsplat        r9, 4(r3);
evmhesmia         r12,r9,r10;
evmhossiaaw       r12,r9,r10;
evstdw            r12,8(r5);

#------------------------------

addi          r3, r3, 0x8;         #/* update r3 to next A matrix */
addi          r4, r4, 0x8;         #/* update r4 to next B matrix */
addi          r5, r5, 0x10;        #/* update r5 to next output */
cmpwi         r16, 0x0             #/* Check if loop should end */
sub           r16, r16, 0x1        #/* Decrement loop end */

bne           Loop_begin           #/* Branch to start of loop */

blr                                #/* Return to function call */
```

4. **After the SPE assembly function has been written and tested, apply the techniques for improving efficiency.**
   - Loop is rolled out twice to create a larger loop that outputs two matrices.
   - Instructions are rescheduled to reduce number of stalls in pipeline.
   - Align load and store instructions to 64-bit boundaries to avoid non-aligned performance hits.
   - Rearrange register usage for store instructions to avoid stalls in pipeline.

   Timing comparison between non-optimized matrix multiplication loop and optimized matrix multiplication loop is shown in Figure 17 below.

Cycle	Latency A	Latency B	Latency C	Non-Optimised 2x2 Matrix SPE	Latency A	Latency B	Latency C	Optimised 2x2 Matrix SPE
				Loop_begin:				Loop_begin:
1	1			evlwwsplat r9, 0(r3);	1			evlwwsplat r9, 0(r3);
2		1		evlwhe r10, 0(r4);		1		evor r10, r10, r11;
3		2	1	evlwhou r11, 4(r4);			1	evmhesmia r11,r9,r10;
4		3	2	STALL	1		2	evmhossiaaw r11,r9,r10;
5			(3)	STALL	2	1	3	evlwwsplat r9, 4(r3);
6	1			evor r10, r10, r11;	(3)		1	evmhesmia r12,r9,r10;
7		1		evmhesmia r11,r9,r10;	1		2	evstdw r11,0(r5);
8		2	1	evmhossiaaw r11,r9,r10;	2	1	3	evmhossiaaw r12,r9,r10;
9		3	2	STALL	3	2	1	evlwhe r10, 8(r4);
10			(3)	STALL	1	(3)	2	evlwhou r11, C(r4);
11	1			evstdw r11,0(r5);	2	1	3	evstdw r12,8(r5);
12	2	1		evlwwsplat r9, 4(r3);	(3)	2	1	evlwwsplat r9, 8(r3);
13	3		1	evmhesmia r11,r9,r10;	1	3		evor r10, r10, r11;
14	1		2	evmhossiaaw r11,r9,r10;		1		evmhesmia r11,r9,r10;
15	2		3	STALL		2	1	evmhossiaaw r11,r9,r10;
16	(3)			STALL	1	3	2	evlwwsplat r9, C(r3);
17		1		evstdw r11,8(r5);		1	(3)	evmhesmia r12,r9,r10;
18		2	1	addi r3, r3, 0x10;		2	1	evstdw r11,10(r5);
19	1	3		addi r4, r4, 0x10;	1	3	2	evmhossiaaw r12,r9,r10;
20		1		addi r4, r4, 0x10;	2	1	3	evlwhe r10, 10(r4);
21			1	cmpwi r16, 0x0	(3)	2	1	evlwhou r11, 14(r4);
22	1			subi r16, r16, 0x1	1	3	2	evstdw r12,18(r5);
23		1		bne Loop_begin	2	1	3	addi r3, r3, 0x10;
24		2		STALL	3		1	addi r4, r4, 0x10;
25		3		STALL	1			addi r4, r4, 0x10;
26				**Non-Optimised Loop**		1		cmpwi r16, 0x0
27				1 Outputs per loop			1	subi r16, r16, 0x1
28				8 Stalls per loop	1			bne Loop_begin
29					2			STALL
30				**Optimised Loop**	3			STALL
				2 Outputs per loop				
				2 Stalls per loop				

Figure 17. Timing Comparison for Rescheduled Optimized Matrix Multiplication Loop

The timing comparison is then used to form the SPE assembly code below.

Assume:

Require 100 matrix outputs.

r3 = Address of first row in “A.” Aligned to 64-bit boundary.

r4 = Address of first row in “B.” Aligned to 64-bit boundary.

r5 = Address of first row in “C.” Aligned to 64-bit boundary.

Input data is 16-bit signed integer.

Output data is 32-bit signed integer.

Optimized SPE assembly code:

```
Matrix_2_2_Multiply:

addi          r16, r0, 48          #/* Loop to return 100 outputs */

#/* load r10 = B[0][0] | 0x0000  | B[0][1] | 0x0000               */
#/* load r11 = 0x0000  | B[1][0] | 0x0000  | B[1][1]              */
evlwhe      r10, 0(r4);
evlwhou     r11, 4(r4);

#------------------------------

#/* load r9  = A[0][0] | A[0][1] | A[0][0] | A[0][1]             */
#/* Combine r10 & r11 = B[0][0] | B[1][0] | B[0][1] | B[1][1]    */
#/*          A[0][0]*B[0][0] ||          A[0][0]*B[0][1]         */
#/* ACC_HI + A[0][1]*B[1][0] || ACC_LO + A[0][1]*B[1][1]         */
#/* load r9  = A[1][0] | A[1][1] | A[1][0] | A[1][1]             */
#/*          A[1][0]*B[0][0] ||          A[1][0]*B[0][1]         */
#/* store to C[0][0] and C[0][1]                                 */
#/* ACC_HI + A[1][1]*B[1][0] || ACC_LO + A[1][1]*B[1][1]         */
#/* load r10 = E[0][0] | 0x0000  | E[0][1] | 0x0000             */
#/* load r11 = 0x0000  | E[1][0] | 0x0000  | E[1][1]             */
#/* store to C[1][0] and C[1][1]                                 */

Loop_begin:

evlwwsplat  r9, 0x0(r3);
evor        r10, r10, r11;

evmhesmia   r11,r9,r10;
evmhossiaaw r11,r9,r10;

evlwwsplat  r9, 0x4(r3);

evmhesmia   r12,r9,r10;

evstdw      r11,0x0(r5);
```

```
evmhossiaaw r12,r9,r10;

evlwhe      r10, 0x8(r4);
evlwhou     r11, 0xC(r4);

evstdw      r12,0x8(r5);

#-------------------------------

#/* load r9  = D[0][0] | D[0][1] | D[0][0] | D[0][1]             */
#/* Combine r10 & r11 = E[0][0] | E[1][0] | E[0][1] | E[1][1]    */
#/*          D[0][0]*E[0][0] ||          D[0][0]*E[0][1]         */
#/* ACC_HI + D[0][1]*E[1][0] || ACC_LO + D[0][1]*E[1][1]        */
#/* load r9  = D[1][0] | D[1][1] | D[1][0] | D[1][1]             */
#/*          D[1][0]*E[0][0] ||          D[1][0]*E[0][1]         */
#/* store to F[0][0] and F[0][1]                                */
#/* ACC_HI + D[1][1]*E[1][0] || ACC_LO + D[1][1]*E[1][1]        */
#/* load r10 = H[0][0] | 0x0000  | H[0][1] | 0x0000             */
#/* load r11 = 0x0000  | H[1][0] | 0x0000  | H[1][1]             */
#/* store to F[1][0] and F[1][1]                                */

evlwwsplat  r9, 0x8(r3);
evor        r10, r10, r11;

evmhesmia   r11,r9,r10;
evmhossiaaw r11,r9,r10;

evlwwsplat  r9, 0xC(r3);

evmhesmia   r12,r9,r10;

evstdw      r11,0x10(r5);

evmhossiaaw r12,r9,r10;

evlwhe      r10, 0x10(r4);
evlwhou     r11, 0x14(r4);

evstdw      r12,0x18(r5);

#-------------------------------

addi        r3, r3, 0x10;     #/* update r3 to next A matrix */
addi        r4, r4, 0x10;     #/* update r4 to next B matrix */
addi        r5, r5, 0x20;     #/* update r5 to next output */
cmpwi       r16, 0x0          #/* Check if loop should end */
subi        r16, r16, 0x1     #/* Decrement loop end */
```

```
bne         Loop_begin        #/* Branch to start of loop */

blr                           #/* Return to function call */
```

5. Results

The optimized SPE function was compared to the non-optimized function, and also to the equivalent function implemented in C, to evaluate the impact on performance.

The measurements were taken using an MPC5554 with this configuration:

- Cache enabled for flash and RAM
- Flash BIUCR optimal settings: 0x00014BFD
-
- System frequency set to 132 MHz
- Branch target buffer enabled

The results from these measurements are shown in Table 2.

Table 2. Performance Results of Optimized SPE Assembly, Non-Optimized SPE Assembly and C Implementation of a Fourth-Order FIR

Function	Outputs (N)	First Call (Clock Cycles)	Later Calls (Clock Cycles)	Performance Improvement versus C
Optimized SPE	100	2557	2531	81%
Non-Optimized SPE	100	3340	3329	75%
C Function	100	13715	13661	N/A

In this example, the optimization techniques used gained an extra 23% in performance compared to the non-optimized SPE function, and an 81% improvement compared to an equivalent C implementation of the function.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or +1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3733
Rev. 0
07/2008

freescale™
semiconductor