**Freescale Semiconductor**
Application Note

AN3288/D
Rev. 0, 7/2006

# Enhanced Table Lookup Performance

## Using the MPC5500 Signal Processing Engine (SPE)

by Bill Terry - MCU Division - Applications Engineering
John West - MCU Division - Systems Engineering

**Table of Contents**

1 Introduction........1
2 Function Overview........2
3 Table Format........2
3.1 General Requirements........3
3.2 Table Data for C Function........3
3.3 Table Data for Optimized assembly Function....3
4 Lookup Algorithms........4
4.1 C-Based Algorithm........4
4.2 SPE Optimized Algorithm........4
4.2.1 Flow Diagram........5
4.2.2 Code Analysis........6
5 Results........12
6 Summary........12
A.1 var_table_lookup_asm( )........13
A.2 var_table_lookup_c( )........15
A.3 Tables........16

## 1 Introduction

Lookup tables are often used to store data in a multi-dimensional array format. These tables provide a means to capture the dynamic behavior of a system and allow it to be expressed as a predetermined set of data points (called breakpoints[1]). The breakpoints are used to store the input-output mapping of the system. That is, for each input value, there are one or more associated output values, depending on the number of dimensions in the array.

In many automotive applications, table lookups comprise a large percentage of the overall execution time of the application, typically 10-20%. The tables used for these functions are two dimensional and have relatively few breakpoints. Traditional search methods such as binary search or hash functions do not work well on small tables. Usually a simple iterative search and compare loop is used to find a corresponding breakpoint for a given input value.

[1] These breakpoints should not be confused with the term breakpoint as used in a software debugger.

This document contains information on a new product. Specifications and information herein are subject to change without notice.

This application note describes the operation and performance results for an optimized table lookup function and compares execution time with a more typical search function. The optimized function is written in assembly using the PowerPC 32-bit Book E instruction set with the Signal Processing Engine (SPE) and its related instructions.

Example code is included for both the typical and optimized lookup functions and the optimized function example may be built into an application with minimal modification.

# 2 Function Overview

The lookup table is a set of values or breakpoints consisting of an input or x value and a corresponding output or y value. The relationship of each discrete input to output value pair usually represents a continuous function where y is some function of x, or f(x) = y.

To generate a return value y, the table lookup function takes the input and attempts to find a matching x value:

- If the input value is less than the first x value or greater than the last x value, the function uses the appropriate end point y value as the output.
- If a matching x value is found, the corresponding y value is returned.
- If neither of the first two conditions are met, the function finds the two nearest x values above and below the input value, and determines the slope of the function between those two points (i.e. is $Y_{LOW}$ greater than or less than $Y_{HIGH}$?) Then, Equation 1 is used to perform a linear interpolation and approximate a y value.

**Eqn. 1**

$$Y = Y_{LOW} + \frac{((input - X_{LOW}) \cdot (Y_{HIGH} \pm Y_{LOW}))}{(X_{HIGH} - X_{LOW})}$$

| | |
|---|---|
| $Y$ | – Interpolated return value. |
| $input$ | – input x value. |
| $X_{LOW}$ | – The nearest x value in the table that is less than the input value. |
| $Y_{LOW}$ | – The y value associated with $X_{LOW}$. |
| $X_{HIGH}$ | – The nearest x value in the table that is greater than the input value. |
| $Y_{HIGH}$ | – The y value associated with $X_{HIGH}$. |

The functions are called with a pointer to the appropriate table, and an input x value and return a 16-bit value.

# 3 Table Format

The table may have a variable number of entries, but the total number of entries must be a multiple of four. A table entry is defined as one x y pair.

As discussed in Section 1, there are two example lookup functions, a typical iterative search function written in C, and an optimized function written in assembly using SPE instructions. The following sections define the required formats and contents of the table data for each of the function types.

**NOTE**

While the table structure and format differs for the C and assembly versions of the lookup table, the size of the tables and the data values are identical to insure valid performance comparisons.

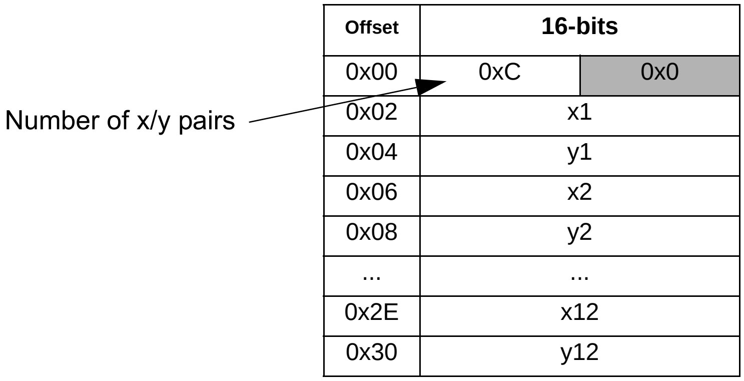
## 3.1 General Requirements

The following are requirements for the lookup data tables used in this application note:

- All x and y table entries should be unsigned, fixed point numbers.
- The lookup table should have each x entry in order of increasing magnitude. That is, x1 must be less than x2, x2 must be less than x3, etc.
- Every x value must have a corresponding y value.
- Each table must be a global data structure.

## 3.2 Table Data for C Function

The table data for the C lookup function is shown in Figure 1. The first byte (lowest memory address of the table) contains the number of table entries. Following that byte, there is a one byte pad to align the first table entry pair on the appropriate memory boundary. Following that are the x y pairs of the table.



Number of x/y pairs

| Offset | 16-bits | |
|---|---|---|
| 0x00 | 0xC | 0x0 |
| 0x02 | x1 | |
| 0x04 | y1 | |
| 0x06 | x2 | |
| 0x08 | y2 | |
| ... | ... | |
| 0x2E | x12 | |
| 0x30 | y12 | |

**Figure 1. C Function Table Memory Layout**

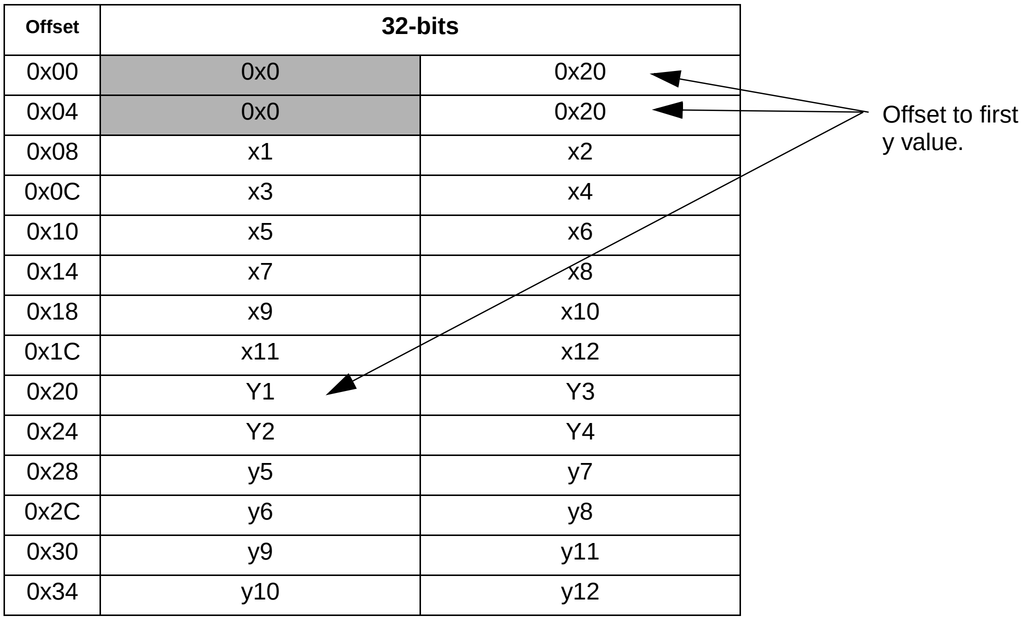
## 3.3 Table Data for Optimized assembly Function

In order to take advantage of the SPE instruction set used in the assembly functions, the table data must be organized in memory to accommodate the use of 64-bit operands found in many SPE instructions. The memory layout for the SPE assembly function lookup table is shown in Figure 2.

In the SPE optimized version of the data table the following rules apply:

- Each table entry must be two bytes in length.
- Each data vector can hold any number of entries; however, in order to avoid an alignment exception error, the total number of table entries must be a multiple of 4.

- Dummy values can be stored in the table if the data has a number of breakpoints that is not a multiple of four.
- The x values must be stored in ascending order.
- The y values must be interleaved as shown in Figure 2.
- The second and fourth 16-bit values of the table must contain the offset to the first y value from the base address of the table.



| Offset | 32-bits | |
|---|---|---|
| 0x00 | 0x0 | 0x20 |
| 0x04 | 0x0 | 0x20 |
| 0x08 | x1 | x2 |
| 0x0C | x3 | x4 |
| 0x10 | x5 | x6 |
| 0x14 | x7 | x8 |
| 0x18 | x9 | x10 |
| 0x1C | x11 | x12 |
| 0x20 | Y1 | Y3 |
| 0x24 | Y2 | Y4 |
| 0x28 | y5 | y7 |
| 0x2C | y6 | y8 |
| 0x30 | y9 | y11 |
| 0x34 | y10 | y12 |

Offset to first
y value.

**Figure 2. Assembly Function Table Memory Layout**

# 4 Lookup Algorithms

This section provides a comparison of the actual algorithms used in the lookup routines.

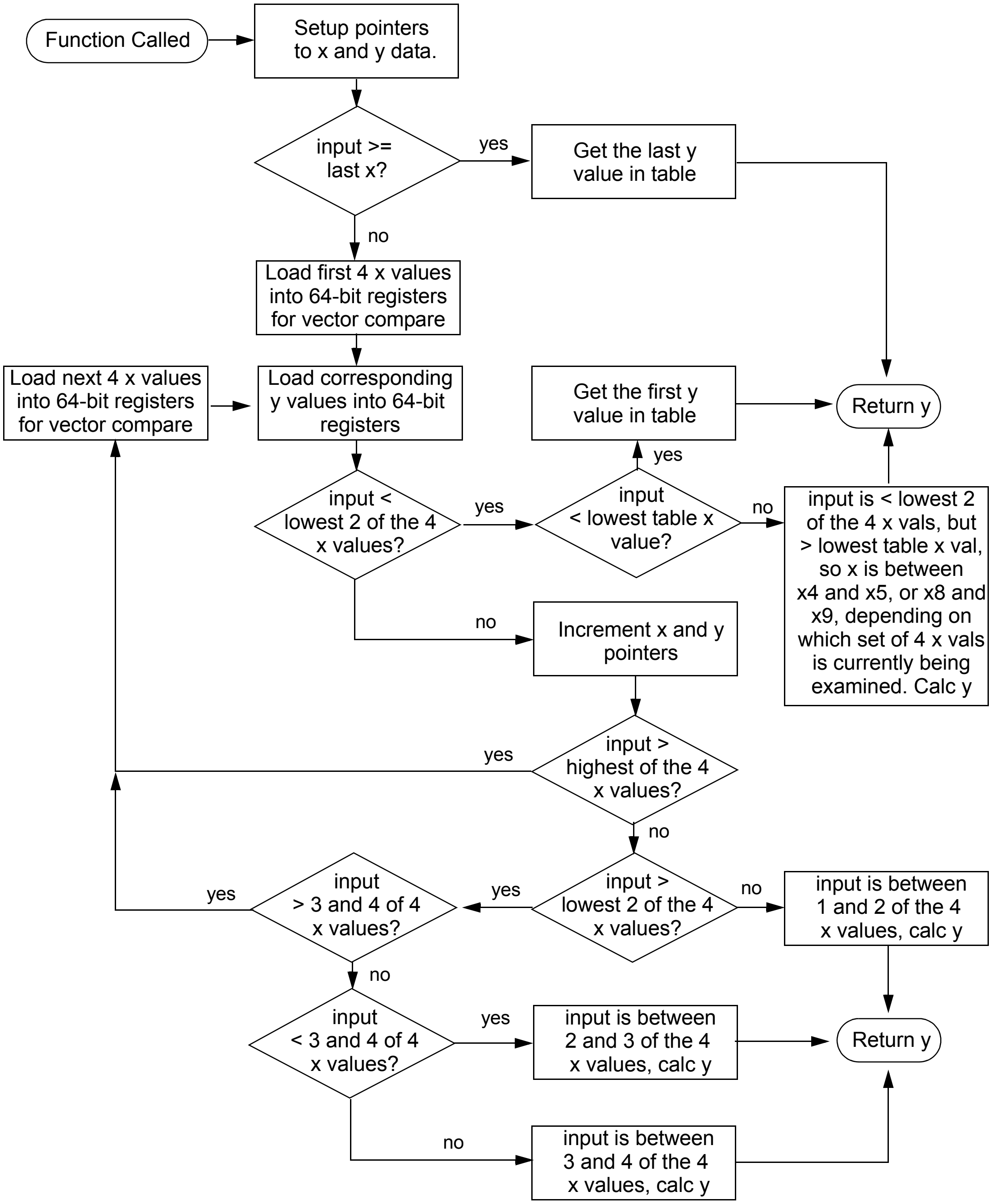
## 4.1 C-Based Algorithm

The non-optimized C code lookup functions do not use SPE instructions to perform table lookups, but rather a traditional iterative search algorithm. This type of algorithm is essentially a simple increment and compare loop, where the farther into a table the match is located, the longer the execution time that is required. This is a well known search algorithm and the details are not discussed in this application note. A code listing for the C code can be found in Section A.2.

## 4.2 SPE Optimized Algorithm

The SPE optimized lookup algorithm is coded in assembly, and due in part to the unique way that the SPE instructions are implemented, is not easily understood by a simple inspection of the code listing. The inherent advantage of the SPE optimized function is that it utilizes 64-bit registers to perform tests or compares on two values simultaneously. The following section and diagram details the steps of the SPE enhanced algorithm. Each related code segment is followed by an explanation of the purpose of the operation(s). The code listing includes line numbers to simplify referencing. Referencing the table data layout (see Section 3.3) will aid in understanding the following algorithm. A complete listing of the SPE based code can be found in Section A.1.

## 4.2.1 Flow Diagram


Function Called
Setup pointers to x and y data.
input >= last x?
yes
Get the last y value in table
no
Load first 4 x values into 64-bit registers for vector compare
Load next 4 x values into 64-bit registers for vector compare
Load corresponding y values into 64-bit registers
Get the first y value in table
Return y
yes
input < lowest 2 of the 4 x values?
yes
input < lowest table x value?
no
input is < lowest 2 of the 4 x vals, but > lowest table x val, so x is between x4 and x5, or x8 and x9, depending on which set of 4 x vals is currently being examined. Calc y
no
Increment x and y pointers
yes
input > highest of the 4 x values?
no
yes
input > 3 and 4 of 4 x values?
yes
input > lowest 2 of the 4 x values?
no
input is between 1 and 2 of the 4 x values, calc y
no
input < 3 and 4 of 4 x values?
yes
input is between 2 and 3 of the 4 x values, calc y
Return y
no
input is between 3 and 4 of the 4 x values, calc y

## 4.2.2 Code Analysis

This section provides a line by line analysis of the SPE based table lookup function.

```
1   unsigned short var_table_lookup_asm( unsigned short *DataPtr, unsigned short Input)
```

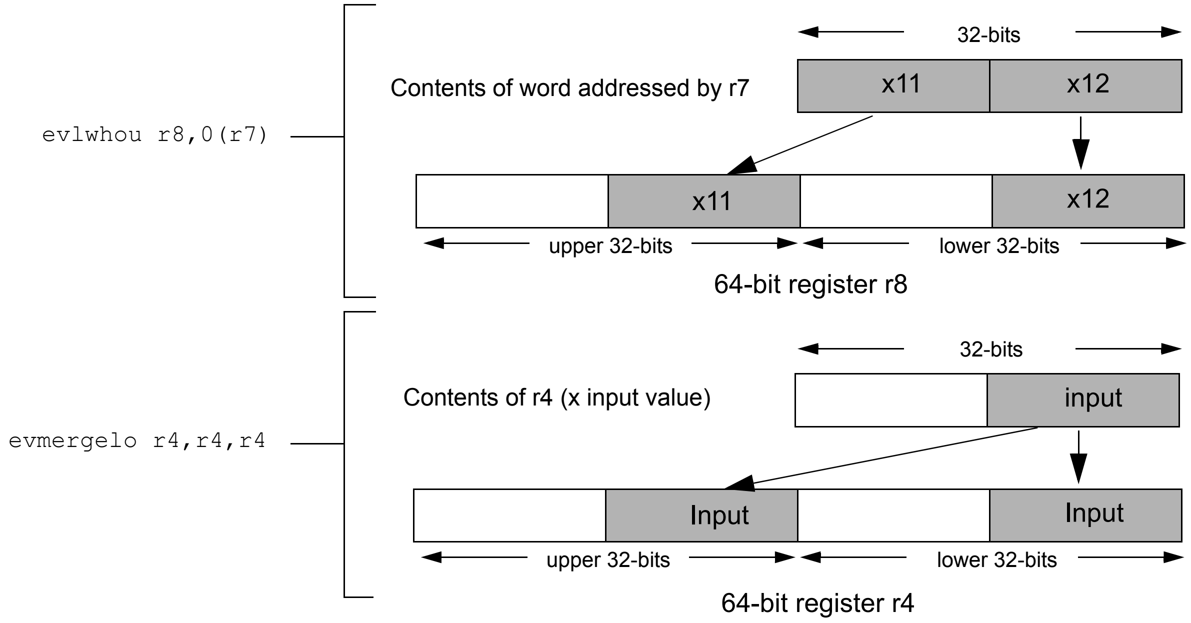
Line 1 is the actual function declaration. This function is called with two args, a pointer to the relevant data table (`*DataPtr`), and the input x value (`Input`). Notice that it returns an unsigned 16-bit value.

```
{
/*Set up X and Y data pointers*/
2   asm("lhz         r12,8(r3)      ");  // initialize r12 to first X val
3   asm("lwz         r5,0(r3)       ");  // Put the y-offset (from table entry) value in r5
4   asm("addi        r6,r5,-8       ");  // Put the number of data points times 2 in r6
5   asm("add         r5,r3,r5       ");  // Set r5 to the address of the first Y value
                                         // r5 = r3 (table address) + r5 (y-offset)
6   asm("addi        r7,r5,-4       ");  // r7 = address of word holding last two x values
```

Lines 2 through 6 perform several operations to setup up the required pointers into the data structure as indicated by the code comments. The PowerPC EABI dictates that when this function is called, r3 holds the address of the data structure, and r4 holds the input value.

```
7   asm("evlwhou     r8,0 (r7)      ");  // 64-bit r8 now holds next to last x and last x
                                         // values
8   asm("evmergelo   r4,r4,r4       ");  // move r4[0:31] to r4[32:63]
```

Lines 7 and 8 perform SPE operations to setup up for a compare as shown below.



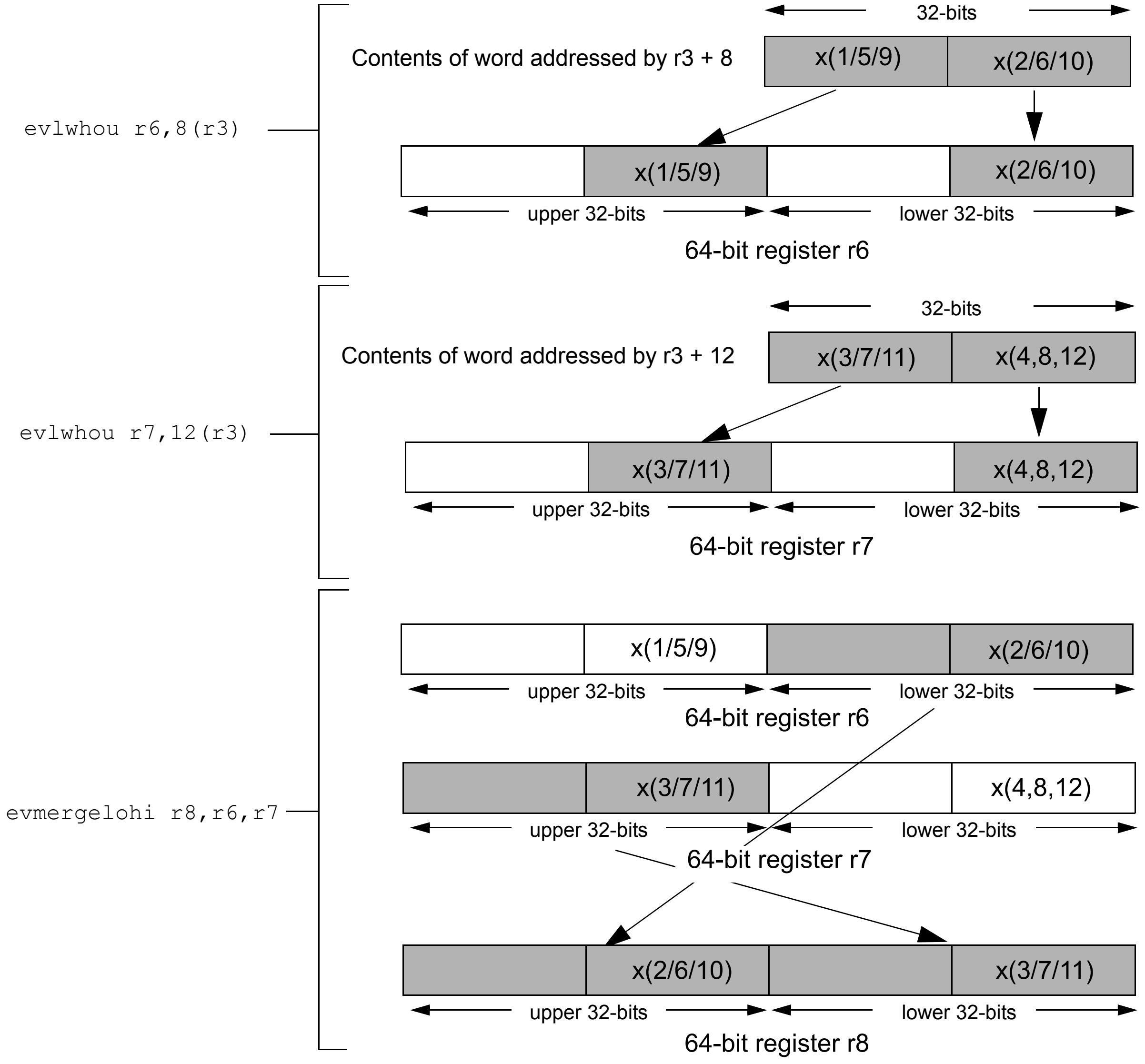
```
9   asm("cmp         1,r8,r4        ");  // bounds check (input >= last x value?)
10  asm("bc          4,4,toohigh    ");  // if > or = last x, then input is out of bounds
```

Line 9 compares the input value in the lower 32-bits of r4 with the maximum x value which is in the lower 32-bits of r8. If the input is greater than or equal to the last x value, the code branches to the label `toohigh`.

```
/*   Pre-load 4 x-values to compare with input. Note that the first time through this loop
```

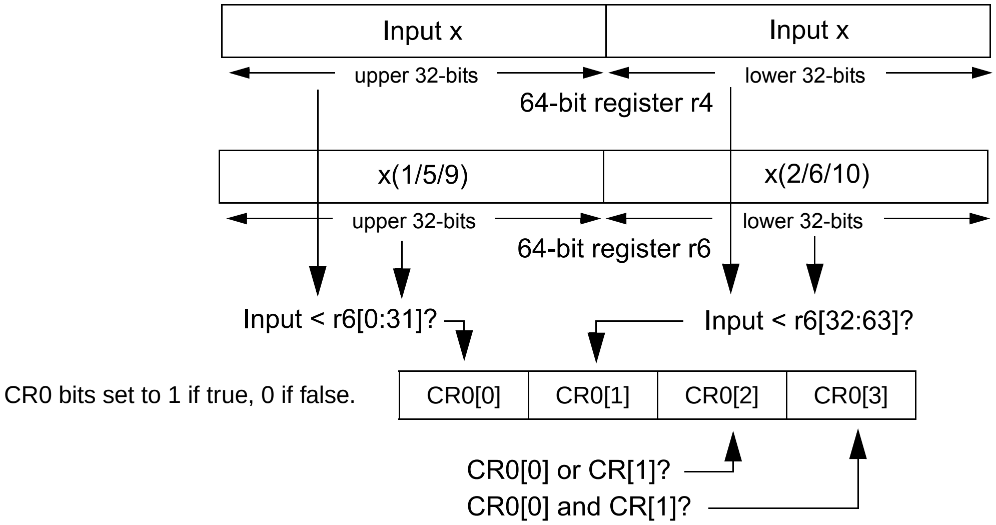
```
      64-bit registers r6 and r7 hold x1, x2, x3, and x4. The second time through this loop, if
      necessary, they will hold x5, x6, x7, and x8, and the last time through this loop, if
      necessary, they will hold x9, x10, x11 and x12. */
11  asm("Loop:");
12  asm("evlwhou     r6,8 (r3)    ");    // r6[0:31]=x(1/5/9), r6[32:63]=x(2/6/10)
13  asm("evlwhou     r7,12 (r3)   ");    // r7[0:31]=x(3/7/11), r7[32:63]=x(4/8/12)
14  asm("cmp         1,r7,r4      ");    // Early Check input > r7[32:63]? result in CR1
15  asm("evmergelohi r8,r6,r7      ");    // r8[0:31]=x2/x6/x10 ,r8[32:63]=x3/x7/x11
```

Depending on which iteration through the loop this is, lines 12 through 15 load the x(1/5/9), x(2/6/10), x(3/7/11) and x(4/8/12) values into vectors as shown below, in preparation for a vector compare with the input value. Note that after these operations a vector compare can be made on x1 and x2 (r6), x2 and x3 (r8), or x3 and x4 (r7). The x(loop 1/loop 2/loop 3) designation indicates which value x takes depending on the loop iteration. Similarly, a y(loop 1/loop 2/loop 3) designation indicates the y value depending on the loop iteration. As an example, the second time through the loop, the x values will be x5, x6, x7 and x8.



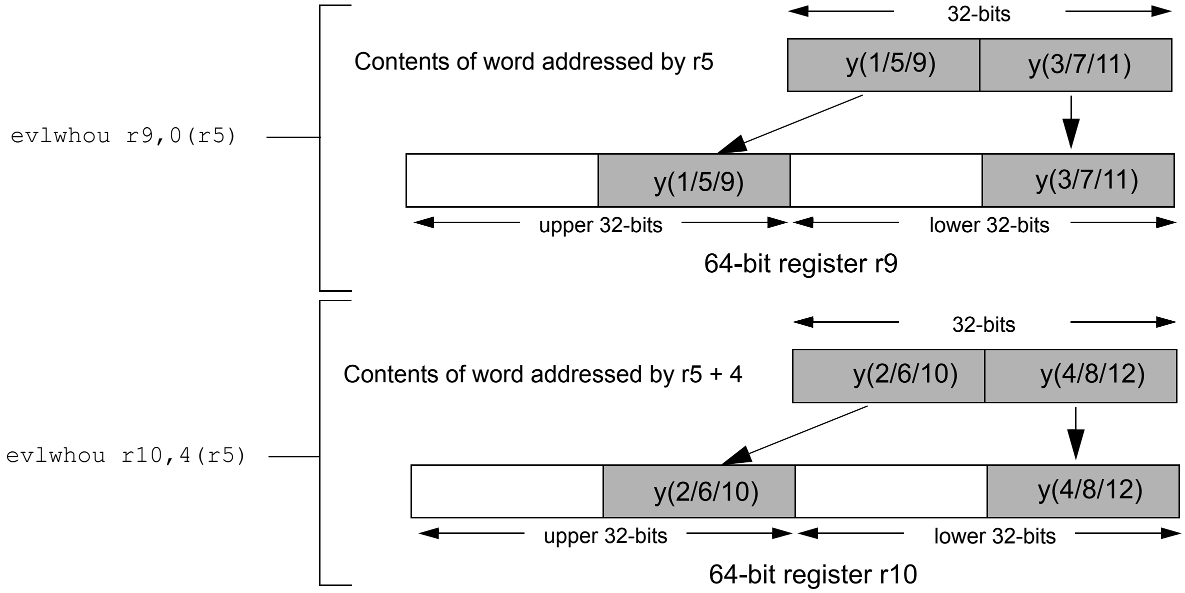
```
/* Check the input against current x value pair in r6 */
16  asm("evcmplts    0,r4,r6   ");       // compare input to current pair of x values in r6,
                                         // result in CR0
```

Line 16 performs a vector compare of r4 to r6, which now holds x(1/5/9) and x(2/6/10). The results of this compare are captured in the CR0 register as shown in the following diagram. Note that this single vector compare instruction provides information that can be used for several conditional branches later in the algorithm.



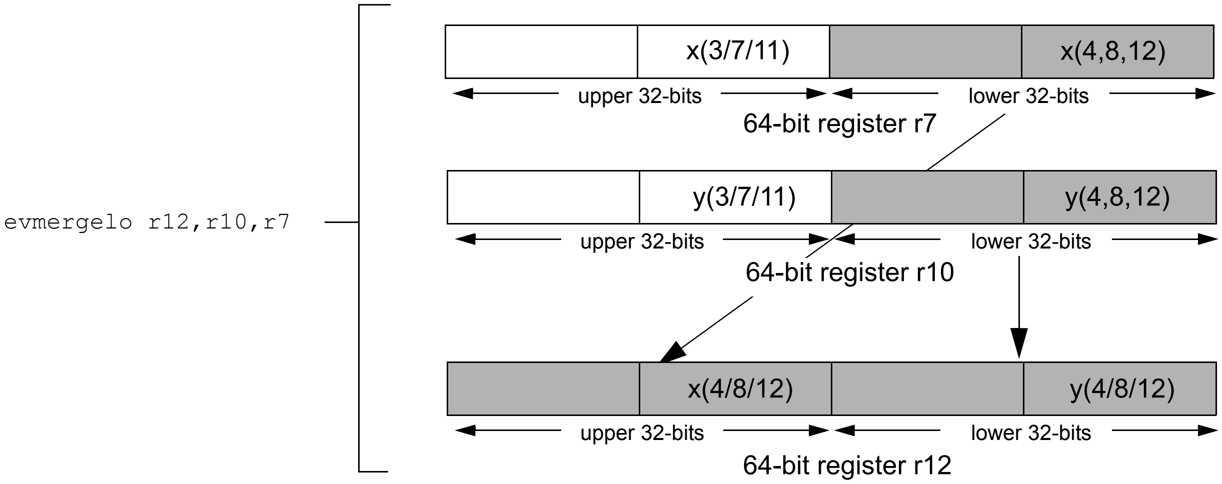
```
/* load the related y values in r9 and r10 */
17  asm("evlwhou     r9,0 (r5)    ");    // r9[0:31]=y1/y5/y9 ,r9[32:63]=y3/y7/y11
18  asm("evlwhou     r10,4(r5)    ");    // r10[0:31]=y2/y6/10 ,r10[32:63]=y4/y8/y12
19  asm("bc          12,3,BackGrd2");    // is input < both r6 x values?, if so branch
```

Lines 17 and 18 move y(1/5/9), y(3/7/11), and y(2/6/10), y(4/8/12) into r9 and r10 respectively (r5 holds the address of the first y value) as shown in the following diagram. These values are needed later to calculate a return value so are loaded now. The results of the input value compare in line 16 are still in CR0 at this point, and now a conditional branch is made in line 19 based on the value in CR0[3]. If the input value is less than both x(1/5/9) and x(2/6/10), the branch is taken, if not execution continues inline.



```
20 asm("evmergelo   r12,r10,r7     ");  // else save r7[32:63] and  r10[32:63] in case input
                                        // is between r7[32:63] and the next greater x value.
21 asm("addi        r3,r3,8        ");  // increment x pointer in case input > r7[32:63]
22 asm("addi        r5,r5,8        ");  // increment y pointer in case input > r7[32:63]
23 asm("bc          12,4,Loop      ");  // if (input > r7[32:63]) get next four x values to
                                        // compare
```

As shown in the following diagram, line 20 saves x(4/8/12) and y(4/8/12) in case the input value is between one of the three ‘groups’ of four x values being examined in 64-bit registers r6 and r7, i.e. x input is greater than x4, but less than x5 or greater than x8, but less than x9. Lines 21 and 22 increment the x and y pointers in case the input x value is greater than x(4/8). Line 23 checks to see if the input is greater than all four x values and if so, branches to `Loop` to get the next four x values in the table.



```
24  asm("evsubfw     r11,r9,r10   ");     // else r11[0:31]=(y2/y6/y10 - y1/y5/y9),
                                          // and r11[32:63]=(y4/y8/y12 - y3/y7/y11),
                                          // this is (y high-y low)
25  asm("bc          4,1, comp    ");     // if(input > x1/x5/x9 & x2/x6/x10) branch
```

Line 24 calculates the difference between y high, and y low for each of the four y values in 64-bit registers r9 and r10. These values are used later during calculation of the return value. Line 25 does a compare of x(1/5/9) and x(2/6/10), and if the x input value is greater than both a branch is taken to `comp`.

```
/*   If no branch was taken, input is known to be between x(1/5/9) and x(2/6/10) so a return
     value is interpolated. */
26  asm("evsubfw     r4,r6,r4         ");  // r4 = r4-r6 (input - xlow)
27  asm("evmergehi   r4,r0,r4         ");  // r4[0:31]-->r4[32:63]
28  asm("evmergehi   r11,r0,r11       ");  // r11[0:31]-->r11[32:63] (y high-y low)
29  asm("mullw       r4,r4,r11        ");  // (input-Xlow) * (Yhigh-Ylow)
30  asm("evsubfw     r8,r6,r8         ");  // (xhigh-Xlow),  r8[0:31]=(X2-X1)
31  asm("evmergehi   r8,r0,r8         ");  // r8[0:31]-->r8[32:63]
32  asm("divw        r4,r4,r8         ");  // 32 bit quotient
33  asm("evmergehi   r9,r0,r9         ");  // r9[0:31]-->r9[32:63]
34  asm("add         r3,r4,r9         ");  // return answer in r3
35  asm("b           return_y         ");
```

If the branch in line 25 was not taken, the input x value is know to be between x(1/5/9) and x(2/6/10) so beginning at line 26, a return value is interpolated based on the formula in Equation 1 on Page 2 and returned to the calling function.

```
/* Checking if input is between x3/x7/x11 and x4/x8/x12 */
36  asm("comp:");
37  asm("evcmplts    0,r4,r7          ");  // input< r7 (input< x3/x7/x11 or input< x4/x8/x12),
                                           // result in CR0
38  asm("bc          4,1, Loop        ");  // if(input > x3/x7/x11 & x4/x8/x12) get more x values
39  asm("bc          12,3,BackGrd1    ");  // if(input < x3/x7/x11 & x4/x8/x12) branch.
```

If the code reaches the `comp` label at line 36, the x input value is now checked to see if it is less than x(3/7/11) and/or x(4/8/12). The results of this compare are in CR0. If the input is greater than both these x values, a branch is taken in line 38 to `Loop` to get the next set of four x values. If the input x value is less than both of these x values it must be between x2 and x3 so execution branches at line 39 to `BackGrd1` to interpolate a y value.

```
/*At this point X input is known to be between x3/x7/x11 and x4/x8/x12 so a return
  value is interpolated. */
40  asm("subf        r4,r8,r4         "); // r4 = r4 -r8 (input-xlow)
41  asm("mullw       r4,r4,r11        "); // (input-Xlow)*(Yhigh-Ylow)
42  asm("subf        r8,r8,r7         "); // (Xhigh-Xlow),  r8[32:63]= x(4/8/12) - x(3/7/11)
43  asm("divw        r4,r4,r8         "); // 32 bit quotient
44  asm("add         r3,r4,r9         "); // return answer in r3
45  asm("b           return_y         ");
```

If neither branch in lines 38 and 39 were taken, the input x value is between x(3/5/7) and x(4/8/12). The code starting at line 40 interpolates a y value and returns it to the calling function.

```
/*At this point X input is known to be between x2/x6/x10 & x3/x7/x11 so a return
  value is interpolated. */
46  asm("BackGrd1:");
47  asm("evmergehi   r10,r0,r10       "); // r10[0:31]-->r10[32:63]= Y2
48  asm("subf        r4,r6,r4         "); // compute (input-Xlow)
49  asm("subf        r9,r10,r9        "); // (Yhigh-Ylow) or (Y3-Y2)
50  asm("mullw       r4,r9,r4         "); // (input-Xlow)*(Yhigh-Ylow)
```

```
51 asm("subf        r8,r6,r8       "); // (Xhigh-Xlow), r8[32:63]=(Y3-Y2)
52 asm("divw        r4,r4,r8       "); // 32 bit quotient
53 asm("add         r3,r4,r10      "); // return answer in r3
54 asm("b           return_y       ");
```

If the branch in line 39 was taken, the input x value is between x(2/6/10) and x(3/7/11). The code starting at line 47 interpolates a y value and returns it to the calling function.

```
/* Input is less than the current x values in r6, so it's either too low,
   or between x4 and x5, or between x8 and x9.*/
55 asm("BackGrd2:");
56 asm("evmergehilo r4,r9,r4       "); // r9[0:31]=Yhigh r4[32:63]=input
```

If the branch in line 19 was taken, the input x value is less than x(1/5/9) and x(2/6/10). If the input x value is not lower than the first x value, the code starting at line 57 interpolates a y value and returns it to the calling function. Line 63 does a compare of the input x value and the lowest possible x table value (x1). If it is less than x1, the code branches to `toolow`.

```
57 asm("evsubfw     r5,r12,r4      "); // else, compute (Yhigh-Ylow) and  (input-Xlow)
58 asm("evmergehi   r10,r0,r5      "); // r4[0:31]-->r10[32:63]
59 asm("mullw       r10,r5,r10     "); // (Yhigh-Ylow)*(input-Xlow)
60 asm("evmergehi   r6,r0,r6       "); //  r6[0:31]-->r6[32:63]
61 asm("subf        r8,r12,r6      "); // (Xhigh-Xlow), r8[32:63]=(X5-X4)
62 asm("divw        r10,r10,r8     "); // 32 bit quotient
63 asm("cmp         1,r4,r12       "); // Bounds Check is input < first x value
64 asm("bc          12,4,toolow    "); // if true, then input out of bounds
65 asm("evmergehi   r3,r0,r12      "); // r12[0:31]-->r12[32:63]
66 asm("add         r3,r10,r3      "); // return answer in r3
67 asm("b           return_y       ");

/* OUT OF BOUNDS */
68 asm("toolow:");
69 asm("evmergehi   r3,r0,r4       "); // merge contents of  r4[0:31]-->r10[32:63]
70 asm("b           return_y       ");            ");
```

If the code branched to `toolow` from line 64, the code beginning at line 69 returns the lowest y value, or y1.

```
71 asm("toohigh:");
72 asm("evlwhoux    r3,r6,r7       "); // r6 = last y value

73 asm("return_y:");
74 return;
}
```

If the code branched to `toohigh` from line 10, the code beginning at line 72 returns the highest y value, or y12.

# 5 Results

Using a test table with a range of input x values the following results were obtained. On average the SPE based lookup code was 41% faster.

**Table 1. Test Results**

| | C code | | SPE code | | |
|---|---|---|---|---|---|
| Input x | y returned | time[1] | y returned | time[1] | SPE Execution vs C Code |
| 0x210A | 0xBCCC | 175 | 0xBCCC | 107 | - 39% |
| 0x3F0A | 0x9EC6 | 201 | 0x9EC6 | 111 | - 45% |
| 0x4325 | 0x9AAA | 201 | 0x9AAA | 111 | - 45% |
| 0x5A35 | 0x8397 | 226 | 0x8397 | 132 | - 42% |
| 0x6428 | 0x799B | 227 | 0x799B | 133 | - 41% |
| 0x7555 | 0x686D | 240 | 0x686D | 145 | - 40% |
| 0x8765 | 0x5658 | 253 | 0x5658 | 137 | - 46% |
| 0x9875 | 0x4543 | 266 | 0x4543 | 160 | - 40% |
| 0xA985 | 0x3438 | 279 | 0x3438 | 159 | - 43% |
| 0xBA95 | 0x2308 | 292 | 0x2308 | 171 | -41% |
| 0xCB05 | 0x128B | 305 | 0x128B | 163 | -47% |
| 0xDC78 | 0x1116 | 90 | 0x1116 | 67 | -26% |
| | | | | Average | -41% |

NOTES:

[1] Timebase counts

# 6 Summary

As shown in the preceding section, the SPE enhanced routine is significantly faster than the non-optimized C function. The performance improvement comes from the ability to perform parallel (vector) operations on multiple sets of data, and by minimizing the number of memory accesses needed per operation. Also, comparing multiple sets of data enables faster search processing. Other optimization techniques were also used, such as loop unrolling and scheduling instructions out of order to minimize pipe line stalls.

# A Source Code Listing

## A.1 var_table_lookup_asm( )

```
// Copyright (c) 2006, Freescale.
//
// -----------------------------------------------------------------------------------------
// RELEASE HISTORY
// VERSION  DATE           AUTHOR             DESCRIPTION
// 1.0   2006-7-6        Bill Terry          Initial release
// ------------------------------------------------------------------------------------------
// PURPOSE: This function processes two dimensional table lookups on 16-bit table entries.
//
// NOTE:    Maximum number of table entries must be a multiple of 4.
//
//------------------------------------------------------------------------------------------

/*******************************************************************************************
 * r3 [32:63] = base address of table
 * r4 [32:63] = input value

********************************************************************************************/

unsigned short var_table_lookup_asm( unsigned short  *DataPtr, unsigned short Input)
{

 /* Set up x and y data pointers */
     asm("lhz       r12,8(r3)      "); // initialize r12 to first X val
     asm("lwz       r5,0(r3)       "); // load offset to y values in r5
     asm("addi      r6,r5,-8       "); // Put the number of data points x 2 in r6
     asm("add       r5,r3,r5       "); // Set r5 to the address of the first Y value
     asm("addi      r7,r5,-4       "); // decrement y pointer to last x value (Bnds Checking)
     asm("evlwhou   r8,0 (r7)      "); // r8 = last x value  (Bnds Checking)
     asm("evmergelo r4,r4,r4       "); // r4[32:63] --> r4[0:31]
     asm("cmp       1,r8,r4        "); // Bounds check (input > last x value?)
     asm("bc        4,4,toohigh    "); // if > or = last x, then input is out of bounds (CR

/* Pre-load 4 x-values to compare with input. Note that the first time through this loop
   64-bit registers r6 and r7 hold x1, x2, x3, and x4. The second time through this loop
   (if necessary) they will hold x5, x6, x7, and x8, and the last time through this loop
   (if necessary) x9, x10, x11 and x12. */
     asm("Loop:");
     asm("evlwhou   r6,8 (r3)      "); // r6[0:31]=x1/x5/x9, r6[32:63]=x2/x6/x10
     asm("evlwhou   r7,12 (r3)     "); // r7[0:31]=x3/x7/x11, r7[32:63]=x4/x8/x12
     asm("cmp       1,r7,r4        "); // Early Check input > r7[32:63]? result in CR1
     asm("evmergelohir8,r6,r7      "); // r8[0:31]=x2/x6/x10 ,r8[32:63]=x3/x7/x11

/* Check the input against current x value pair in r6 */
     asm("evcmplts  0,r4,r6        "); // compare input to current pair of x values in r6,
                                        // result in CR0
/* load the related y values in r9 and r10 */
     asm("evlwhou   r9,0 (r5)      "); // r9[0:31]=y1/y5/y9 ,r9[32:63]=y3/y7/y11
     asm("evlwhou   r10,4(r5)      "); // r10[0:31]=y2/y6/10 ,r10[32:63]=y4/y8/y12
     asm("bc        12,3,BackGrd2  "); // is input < both r6 x values?, if so branch
     asm("evmergelo r12,r10,r7     "); // else save r7[32:63] and  r10[32:63] in case input
                                        // is between r7[32:63] and the next greater x value.
     asm("addi      r3,r3, 8       "); // increment x pointer in case input > r7[32:63]
     asm("addi      r5,r5, 8       "); // increment y pointer in case input > r7[32:63]
     asm("bc        12,4,Loop      "); // if (input > r7[32:63]) get more values to compare
     asm("evsubfw   r11,r9,r10     "); // else r11[0:31]=(y2/y6/y10 - y1/y5/y9),
                                        // and r11[32:63]=(y4/y8/y12 - y3/y7/y11), this is (y
                                        // high-y low)
     asm("bc         4,1, comp     "); // if(input > x1/x5/x9 & x2/x6/x10) branch

/* At this point X input is known to be between r6[0:31] and r6[32:63] so a return
   value is interpolated. */
     asm("evsubfw   r4,r6,r4       "); // r4 = r4-r6 (input - xlow)
     asm("evmergehi r4,r0,r4       "); // r4[0:31]-->r4[32:63]
     asm("evmergehi r11,r0,r11     "); // r11[0:31]-->r11[32:63] (y high-y low)
     asm("mullw     r4,r4,r11      "); // (input-Xlow) * (Yhigh-Ylow)
     asm("evsubfw   r8,r6,r8       "); // (xhigh-Xlow),  r8[0:31]=(X2-X1)
     asm("evmergehi r8,r0,r8       "); // r8[0:31]-->r8[32:63]
     asm("divw      r4,r4,r8       "); // 32 bit quotient
     asm("evmergehi r9,r0,r9       "); // r9[0:31]-->r9[32:63]
     asm("add       r3,r4,r9       "); // return answer in r3
     asm("b         return_y");

/* Checking if input is between x3/x7/x11 and x4/x8/x12 */
     asm("comp:");
     asm("evcmplts  0,r4,r7        "); // input< r7 (input< x3/x7/x11 or input < x4/x8/x12),
                                        // result in CR0
     asm("bc        4,1, Loop      "); // if (input > x3/x7/x11 & x4/x8/x12) get more x values
     asm("bc        12,3,BackGrd1  "); // if (input < x3/x7/x11 & x4/x8/x12) see if input is
                                        // between x2/x6/x10 & x3/x7/x11

/*At this point X input is known to be between x3/x7/x11 and x4/x8/x12 so a return
  value is interpolated. */
     asm("subf      r4,r8,r4       "); // r4 = r4 -r8 (input-xlow)
     asm("mullw     r4,r4,r11      "); // (input-Xlow)*(Yhigh-Ylow)
     asm("subf      r8,r8,r7       "); // (Xhigh-Xlow),  r8[32:63]=(X40-X3)
     asm("divw      r4,r4,r8       "); //  32 bit quotient
     asm("add       r3,r4,r9       "); //  return answer in r3
     asm("b         return_y");

/*At this point X input is known to be between x2/x6/x10 & x3/x7/x11 so a return
  value is interpolated. */
     asm("BackGrd1:");
     asm("evmergehir10,r0,r10      "); // r10[0:31]-->r10[32:63]= Y2
     asm("subf      r4,r6,r4       "); // compute (input-Xlow)
     asm("subf      r9,r10,r9      "); // (Yhigh-Ylow) or (Y3-Y2)
     asm("mullw     r4,r9,r4       "); // (input-Xlow)*(Yhigh-Ylow)
     asm("subf      r8,r6,r8       "); // (Xhigh-Xlow), r8[32:63]=(Y3-Y2)
     asm("divw      r4,r4,r8       "); // 32 bit quotient
     asm("add       r3,r4,r10      "); // return answer in r3
     asm("b         return_y");

/* Input is less than the current x values in r6, so it's either too low,
   or between x4 and x5, or between x8 and x9.*/
     asm("BackGrd2:");
     asm("evmergehilor4,r9,r4      "); // r9[0:31]=Yhigh r4[32:63]=input
     asm("evsubfw   r5,r12,r4      "); // else, compute (Yhigh-Ylow) and  (input-Xlow)
     asm("evmergehi r10,r0,r5      "); // r4[0:31]-->r10[32:63]
     asm("mullw     r10,r5,r10     "); // (Yhigh-Ylow)*(input-Xlow)
     asm("evmergehi r6,r0,r6       "); //  r6[0:31]-->r6[32:63]
     asm("subf      r8,r12,r6      "); // (Xhigh-Xlow), r8[32:63]=(X5-X4)
     asm("divw      r10,r10,r8     "); // 32 bit quotient
     asm("cmp       1,r4,r12       "); // Bounds Check, is input < first x value
     asm("bc        12,4,toolow    "); // if true, then input out of bounds
     asm("evmergehi r3,r0,r12      "); // r12[0:31]-->r12[32:63]
     asm("add       r3,r10,r3      "); // return answer in r3
     asm("b         return_y");


/* OUT OF BOUNDS */
```

```
     asm("toolow:");
     asm("evmergehi r3,r0,r4       "); //  merge contents of  r4[0:31]-->r10[32:63]
     asm("b         return_y       ");
     asm("toohigh:");
     asm("evlwhoux  r3,r6,r7       "); // r6 = last y value

     asm("return_y:");
     return;

}
```

## A.2 var_table_lookup_c( )

```
//-------------------------------------------------
// C based table lookup code
//-------------------------------------------------

unsigned short var_table_lookup_c( unsigned short *tbl_ptr, unsigned short in_value )
{

     unsigned char  num_XYpairs;         // number of x-y pairs
     unsigned short y_high;              // upper bound of the y value
     unsigned short y_low;               // lower bound of the y value
     unsigned short y_input_diff;        // difference between y_low and the Y value from the
                                         // input.
     unsigned short y_return;            // return value
     unsigned short x_input_diff;        // difference between input and the lower bounding X
                                         // value
     unsigned short x_diff;              // difference between the bounding X values
     unsigned short x_val;               // x value pointed at by x_ptr
     unsigned short *x_ptr;              // pointer to the current x_val

     /* get the count of xy pairs and check if 0 */
     if (num_XYpairs = *((unsigned char *) tbl_ptr) )
     {

          if (in_value <= *(tbl_ptr + 1)){   /* is the input <= first x? */
               y_return = *(tbl_ptr + 2);    /* if so, return first y */
          }
          else if (in_value >= *(tbl_ptr + ((num_XYpairs * 2) - 1))){
                                             /* is the input >= last x? */
               y_return = *(tbl_ptr + (num_XYpairs * 2));/* if so, return last y */
          }
          else {                             /* if neither above, it must be in the middle */
               x_ptr = tbl_ptr + 1;/* move ptr to first entry */

          /* search up through the X points to find which two X entries input is between */
          while( *x_ptr <= in_value)
               x_ptr+=2;

          x_val = *x_ptr;
          y_low = (unsigned short) *(x_ptr - 1);

          if (in_value == (unsigned short) *(x_ptr - 2)){
               y_return = y_low;
          }
          else {
               /* Determine the difference between the input and the lower bounding X value */
               x_input_diff = (in_value - *(x_ptr - 2));

               /* Determine the difference between the bounding X values */
               x_diff = (x_val - *(x_ptr - 2));
               y_high = (unsigned short) *(x_ptr + 1 );

               if (y_high > y_low){
                    /* if Slope is positive find the increase for the change in input then add
                    it to the y_low value */
                    y_input_diff =((unsigned short)((y_high - y_low) * x_input_diff))/x_diff;
                    y_return = (unsigned short) y_low + y_input_diff;
               }
               else {
               /* if Slope is negative find the decrease for the change in input then subtract
               it from the y_low value */
                    y_input_diff =((unsigned short)((y_low - y_high) * x_input_diff))/x_diff;
                    y_return = (unsigned short) y_low - y_input_diff;
                    }
               }
          }
          return(y_return);
     }
     else {
          /* return first Y value if Count == 0 */
          return(*(tbl_ptr + 2));
     }
}
```

# A.3 Tables

```
/* This table holds the data for the c-based table lookup function */

unsigned short table_var_c[]={
     0x0c00,   // header, first byte is number of x-y pairs
     0x110a,   // x1
     0xccc8,   // y1
     0x2214,   // x2
     0xbbC2,   // y2
     0x331e,   // x3
     0xaaB5,   // y3
     0x4428,   // x4
     0x99A6,   // y4
     0x5532,   // x5
     0x889f,   // y5
     0x663c,   // x6
     0x7784,   // y6
     0x7746,   // x7
     0x667c,   // y7
     0x8850,   // x8
     0x556c,   // y8
     0x995A,   // x9
     0x445d,   // y9
     0xaa64,   // x10
     0x3359,   // y10
     0xbb6e,   // x11
     0x222d,   // y11
     0xcc78,   // x12
     0x1116};  // y12

/* This table holds the data for the SIMD-based table lookup function */

unsigned short table_var_asm[]=
    { 0x0000,  // first four entries are the header
     0x0020,   // offset to start of y vals
     0x0000,
     0x0020,
     0x110a ,  // x1
     0x2214,   // x2
     0x331e,   // x3
     0x4428,   // x4
     0x5532,   // x5
     0x663c,   // x6
     0x7746,   // x7
     0x8850,   // x8
     0x995a,   // x9
     0xaa64,   // x10
     0xbb6e,   // x11
     0xcc78,   // x12

     0xccc8,   // y1 - Note the ‘interleaved’ order of the y data that allows
     0xaaB5,   // y3   the efficient use of the 64-bit SIMD instructions.
     0xbbc2,   // y2
     0x99a6,   // y4
     0x889f,   // y5
     0x667c,   // y7
     0x7784,   // y6
     0x556c,   // y8
     0x445d,   // y9
     0x222d,   // y11
     0x3359,   // y10
     0x1116};  // y12
```

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

*HOW TO REACH US:*

**USA/Europe/Locations not listed:**
Freescale Semiconductor Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

**Japan:**
Freescale Semiconductor Japan Ltd.
Technical Information Center
3-20-1, Minami-Azabu, Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

**Asia/Pacific:**
Freescale Semiconductor H.K. Ltd.
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
852-26668334

*Learn More:*
For more information about Freescale
Semiconductor products, please visit
**http://www.freescale.com**

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

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor 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 which may be provided in Freescale Semiconductor 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 Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.