Freescale Semiconductor
Application Note

AN2743
Rev. 0.1, 08/2004

Software Analysis on Genesi Pegasos II Using PMON and AltiVec

by *Maurie Ommerman and Sergei Larin*
CPD Applications
Freescale Semiconductor, Inc.
Austin, TX

This application note is the sixth in a series describing the Genesi Pegasos II system, which contains a PowerPC™ microprocessor, and its various applications. This document describes software analysis using the PMON facility for using its PowerPC processor performance measurement registers. It also describes the general compiler tool set, GCC, and some AltiVec constructs.

Contents

1 Introduction

This application note describes some features of the AltiVec constructs and the PMON kernel interface and how to use one of the PowerPC performance monitor measuring facilities. The PMON facility is an application written by the Freescale application team and is described in this applications note. Even though this document is part of the series on Genesi Pegasos II systems, the PMON facility is available on any Linux System running on a PowerPC processor. The PMON facility is preloaded on the Genesi Pegasos II system, but may be download by request for any PowerPC Linux Platform.This paper assumes that the user will log in as `guest` with password `guest`, and all the examples discussed in this paper, with the exception of the hello world programs, are in the /home/guest/fae-training-04/library directory.

2 Terminology

The following terms are used in this document.

Linux OS	Linux operating system
PMON	Performance monitor facility
GCC	Gnu compiler collections and GNU utilities
Performance monitors	MPC74xx processors contain registers that can be used to monitor system activity
AltiVec	Processing engine on the MPC74xx processors that allows for SIMD functionality
SIMD	Single instruction multiple data paths
POSIX	Portable operating system interface (POSIX) standardization effort that was formerly run by the POSIX standards committee
OEA	PowerPC operating environment architecture that defines supervisor-level resources typically required by an operating system

3 Introduction to Compiling on Linux with GCC

The GNU native tool chain is available on the Genesi Pegasos II system with both the Debian and Yellow Dog Linux distribution. This applications note will concentrate on the Debian Linux, however, it translates directly to the Yellow Dog Linux system and any other native PowerPC Linux distribution.

The GCC compiler executable command resides in the /usr/bin directory. The include files all reside in the /usr/include. All of the relative tool chain libraries reside in /usr/lib/gcc-lib. Version 3.3.3 GCC is used in this paper.

In order to compile any C application program, the simple `gcc` command can be used with all the default parameters.

All the examples, except the “hello world” programs discussed in this paper are in the /home/guest/fae-training-04/library directory.

3.1 The Objective and Tools for Achieving Software Development

The main objective is to familiarize ourselves with this new software development platform

3.1.1 Software Development Tools

The Pegasos software development system consists of the following items.

- MPC7447 + Discovery II
- Running Debian Linux
- Standard GNU tool set
- GCC V3.3.3 for PPC, Supports AltiVec
- GNU utilities: gdb, objdump etc.
- Customized tool set for PPC monitoring
  - PMON
  - SimG4+
- Text editor of your choice:

— vi, vim, emacs, gnome text editor (gedit)

3.1.2 PMON – Performance Monitor

PMON is a kernel module, which allows non-root users to set the performance monitors to count specific CPU activities, such as cycles. See Section 5, “Using the Performance Monitors for Performance Gathering” for more details.

PMON is a limited application which can count only (32 bits) approximately 4 billion items. Similar tools are available from commercial software vendors.

A user can read these registers and develop statistical analysis, however, to determine which CPU activities to gather requires changing OEA/supervisor registers, thus by calling PMON, the user can request that the performance registers collect specific counts.

Using PMON is described in this applications note, however, the implementation of the PMON kernel module is described in the Freescale application note *PMON Module—An Example of Writing Kernel Module Code for Debian 2.6 on Genesi Pegasos II* (AN2744).

3.1.3 A Simple “Hello World” Program

Navigate to the directory /home/fae-training-04/library directory, create a local directory, navigate to it and type in this program.

```
cd fae-training-04/library

mkdir localperson

cd localperson

<editor of your choice> hello.c

#include <stdio.h>

main()

{

        printf("Hello World!\n");

}
```

Compile and run this program, the executable elf file will be called a.out by default. Since the local directory is not in the PATH, local executables must be preceded by the ‘.’’/’ two characters, thus the construct, ./a.out.

You can get general help information for GCC with the `gcc -h` command.

```
gcc hello.c

./a.out

Hello World!
```

3.2 A Simple AltiVec “hello World” Program

Using the same program above, compile it with the AltiVec flags, -maltivec -mabi=altivec.

```
gcc -maltivec -mabi=altivec hello.c

./a.out

Hello World!
```

There is not any difference. That is because there are no AltiVec constructs in this program.

3.3 An AltiVec “Hello AltiVec from vecChar” Program with Some AltiVec Char Constructs.

AltiVec intrinsics are built into the GCC compiler and will be explained as they are encountered in this program. See Section 9, “References” 9 and 10.

This program illustrates some AltiVec constructs. The numbers allow a description of the constructs, do not type in the numbers if you wish to try this program for yourself.

```
guest@debian:~/fae-training-04/library/maurie$ cat -n vecChar.c
     1  #include <altivec.h>
     2  #include <stdio.h>
     3
     4  void print_char_vector(vector unsigned char *this_one);
     5  void print_int_vector(vector int *this_one);
     6  vector unsigned char vec_array[16];//only using vec_array[1]
     7
     8  main()
     9  {
    10  int i;
    11  vector unsigned char vec_a;
    12  unsigned char a1[16] __attribute__ ((aligned (16)))
    13          ={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
    14  unsigned char a2[16] __attribute__ ((aligned (16)))
    15          ={11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26};
    16          vec_a = vec_ld(0,a1);
    17          vec_array[1] = vec_ld(0,a2);
    18  printf("\nHello AltiVec from vecChar!\n");
    19  printf("vec_a = ");
    20  print_char_vector(&vec_a);
    21  printf("\n");
```

```
    22  printf("vec_array[1] = ");
    23  print_char_vector(&vec_array[1]);
    24  printf("\n");
    25  }
    26
    27  void print_char_vector(vector unsigned char *this_one)
    28  {
    29
printf("(%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,
%02x)",
    30                  ((unsigned char*)this_one)[0],
    31                  ((unsigned char*)this_one)[1],
    32                  ((unsigned char*)this_one)[2],
    33                  ((unsigned char*)this_one)[3],
    34                  ((unsigned char*)this_one)[4],
    35                  ((unsigned char*)this_one)[5],
    36                  ((unsigned char*)this_one)[6],
    37                  ((unsigned char*)this_one)[7],
    38                  ((unsigned char*)this_one)[8],
    39                  ((unsigned char*)this_one)[9],
    40                  ((unsigned char*)this_one)[10],
    41                  ((unsigned char*)this_one)[11],
    42                  ((unsigned char*)this_one)[12],
    43                  ((unsigned char*)this_one)[13],
    44                  ((unsigned char*)this_one)[14],
    45                  ((unsigned char*)this_one)[15]);
    46  }
    47  void print_int_vector(vector int *this_one)
    48  {
    49          printf("(%08x,%08x,%08x,%08x)",
    50                  ((int*)this_one)[0],
    51                  ((int*)this_one)[1],
    52                  ((int*)this_one)[2],
```

```
53                  ((int*)this_one)[3]);
54  }
```

The line numbers described here are the AltiVec constructs, the non-AltiVec C constructs will not be explained.

1 Include the GCC standard AltiVec header file. AltiVec intrinsics are built into the GCC compiler, this header will expand the constructs during compilation time.

4 and 5 are standard prototypes for these functions, which will be described later. However, the construct “vector”, indicates that a vector variable is being used.

5 and 11 The construct vector invokes a vector array of 16 vector (i.e. 128 bit) elements in memory, which are aligned on a 16 byte boundary, i.e. an address that ends with 4 bits of zero, e.g. 0x10105660.This example only uses one of those vectors.

12, 13, 14, and 15. These are normal character arrays, however, the attribute signature forces 16 byte boundary alignment. We will discuss this more in the align example in Section 3.5, “An AltiVec “Alignment” Program Demonstrating Alignment Considerations.”

16 and 17. Load a single vector vec_a from the address of the character array a1, and load a single 16 byte element of a vector array from the address of a character array a2. This will be discussed in more detail in Section 4, “Defining and Using an AltiVec Vector.” Suffice it to say that the 16 bytes starting at the address of the a1 character array, ‘1’,’2’,’3’, etc will be written to the vector (either a true AltiVec register, or a memory location representing that AltiVec register) and the result is that the location of vec_a will now contain the characters, ‘1’,’2’,’3’, etc. The same logic applies to line 17, except that the 2nd element (i.e. address of vec_array + 16 will contain the characters from the char array a2, ‘11’,’12’,’13’, etc.

20 calls the function print_char_vector, which I will discuss in line 27.

27 through 46 is a function to print the contents of the vector in memory, one byte at a time for a total of 16 bytes. Since a vector is a 16 byte quantity, we can treat each byte independently, similar to a char array of 16 bytes. Since we are giving the address of the first byte of the vector to this function, we access each additional byte by an array increment, which is equivalent to adding 1 to the previous address.

47 though 54 is a similar function to print the contents of the vector in memory, one int (i.e. 4 bytes) at a time, for a total of 4 ints (16 bytes). This function will be used in the next example explained in, Section 3.4, “An AltiVec “Hello AltiVec from vecInt” Program with Some AltiVec Int Constructs.”

We can compile and execute this example in several ways. I will describe three ways here.

1. Explicitly type in the command, as shown in the second example above in Section 3.2, “A Simple AltiVec “hello World” Program”
   ```
   gcc -maltivec -mabi=altivec vecChar.c
   ./a.out
   ```
2. Create a shell script that is easier to remember and type. Use any editor to create a file, call it compile, and add the `gcc` command, and change the permission set to execute. The contents of the file can just be input from the `cat` command, the $1 indicates the first parameter and $2 indicates the second parameter. In this case we only have one parameter vecChar.c

```
cat >compile

gcc -maltivec -mabi=altivec $1 $2

^d (i.e. a control d on the keyboard)
```

```
chmod 777 compile

./compile vecChar.c

./a.out
```

3. Create a Makefile. Edit the file. The permissions do not need to be changed. However, type in a tab character, not 8 spaces in front of the `make` commands. So ensure that while typing the contents to Makefile, that indentations are made with the tab key.

```
cat >Makefile

make: vecChar.c

        gcc -maltivec -mabi=altivec vecChar.c

clean:

        rm -rf *.o a.out

^d (i.e. control d)

make clean

make

./a.out
```

In any case the output we see is this.

```
guest@debian:~/fae-training-04/library/maurie$ ./a.out


Hello AltiVec from vecChar!

vec_a = (01,02,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f,10)

vec_array[1] = (0b,0c,0d,0e,0f,10,11,12,13,14,15,16,17,18,19,1a)
```

Note that the first array, vec_a contains the bytes loaded from char array a1, and vec_array[1] contains the bytes loaded from char array a2.

3.4 An AltiVec “Hello AltiVec from vecInt” Program with Some AltiVec Int Constructs.

The only difference in this program is that we are loading from an integer array and we can demonstrate the offset capability of the vec_ld(a,b) intrinsics.

The numbers allow a description of the constructs, do not type in the numbers if you wish to try this program for yourself.

```
guest@debian:~/fae-training-04/library/maurie$ cat -n vecInt.c

     1  #include <altivec.h>

     2  #include <stdio.h>
```

```
 3
 4  void print_char_vector(vector unsigned char *this_one);
 5  void print_int_vector(vector int *this_one);
 6  vector unsigned char vec_array[256];
 7  vector int vec_int;
 8
 9  main()
10  {
11  int i;
12  unsigned int  a3[16] __attribute__ ((aligned (16)))
13          ={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
14          vec_int = vec_ld(0,(vector int *)a3);
15  printf("\nHello AltiVec from vecInt!\n");
16  printf("vec_int offset by 0 = ");
17  print_int_vector(&vec_int);
18  printf("\n");
19  print_char_vector((vector unsigned char *)&vec_int);
20  printf("\n\n");
21          vec_int = vec_ld(16,(vector int *)a3);
22  printf("\nHello AltiVec from vecInt!\n");
23  printf("vec_int offset by 16 = ");
24  print_int_vector(&vec_int);
25  printf("\n\n");
26          vec_int = vec_ld(32,(vector int *)a3);
27  printf("\nHello AltiVec from vecInt!\n");
28  printf("vec_int offset by 32 = ");
29  print_int_vector(&vec_int);
30  printf("\n\n");
31          vec_int = vec_ld(48,(vector int *)a3);
32  printf("\nHello AltiVec from vecInt!\n");
33  printf("vec_int offset by 48 = ");
34  print_int_vector(&vec_int);
```

```
    35  printf("\n\n");
    36  }
    37
    38  void print_char_vector(vector unsigned char *this_one)
    39  {
    40
printf("(%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,
%02x)",
    41                  ((unsigned char*)this_one)[0],
    42                  ((unsigned char*)this_one)[1],
    43                  ((unsigned char*)this_one)[2],
    44                  ((unsigned char*)this_one)[3],
    45                  ((unsigned char*)this_one)[4],
    46                  ((unsigned char*)this_one)[5],
    47                  ((unsigned char*)this_one)[6],
    48                  ((unsigned char*)this_one)[7],
    49                  ((unsigned char*)this_one)[8],
    50                  ((unsigned char*)this_one)[9],
    51                  ((unsigned char*)this_one)[10],
    52                  ((unsigned char*)this_one)[11],
    53                  ((unsigned char*)this_one)[12],
    54                  ((unsigned char*)this_one)[13],
    55                  ((unsigned char*)this_one)[14],
    56                  ((unsigned char*)this_one)[15]);
    57  }
    58  void print_int_vector(vector int *this_one)
    59  {
    60          printf("(%08x,%08x,%08x,%08x)",
    61                  ((int*)this_one)[0],
    62                  ((int*)this_one)[1],
    63                  ((int*)this_one)[2],
    64                  ((int*)this_one)[3]);
    65  }
```

The code is almost the same. The code in these line numbers are different.

7 defines a vector of type int, instead of type char, vec_int. A vector of type char, indicates that the 128 bit vector is divided into 16 bytes of 8 bits each. A vector of type int, indicates that the 128 bit vector is divided into 4 ints of 32 bits each.

12 invokes an int array of 16 elements, each 32 bits long. The values, ‘1’,’2’,’3’, etc are stored into these ints. In the previous example, then the char values, ‘1’,’2’,’3’, were each 8 bits long. In this example, the ints ‘1’,’2’,’3’, etc are 32 bits long. Hence the char values contained all 16 values, ‘1’,’2’,’3’, etc in 128 bits. The int values contains only ‘1’,’2’,’3’,’4’ in the first 128 bits, and ‘5’,’6’.’7’,’8’ in the next 128 bits, etc.

14 stores the 4 int values, 128 bits of 0x00000001, 0x00000002, 0x00000003, 0x00000004, into the vec_int.

21 stores the 4 int values, 128 bits of 0x0000005, 0x00000006, 0x00000007, 0x00000008 into the vec-int, because the a value in vec_ld(a,b) indicates to offset 16 bytes. Now if 0<=a<16, we offset by 0, because a must be a multiple of 16. Thus in this case, 16<=a<32, so the offset is 16 bytes.

26 store the 4 int values, 128 bits of 0x00000009, etc in vec_int.

31 stores the next int values, starting with 0x0000000d.

We can now use any of the three methods described in the previous example to compile and execute this program, changing vecChar.c to vecInt.c

```
guest@debian:~/fae-training-04/library/maurie$ ./compile vecInt.c

guest@debian:~/fae-training-04/library/maurie$ ./a.out


Hello AltiVec from vecInt!

vec_int offset by 0 = (00000001,00000002,00000003,00000004)

(00,00,00,01,00,00,00,02,00,00,00,03,00,00,00,04)



Hello AltiVec from vecInt!

vec_int offset by 16 = (00000005,00000006,00000007,00000008)



Hello AltiVec from vecInt!

vec_int offset by 32 = (00000009,0000000a,0000000b,0000000c)



Hello AltiVec from vecInt!

vec_int offset by 48 = (0000000d,0000000e,0000000f,00000010)
```

```
guest@debian:~/fae-training-04/library/maurie$
```

3.5 An AltiVec “Alignment” Program Demonstrating Alignment Considerations

This example demonstrates the necessity of forcing alignment onto a 16 byte boundary for AltiVec vector operations. You can find this example in /home/guest/fae-training-04/library/align. It has been modified slightly to emphasize what we are demonstrating.

.

```
guest@debian:~/fae-training-04/library/align$ cat -n align.c
     1
     2  // Modified slightly from the example
     3  // Alignment example
     4  //
     5
     6  #include <altivec.h>
     7  #include <stdio.h>
     8
     9
    10  #define START_TIMER     \
    11     start_time      = read_744x_upmc1();\
    12     start_ins       = read_744x_upmc2();
    13
    14  #define STOP_TIMER \
    15     asm volatile("eieio"); \
    16     stop_time       = read_744x_upmc1(); \
    17     stop_ins        = read_744x_upmc2();
    18
    19  #if TRACE
    20  #define START_TRACING         asm (".long 0x14000001");
    21  #define STOP_TRACING          asm (".long 0x14000002");
    22  #define MAX_SIZE        64
    23  #define REPEAT          1
    24  #else
```

```
25  #define START_TRACING
26  #define STOP_TRACING
27  #define MAX_SIZE        256
28  #define REPEAT          10000
29  #endif
30
31  #define FORCE_ALIGNMENT 0   /*
32                               * 1 forces alignment
33                               * 0 forces non alignment
34                               */
35
36  int start_pmon(int p1,int p2,int p3, int p4);
37  unsigned int read_744x_upmc1(void);
38  unsigned int read_744x_upmc2(void);
39  unsigned int read_744x_upmc3(void);
40  unsigned int read_744x_upmc4(void);
41
42  char outOfAlignment;
43  #if FORCE_ALIGNMENT
44  char aa_array[MAX_SIZE] __attribute__ ((aligned (16))) ;
45  char ab_array[MAX_SIZE] __attribute__ ((aligned (16))) ;
46  #else
47  char aa_array[MAX_SIZE];
48  char ab_array[MAX_SIZE];
49  #endif
50
51  void print_int_vector(vector int *this_one){
52          printf("{%08x,%08x,%08x,%08x}\n",
53                  ((int *)this_one)[0],
54                  ((int *)this_one)[1],
55                  ((int *)this_one)[2],
56                  ((int *)this_one)[3]);
```

```
    57  }
    58
    59  void print_char_vector(vector unsigned char *this_one){
    60
printf("{%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,%02x,
%02x}\n",
    61                  ((unsigned char*)this_one)[0],
    62                  ((unsigned char*)this_one)[1],
    63                  ((unsigned char*)this_one)[2],
    64                  ((unsigned char*)this_one)[3],
    65                  ((unsigned char*)this_one)[4],
    66                  ((unsigned char*)this_one)[5],
    67                  ((unsigned char*)this_one)[6],
    68                  ((unsigned char*)this_one)[7],
    69                  ((unsigned char*)this_one)[8],
    70                  ((unsigned char*)this_one)[9],
    71                  ((unsigned char*)this_one)[10],
    72                  ((unsigned char*)this_one)[11],
    73                  ((unsigned char*)this_one)[12],
    74                  ((unsigned char*)this_one)[13],
    75                  ((unsigned char*)this_one)[14],
    76                  ((unsigned char*)this_one)[15]);
    77  }
    78
    79  vector unsigned char vectorLoadUnaligned( vector unsigned char *v ){
    80       vector unsigned char permuteVector = vec_lvsl( 0, (int*) v );
    81       vector unsigned char low  = vec_ld( 0, v );
    82       vector unsigned char high = vec_ld( 16, v );
    83       return vec_perm( low, high, permuteVector );
    84  }
    85
    86  void vectorStoreUnaligned( vector unsigned char v, vector unsigned char
*where){
```

```
    87      vector unsigned char  permuteVector         = vec_lvsr( 0, (int*) where );
    88       vector unsigned char  low,high,tmp,mask;
    89       vector unsigned char  ones                   = vec_splat_u8( 0xff );
    90       vector unsigned char  zeroes                 = vec_splat_u8( 0 );
    91
    92       low    = vec_ld ( 0, where );        //Load the surrounding area
    93       high   = vec_ld ( 16, where );
    94       //Make a mask for which parts of the vectors to swap out
    95       mask   = vec_perm( zeroes, ones, permuteVector );
    96       tmp    = vec_perm( tmp, tmp, permuteVector );         //Right rotate our
input data
    97       low    = vec_sel( tmp, low, mask );           // Insert masked data to
aligned vector
    98       high   = vec_sel( high, v, mask );
    99
   100     vec_st ( low, 0, where );                          //Store aligned results
   101      vec_st ( high, 16, where );
   102  }
   103
   104  int main(){
   105
   106     vector unsigned char vec_a;
   107     int j,i;
   108     int s1,s2,s3;
   109     unsigned int    start_time, stop_time;
   110     unsigned int    start_ins, stop_ins;
   111
   112     for(i=0;i<MAX_SIZE;i++){
   113          aa_array[i] = i;
   114          ab_array[i] = i;
   115     }
   116
   117     printf("\nAlignment Test\n");
```

```
118
119     start_pmon(1,2,1,2);
120
121      START_TIMER;
122      for(i=0;i<REPEAT;i++)
123           vec_a = vec_ld(0,(vector unsigned char *)aa_array);
124      STOP_TIMER;
125
126      print_char_vector(&vec_a);
127    printf("%d\tInstructions,\t%d Cycles\t%f IPC \n",
128           stop_time-start_time,
129           stop_ins-start_ins,
130           ((double)(stop_ins-start_ins)/(double)(stop_time-start_time)));
131
132
133      START_TIMER;
134      for(i=0;i<REPEAT;i++)
135           vec_a = vectorLoadUnaligned( (vector unsigned char *)aa_array);
136      STOP_TIMER;
137
138      print_char_vector(&vec_a);
139      printf("%d\tInstructions,\t%d Cycles\t%f IPC\n",
140           stop_time-start_time,
141           stop_ins-start_ins,
142           ((double)(stop_ins-start_ins)/(double)(stop_time-start_time)));
143
144      return 0;
145   }
guest@debian:~/fae-training-04/library/align$
```

This example serves double duty, it demonstrates the necessities of alignment, and is a simple example of using performance monitoring, which is in Section 3.6, “An AltiVec Program Demonstrating the Use of PMON for Obtaining Performance Statistics.” Here the lines associated with demonstrating alignment are discussed.

31 through 34 This defined variable can force 16 byte alignment or not, as shown starting in line 42.

42 through 49 can force 16 byte alignment due to the __attribute__ ((aligned(16))) intrinsic. When FORCE_ALIGNMENT is 0 we skip the two lines with this attribute and use the two lines without the attribute. line 42, aligns to a byte, which will not be on a 16 byte address. FORCE_ALIGNMENT is 1, we use the two lines with the attribute and skip the two lines without the attribute. Thus, this code will run with non 16 byte alignment and will not give us the correct answer. We will change this later and show the correct answer.

51 through 77 is the previously described print an int vector and a char vector.

79 through 84 is a function to load unaligned vectors correctly.

86 through 102 can store vectors correctly to unaligned memory.

112 though 115 fills these two aligned or unaligned, depending on FORCE_ALIGNMENT, arrays with the numbers 0 through 15.

123 stores the 16 bytes starting at aa_array into the vector vec_a, but since it is unaligned we will get unexpected results.

Since the macro REPEAT is set to 1, we only perform this loop once.

135 stores the 16 bytes starting at ab_array into the vector vec_a, but regardless of alignment of this array, the function vectorLoadUnaligned will aligned the data properly.

126 and 138 print the vectors.

Compile and execute this example using this Makefile.

```
guest@debian:~/fae-training-04/library/align$ cat Makefile
test: align.c pmon.c
        gcc -maltivec -mabi=altivec pmon.c align.c -o  $@
clean:
        rm -rf *.o pmon_test
make clean
make
./test
guest@debian:~/fae-training-04/library/align$ make clean
rm -rf *.o pmon_test
guest@debian:~/fae-training-04/library/align$ make
gcc -maltivec -mabi=altivec pmon.c align.c -o  test
guest@debian:~/fae-training-04/library/align$ ./test
```

Ignore the lines in italics, they will be described later.

```
Alignment Test
```

CPU = 7457,

CPU 7457, has 6 PMCs

Monitoring events are PMC[0]:1

Monitoring events are PMC[1]:2

Monitoring events are PMC[2]:1

Monitoring events are PMC[3]:2

Monitoring events are PMC[4]:0

Monitoring events are PMC[5]:0

```
{fc,fd,fe,ff,00,01,02,03,04,05,06,07,08,09,0a,0b}
```

.

125266 Instructions, 110163 Cycles 0.879433 IPC

```
{00,01,02,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f}
```

549870 Instructions, 480786 Cycles 0.874363 IPC

The ignored lines are PMON output and will be explained in the next example, in Section 3.6, “An AltiVec Program Demonstrating the Use of PMON for Obtaining Performance Statistics.”

Looking at the two lines of output, we see that the first un aligned array starts with the value ‘fc’ and it should start with ‘00’. This is because the arrays aa_array and ab_array are not on a 16 byte boundary. The second unaligned array vector will print correctly, since the vectorLoadUnaligned function aligned the data before storing it in the vector.

By using the command, which will disassemble an elf executable file:

```
guest@debian:~/fae-training-04/library/align$ objdump -D test > j
```

Looking at the assembly saved in file, j, we see that aa_array and ab_array start at address,10011a04, and 10011b04, which are not on a 16 byte boundary, the last 4 bits of the address are not zero.

```
10011a04 <ab_array>:
        ...


10011b04 <aa_array>:
        ...
```

Recompile and rerun the program, setting FORCE_ALIGNMENT to 1 and we get this result.

```
guest@debian:~/fae-training-04/library/align$ make clean
rm -rf *.o pmon_test
guest@debian:~/fae-training-04/library/align$ make
gcc -maltivec -mabi=altivec pmon.c align.c -o  test
```

```
guest@debian:~/fae-training-04/library/align$ ./test
```

Ignore the lines in italics, they will be described later.

```
Alignment Test
```

CPU = 7457,

CPU 7457, has 6 PMCs

Monitoring events are PMC[0]:1

Monitoring events are PMC[1]:2

Monitoring events are PMC[2]:1

Monitoring events are PMC[3]:2

Monitoring events are PMC[4]:0

Monitoring events are PMC[5]:0

```
{00,01,02,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f}
```

125423 Instructions, 110162 Cycles 0.878324 IPC

```
{00,01,02,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f}
```

549744 Instructions, 480785 Cycles 0.874562 IPC

```
guest@debian:~/fae-training-04/library/align$
```

We now see that both arrays are loaded correctly because the aa_array and the ab_array are on a 16 byte boundary.

```
10011a10 <ab_array>:
        ...

10011b10 <aa_array>:
        ...
```

Thus, it is important to guarantee 16 byte alignment for all memory that will be associated with AltiVec operations.

3.5.1 More Information on AltiVec Data Alignment.

3.5.1.1 Obtaining Data Alignment for AltiVec with Compiler Constructs

It is strongly recommended that you align all data structures to 16 byte boundary if AltiVec is used

Different compilers have different means of achieving it, but all of them have some method.

Here is a GCC example…

```
#include <altivec.h>
typedef union {
```

```
vector unsigned int vec;
int elements[4];
} LongVector __attribute__ ((aligned (16)));

unsigned char bitbuf8[16] __attribute__ ((aligned (16)))
={
  #include "attribute_table.txt"
};
```

Where the file, attribute_table.txt, is in the local directory and contains some constant data, such as “data”.

In this example every variable of data type LongVector will be aligned on quad-word boundaries and bitBuf8 is also aligned In this case bitBuf8 will be filled with the data in the _table.txt file that is in this local directory. I.e., including attribute_table.txt is another way of initializing an array.

Data Alignment is absolutely critical for mapping algorithms on AltiVec.

3.5.1.2 Obtaining Data Alignment for AltiVec with a Function

Loading Unaligned Data using the function, vectorLoadUnaligned, requires loading twice the data you really need, which is more inefficient then just aligning with the compiler __attribute__ ((aligned (16))).

```
vector unsigned char vectorLoadUnaligned( vector unsigned char *v ){
     vector unsigned char permuteVector = vec_lvsl( 0, (int*) v );
     vector unsigned char low  = vec_ld( 0, v );
     vector unsigned char high = vec_ld( 16, v );
     return vec_perm( low, high, permuteVector );
}
```

3.5.2 Obtaining Data Alignment for Altivec with a More Efficient Function

This function is more efficient than the previous one, vectorLoadUnaligned, but is still less efficient then just aligning the data with the compiler __attribute__ ((aligned (16))).

```
void vectorStoreUnaligned( vector unsigned char v, vector unsigned char *where){
    vector unsigned char  permuteVector = vec_lvsr( 0, (int*) where );
    vector unsigned char  low,high,tmp,mask;
    vector unsigned char  ones = vec_splat_u8( 0xff );
    vector unsigned char  zeroes = vec_splat_u8( 0 );

    low     = vec_ld ( 0, where ); //Load the surrounding area
```

```
    high    = vec_ld ( 16, where );
    //Make a mask for which parts of the vectors to swap out
    mask   = vec_perm( zeroes, ones, permuteVector );
    tmp     = vec_perm( tmp, tmp, permuteVector ); //Right rotate our input data
    low      = vec_sel( tmp, low, mask ); // Insert masked data to aligned vector
    high     = vec_sel( high, v, mask );

    vec_st ( low, 0, where ); //Store aligned results
    vec_st ( high, 16, where );
}
```

3.6 An AltiVec Program Demonstrating the Use of PMON for Obtaining Performance Statistics

The code for this example and that used in Section 3.5, “An AltiVec “Alignment” Program Demonstrating Alignment Considerations” is the same. What follows discusses the lines that are associated with using the pmon.c code facility.

A more complete discussion of using the PMON facility is discussed in Section 5, “Using the Performance Monitors for Performance Gathering.”

10 through 17 defines a macro that can be used to turn on performance monitor gathering, the functions here are described in Section 5, “Using the Performance Monitors for Performance Gathering.”

11,12 and 16, 17 call the functions read_744x_upmc, which is defined in pmon.c, described in Section 5.2.2, “PMON Interface File Code.”

19 through 29 is used for the simg4plus facility

36 through 40 are prototypes.

119 through 121 call the PMON facility and tell it to monitor performance monitors 1 and 2, which count number of instructions and number of cycles, see Section 5.2.2, “PMON Interface File Code.”

124 turns off the counters.

127 through 130 display the results, the number of instructions and cycles used to perform the code between the START_TIMER and the STOP_TIMER.

133 and 134 same as 119 through 121, except that it calls the function vecotrLoadUnaligned, which will execute more instruction then the previous 133 and 134 lines.

136 same as 124.

139 through 142 same as 127 through 130.

Lets look at the result of running this program again and explain the previously ignored output lines.

```
guest@debian:~/fae-training-04/library/align$ ./test > j
```

```
guest@debian:~/fae-training-04/library/align$ cat -n j
     1
     2  Alignment Test
     3  CPU = 7457,
     4  CPU 7457, has 6 PMCs
     5  Monitoring events are PMC[0]:1
     6  Monitoring events are PMC[1]:2
     7  Monitoring events are PMC[2]:1
     8  Monitoring events are PMC[3]:2
     9  Monitoring events are PMC[4]:0
    10  Monitoring events are PMC[5]:0
    11  {00,01,02,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f}
    12  124813  Instructions,   110166 Cycles   0.882648 IPC
    13  {00,01,02,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f}
    14  541317  Instructions,   480033 Cycles   0.886787 IPC
guest@debian:~/fae-training-04/library/align$
```

3 and 4 tells us what processor we are using and the number of performance monitors that are available.

5 though 8 indicate we are monitoring 1 and 2, twice, i.e. instructions and cycles. Note we are only using PMC1 and PMC2 in the START_TIME and STOP_TIME macros. We are ignoring the other 4 counters.

12 tells us that it took 124813 instructions and 110166 cycles to load vec_a for one (REPEAT=1) time. By dividing the instructions by the cycles we get 0.882648 instructions per cycle.

14 tells us that we executed 4 times the number of instructions and cycles to perform vectorLoadUnalligned, than the previous code, which just loaded aligned data. Hence, it is obviously more efficient to use 16 byte aligned data then to execute a function to align non 16 byte aligned data for AltiVec operations.

As you can see this is a rudimentary look at our code, but we can become much more sophisticated in our measurements, which we will see in Section 7, “More Advanced Examples”

4 Defining and Using an AltiVec Vector

AltiVec vectors are a 128 bit quantity aligned on a 16 byte boundary. How do we manipulate it?

We load it with the vec_ld(a,b) which loads 16 bytes into the vector, regardless of how we define it, char, short, int, float, double. So the receiver is a vector and the sender is an offset (offset by 16) and a memory address of 16 bytes.

Thus if we define an array of 16 bytes and a char vector as

```
char aa_array[MAX_SIZE] __attribute__  ((aligned (16)));

vector unsigned char vec_a;
```

Remember to align to 16 bytes. And then we fill it with numbers from 0 to 15.

```
for(i=0;i<MAX_SIZE;i++)
        {
                aa_array[i]=i;
}
```

Now starting at the address of aa_array, i.e. &aa_array, we have the 16 bytes set to the numbers from 0 to 15.

E.g. assume that aa_array starts at address 10011b10 <aa_array>, then the following values are stored in memory.

```
10011b10 0
10011b11 1
10011b12 2
...
10011b1f 15
```

Now when we use the intrinsic,

```
vec_a = vec_ld(0,(vector unsigned char *)aa_array);
```

We are loading the values one byte at a time from 10011b10 through 10011b1f into the vector (register or memory location). thus the vector vec_a now contains the char values from 0 to 1 in each of the bytes of the vector.

For int arrays, we have the same scenario, however, each int is 4 bytes. So for the example below, we are assigning 16 integer values of 4 bytes each, which is 64 bytes.

```
vector int vec_int;
unsigned int  a3[16] __attribute__ ((aligned (16)))
        ={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
        vec_int = vec_ld(0,(vector int *)a3);
```

Lets just consider the first 4 integers, which is 16 bytes. Assume that int array a3 starts at 10011b10.

```
10011b10 0
10011b11 0
10011b12 0
10011b13 1
10011b14 0
10011b15 0
10011b16 0
10011b17 2
.....
10011b1c 0
```

```
10011b1d 0

10011b1e 0

10011b1f 4
```

Now after the vec_ld instruction, the vector, vec_int, has 16 bytes of data copied from the addresses 10011b10 through 10011b1f, which is 0,0,0,1,0,0,0,2,0,0,0,3,0,0,0,4. It still has 16 bytes, but only four integer values.

5 Using the Performance Monitors for Performance Gathering

5.1 General Description

All G4 parts contain special hardware to collect certain statistical information about the CPU state and events.

The MPC7447 contains six performance counters accessible as privileged SPRs: PMC1-PMC6, which can monitor up to 242 unique events.

Normally these are 32bit HW counters. If you count an event every cycle at a speed of 1GHz, you will overflow these counters in 4.3 seconds. It is possible to extend them to 64 bits, and/or write code to off load the results, but that is beyond the scope of this applications note.

The full list of monitor able statistics is given in *MPC7450 RISC Microprocessor Family User's Manual.*

Below is a short list.

Table 11-9. PMC1 Events—MMCR0[PMC1SEL] Select Encodings

Number	Event	Description
0 (000_0000)	Nothing	Register counter holds current value
1 (000_0001)	Processor cycles	Counts every processor cycle
2 (000_0010)	Instructions completed	Counts all completed PowerPC and AltiVec instructions. Load/store multiple instructions (**lmw**, **stmw**) and load/store string instructions (**lswl**, **lswx**, **stswl**, **stswx**) are only counted once. Does not include folded branches. The counter can increment by 0, 1, 2, or 3, depending on the number of completed instructions per cycle. Branch folding must be disabled (HID0[FOLD] = 0) in order to count all the instructions.
3 (000_0011)	TBL bit transitions	Counts transitions from 0 to 1 of TBL bits specified through MMCR0[TBSEL]. 00 = uses the TBL[31] bit to count 01 = uses the TBL[23] bit to count 10 = uses the TBL[19] bit to count 11 = uses the TBL[15] bit to count
4 (000_0100)	Instructions dispatched	Counts dispatched instructions.The counter can increment by 0, 1, 2, or 3, depending on the number of dispatched instructions per cycle.

In this applications note, we have seen the use of counting cycles and processor instructions, which can be seen from the above table are counter 1, processor cycles, and 2, instructions completed.

5.2 A Code Example Using the PMON Facility to Gather Performance Statistics

As described earlier, the align.c program is linked with the pmon.c program, which supplies the calls to the PMON facility, which is described in Section 6, “Using the PMON Facility.”

Looking in the directory /home/guest/fae_training-04/library/align we see two c files, align.c and pmon.c.

5.2.1 Makefile

The Makefile shown below, compiles align.c and pmon.c and links them together. Since the target line 1 is named test, then the gcc line 2 generates an elf executable named test, and that is the file we execute with the `./test` command. There is a bug in this makefile, in the clean target line 4, we rm pmon_test, however, the Makefile generates the file, test. So, the clean target does not work, change pmon_test to test in the clean target and the clean target will work as expected, which is to remove the executable.

```
guest@debian:~/fae-training-04/library/align$ cat -n Makefile
     1  test: align.c pmon.c
     2          gcc -maltivec -mabi=altivec pmon.c align.c -o  $@
     3  clean:
     4          rm -rf *.o pmon_test
```

```
guest@debian:~/fae-training-04/library/align$
```

Change the following line in the makefile to remove the executable, text, with the clean target.

```
        4           rm -rf *.o test
```

5.2.2 PMON Interface File Code

We have already discussed align.c

The program pmon.c described below is intended to be linked with any other program that wishes to set up performance monitors. It has a limit of 4 performance monitor registers that can be used at any one time. It can be easily changed to handle up to 6 registers by changing line 24 to accept 6 arguments, and changing lines 88 and 89 to use these two new arguments.

Another example of this interface program rewritten to be stand-alone and allow the user to input performance monitor register numbers via the keyboard is available in /root/ppctools/pmon/usr/pmon_test.c.

The pmon.c program is listed here, taken from /home/guest/fae-training-04/library/align, with line numbers, which are obtained with the `cat -n` command. A description of all these lines follow.

The program pmon.c described below. It is intended to be linked with any other program that wishes to set up performance monitors. It has a limit of 4 performance monitor registers that can be used at any one time. It can be easily changed to handle up to 6 registers by changing line 24 to accept 6 arguments, and changing lines 88 and 89 to use these two new arguments.

Another example of this interface program rewritten to be stand-alone and allow the user to input performance monitor register numbers via the keyboard is available in /root/ppctools/pmon/usr/pmon_test.c.

guest@debian:~/fae-training-04/library/align$ cat -n pmon.c

```
guest@debian:~/fae-training-04/library/align$ cat -n pmon.c


1
/*****************************************************************************

     2   * Filename: pmon_test.c

    3   * Note:     this file test kernel module pmon.c which is registered as a char
device

     4   *            at /dev/pmon


5
*****************************************************************************/

     6  #define _GNU_SOURCE

     7  #include <stdio.h>

     8  #include <stdlib.h>
```

```
 9  #include <string.h>
10  #include <fcntl.h>
11  #include <unistd.h>
12  #include <sys/uio.h>
13  #define MAX_PMC_NUM (6)
14  static unsigned int pmc_sel[MAX_PMC_NUM];
15
16  unsigned int read_744x_upmc1(void);
17  unsigned int read_744x_upmc2(void);
18  unsigned int read_744x_upmc3(void);
19  unsigned int read_744x_upmc4(void);
20  unsigned int read_744x_upmc5(void);
21  unsigned int read_744x_upmc6(void);
22  void show_upmcs(unsigned int* upmc);
23
24  int start_pmon(int p1,int p2,int p3, int p4)
25  {
26    static unsigned int upmc_begin[6],upmc_end[6];
27    int i, fd, byteCount, len, n_read;
28    unsigned int cycles;
29    char* textLine= NULL;
30    char item[32], delim[32], name[32];
31    int total_pmc= 0;
32    FILE* p_cpuinfo;
33
34    /* id CPU to decide how many PMCs we have on this machine */
35    p_cpuinfo = fopen("/proc/cpuinfo", "r");
36    if(p_cpuinfo == NULL)
37      {
38        printf("ERR: unable to open  cpuinfo \n");
39        return 0;
40      }
```

```
41    while((n_read = getline(&textLine, &len, p_cpuinfo))!=-1)
42      {
43        sscanf( textLine,"%s%s%s*", item, delim,name);
44  #ifdef DBG_PMON
45        printf("INFO: getline %s\n", textLine);
46        printf("INFO:item=%s, delim=%s, name=%s\n", item, delim,name);
47  #endif
48        if(!memcmp(item, "cpu", 4))
49          {
50            printf("CPU = %s\n", name);
51            if(!memcmp(name, "744",3))
52              total_pmc = 6;
53            else if(!memcmp(name, "745",3))
54              total_pmc = 6;
55            else if(!memcmp(name, "741",3))
56              total_pmc = 4;
57            else
58              {
59                printf("ERR: unsupported CPU %s\n", name);
60                return 0;
61              }
62            printf("CPU %s has %d PMCs\n", name, total_pmc);
63            break;
64          }
65      }
66    if(textLine)
67      free(textLine);
68    fclose(p_cpuinfo);
69
70    /* FIXME: let usr choose which event for which PMCs based on cpuinfo */
71    /* hardcode pmc event number for each pmc */
72    /*
```

```
73    printf("Please choose PMC events\n");
74    for(i=0;i<total_pmc;i++)
75      {
76        printf("PMC[%d] event:\t", i);
77        if(getline(&textLine, &len, stdin)!=-1)
78          {
79            sscanf( textLine,"%d*", &pmc_sel[i]);
80            //      printf("%d\n", pmc_sel[i]);
81          }
82      }
83      */
84    pmc_sel[0]    = p1;
85    pmc_sel[1]    = p2;
86    pmc_sel[2]    = p3;
87    pmc_sel[3]    = p4;
88    pmc_sel[4]    = 0;
89    pmc_sel[5]    = 0;
90    for (i=0;i<total_pmc;i++)
91          printf("Monitoring events are PMC[%d]:%d\n", i, pmc_sel[i]);
92
93    fd = open("/dev/pmon", O_RDWR);
94    if(fd == -1)
95      {
96        printf("ERR: unable to open device /dev/pmon\n");
97        return 0;
98      }
99
100   /* Write to pmc selection information to pmon device driver */
101   write(fd, pmc_sel, sizeof(pmc_sel));
102   for(i=0; i<=10;i++)
103     {
104       byteCount=read(fd, &cycles, sizeof(int) );
```

```
105         if(byteCount == -1)
106           {
107             printf("ERR: read failed\n");
108             return 0; // can read again
109           }
110         else if (byteCount < sizeof(cycles))
111           {
112             printf("ERR: not read enough data\n");
113             return 0; // can read again
114           }
115         //printf("PMC count = 0x%08X\n", cycles);
116       }
117     close(fd);
118     //show_upmcs(upmc_begin);
119     //printf("Running my code ..\n\n\n");
120     //asm volatile("eieio");
121     //show_upmcs(upmc_end);
122     return 0;
123   }
124
125    void show_upmcs(unsigned int* upmc)
126   {
127     int i;
128     upmc[0]= read_744x_upmc1();
129     upmc[1]= read_744x_upmc2();
130     upmc[2]= read_744x_upmc3();
131     upmc[3]= read_744x_upmc4();
132     upmc[4]= read_744x_upmc5();
133     upmc[5]= read_744x_upmc6();
134     for (i=0;i<6;i++)
135       printf("UPMC[%d]=0x%08x\n", i,upmc[i]);
136   }
```

```
137
138
139  unsigned int read_744x_upmc1(void)
140  {
141    unsigned int val32;
142    asm volatile("mfspr %0, 937" : "=r"(val32));
143    return val32;
144  }
145  unsigned int read_744x_upmc2(void)
146  {
147    unsigned int val32;
148    asm volatile("mfspr %0, 938" : "=r"(val32));
149    return val32;
150  }
151  unsigned int read_744x_upmc3(void)
152  {
153    unsigned int val32;
154    asm volatile("mfspr %0, 941" : "=r"(val32));
155    return val32;
156  }
157  unsigned int read_744x_upmc4(void)
158  {
159    unsigned int val32;
160    asm volatile("mfspr %0, 942" : "=r"(val32));
161    return val32;
162  }
163  unsigned int read_744x_upmc5(void)
164  {
165    unsigned int val32;
166    asm volatile("mfspr %0, 929" : "=r"(val32));
167    return val32;
168  }
```

```
169  unsigned int read_744x_upmc6(void)
170  {
171    unsigned int val32;
172    asm volatile("mfspr %0, 930" : "=r"(val32));
173    return val32;
174  }
```

Description of the output lines follow:

1-5 comments

6 Don’t know what defining _GNU_SOURCE does

7 through 12 include header files from the standard include directory at /usr/include, not from the kernel sources.

13 Define the maximum PMC counters

14 create an array of unsigned ints to store the values for each counter selection.

16 through 21 are prototypes for functions that read the non privileged counter registers UPMC1 through UPMC6.

22 is the prototype for the function that prints out the contents of the UPMC registers obtained from the functions prototypes in lines 16 through 21. It is not used in this program.

24 through 123 is the function that initializes the performance registers.

26 through 31 are declarations.

32 through 68 declares a FILE type which will be used to read the /proc/cpuinfo pseudo file, which specifies the CPU information on the running system. Try the shell command `cat /proc/cpuinfo.` It is this information that is being read here and checking for the existence of an MPC744x, MPC745x, or MPC741x processor, which are the only processors that have performance monitor registers.

70 through 83 is commented out, hence we do not give the caller the opportunity to choose which events to count, we just use the events passed to this function in four arguments, p1 through p4.

84 through 89, set the pmc_sel to the arguments in preparation to monitor these counters.

90 and 91 print out the events that are going to be monitored, which corresponds to the lines 5 through 10 in the output listings below

93 through 99, opens the char device we have defined for this PMON facility, /dev/pmon using the standard IO call, open, which will invoke the pmon26.ko module function, pmon_open. We check to see if it is available and if not print the error message we see in line 11 in the Section 5.2.3, “Results When /dev/pmon is Not Available.”

101 Calls the pmon26.ko module function, pmon_write, to write the selection bits to the privileged performance monitor selector registers, MMCR0 and MMCR1 and zero the counters.

102 through 117 calls the pmon26.ko function pmon_read to read the contents of SPR937, UPMC1, which does not even require this module, and then we would print out the values in line 115, but it is commented out, so this is some debug code.

118 through 121 are commented out, so they are some debug lines.

122 through 123 returns from this function.

125 through 136 is the function to print out all the UMPC values, which are not privileged. This function is not called by align.c.

139 through 144 is a function to read the UPMC1, SPR938 register, which is not privileged. Thus we can just read this register as a normal user. It is used by align.c to read each of the counters and print out the lines 13 and 15 in the results shown below.

145 through 174 are the functions to get all the other counter values.

5.2.3 Results When /dev/pmon is Not Available

```
guest@debian:~/fae-training-04/library/align
```

Running the executable again, we get this result.

```
guest@debian:~/fae-training-04/library/align$ ./test > j

guest@debian:~/fae-training-04/library/align$ cat -n j

     1
     2  Alignment Test
     3  CPU = 7457,
     4  CPU 7457, has 6 PMCs
     5  Monitoring events are PMC[0]:1
     6  Monitoring events are PMC[1]:2
     7  Monitoring events are PMC[2]:1
     8  Monitoring events are PMC[3]:2
     9  Monitoring events are PMC[4]:0
    10  Monitoring events are PMC[5]:0
    11  ERR: unable to open device /dev/pmon
    12  {00,01,02,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f}
    13  0       Instructions,   0 Cycles        nan IPC
    14  {00,01,02,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f}
    15  0       Instructions,   0 Cycles        nan IPC
```

In this case, line 13 and 15 gives 0 answers, because at line 11, the module gave an error, because pmon26.ko has not been started or /dev/pmon does not exist, or /dev/pmon has the wrong permissions, they must be 777 all permissions.

5.2.4 All These Conditions Must Be Met for the PMON Facility to Work.

1. The module, pmon26.ko must be built
2. The module, pmon26.ko must be installed, `insmod pmon26.ko`
3. /dev/pmon must be created, `mknod /dev/pmon c <node number> 0`
4. The permissions must be 777, `chmod 777 /dev/pmon`

The node number can be determined from the /proc/devices file. After the `insmod pmon26.ko`, look at the /dev/devices files, find the entry for PMON, and the node number will be displayed. Then enter the `mknod` command. It may be necessary to remove the current /dev/mknod entry if it does not correspond to the /proc/devices id number as listed.

```
for example.
root@debian:~/ppctools/pmon# insmod pmon26.ko

root@debian:~/ppctools/pmon# cat /proc/devices

Character devices:

  1 mem

  4 /dev/vc/0

... intervening lines removed

171 ieee1394

180 usb

254 pmon


Block devices:

  1 ramdisk

  3 ide0

  8 sd

....remaining lines removed

root@debian:~/ppctools/pmon# mknod /dev/pmon c 254 0

root@debian:~/ppctools/pmon# chmod 777 /dev/pmon

root@debian:~/ppctools/pmon# ls -l /dev/pmon

crwxrwxrwx    1 root     root     254,   0 Jul 12 16:28 /dev/pmon
```

As can be seen from this example /proc/devices shows that the PMON device is assiged to id 254.

Further, for this example, align, to work, these conditions must also be met

1. align.c and pmon.c must be built
2. The resultant executable must be run.

5.2.5 Results When /dev/pmon is Available and pmon26.ko is Installed

Now that all these conditions have been met, lets run it again.

```
guest@debian:~/fae-training-04/library/align$ cat -n j

     1

     2  Alignment Test
```

```
 3  CPU = 7457,
 4  CPU 7457, has 6 PMCs
 5  Monitoring events are PMC[0]:1
 6  Monitoring events are PMC[1]:2
 7  Monitoring events are PMC[2]:1
 8  Monitoring events are PMC[3]:2
 9  Monitoring events are PMC[4]:0
10  Monitoring events are PMC[5]:0
11  {00,01,02,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f}
12  134610  Instructions,   110917 Cycles   0.823988 IPC
13  {00,01,02,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f}
14  540918  Instructions,   480033 Cycles   0.887441 IPC
guest@debian:~/fae-training-04/library/align$
```

Description of the output lines follow.

Previous line 11 is not printed, so /dev/pmon was found correctly. Lines 12 and 14 have values.

2 printed by line 117 in align.c

3 and 4 printed by line 50 and 62 in pmon.c

5 through 10 printed by line 91 in pmon.c

11 printed by line 126 in align.c

12 printed by line 127 in align.c

13 printed by line 138 in align.c

14 printed by line 139 in align.c

6 Using the PMON Facility

Since, some of the performance registers are privileged registers, only the Linux root user can change those. Therefore, it is necessary for a normal user to call a kernel support function to set these registers. Linux does not supply such a facility, however, the PMON facility included in the pegasos II system does contain such a facility, which was written by the Freescale CPD applications team. This facility, called PMON, is supplied as a kernel module in the root directory at /root/ppctools/pmon.

Since PMON is not a normally supplied module, the user is required to start and stop it. In addition, PMON uses the char device /dev/pmon for it’s operation. The user must therefore create this device.

Create the /dev/pmon device with the command:

```
mknod -c /dev/pmon c <pmon id> 0
```

To determine the PMON ID, see Section 5.2.4, “All These Conditions Must Be Met for the PMON Facility to Work.”

Instantiate the PMON facility by navigating to the /root/ppctools/pmon directory and performing this command:

```
insmod pmon26.ko
```

To stop this facility, use this command:

```
rmmod pmon26.ko
```

Remember to change back into a regular user after starting PMON.

To reiterate, in order to use the PMON facility, these steps must be performed.

1. The module, pmon26.ko must be built
2. The module, pmon26.ko must be installed, `insmod pmon26.ko`
3. /dev/pmon must be created, `mknod /dev/pmon c <node number> 0`
4. The permissions must be 777, `chmod 777 /dev/pmon`

The node number can be determined from the /proc/devices file. After the `insmod pmon26.ko`, look at the /dev/devices files, find the entry for PMON, and the node number will be displayed. Then enter the `mknod` command. It may be necessary to remove the /dev/mknod entry if it does not correspond to the /proc/devices id number as listed.

Further, for this example, align, to work, these conditions must also be met

1. align.c and pmon.c must be built
2. The resultant executable must be run.

See Freescale application note *PMON Module—An Example of Writing Kernel Module Code for Debian 2.6 on Genesi Pegasos II* (AN2744) for more information.

A normal user cannot do any of the above, thus the user is required to change to the root user.

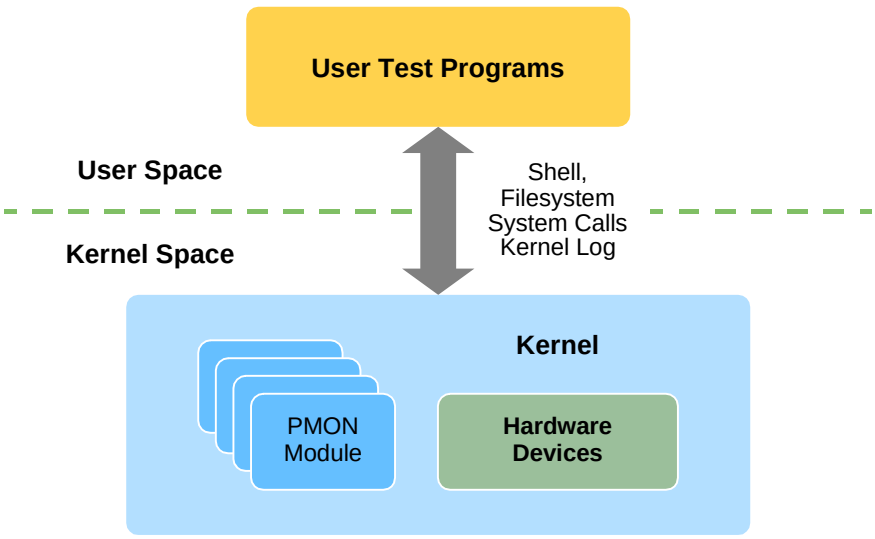


Figure 1. User Interaction with Kernel Module PMON

Figure 1 shows that the User program, which runs as a normal user, interfaces to the kernel via a call to the PMON module, which can in turn perform root activities for the user program.

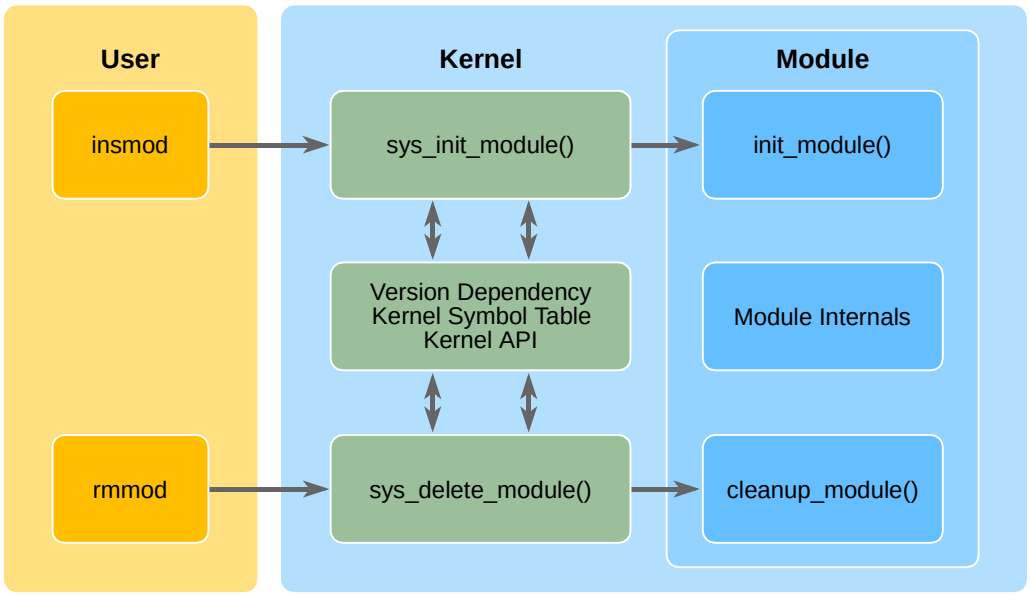


Figure 2. Instantiating the PMON Module

A user cannot instantiate a kernel module. Hence the module must be instantiated at boot time or by a root user with the `insmod` command. The `insmod` command, which means, instantiate module, will install the module in the kernel, then call the init_module of PMON, which will initialize itself and wait for user calls to the PMON module. The `rmmod` command will de-instantiate it, that is, call the cleanup_module to clean up any memory or other resources it is using and then remove it from the kernel module list.

Figure 2 shows the interaction of the user with the kernel, however, the user in this case must be the root user. The command `insmod` calls the kernel function sys_inti_module which adds the module to the kernels list and invokes the initialization function of the module. Modules are version dependent since once instantiated, they are part of the kernel and must have access to all the kernel symbols. The module internals are invoked by normal users making function calls to the device that is owned by the module. Finally, only the root user can remove a module with the `rmmod` command, which will call the modules clean up code and remove the module from the kernel list.

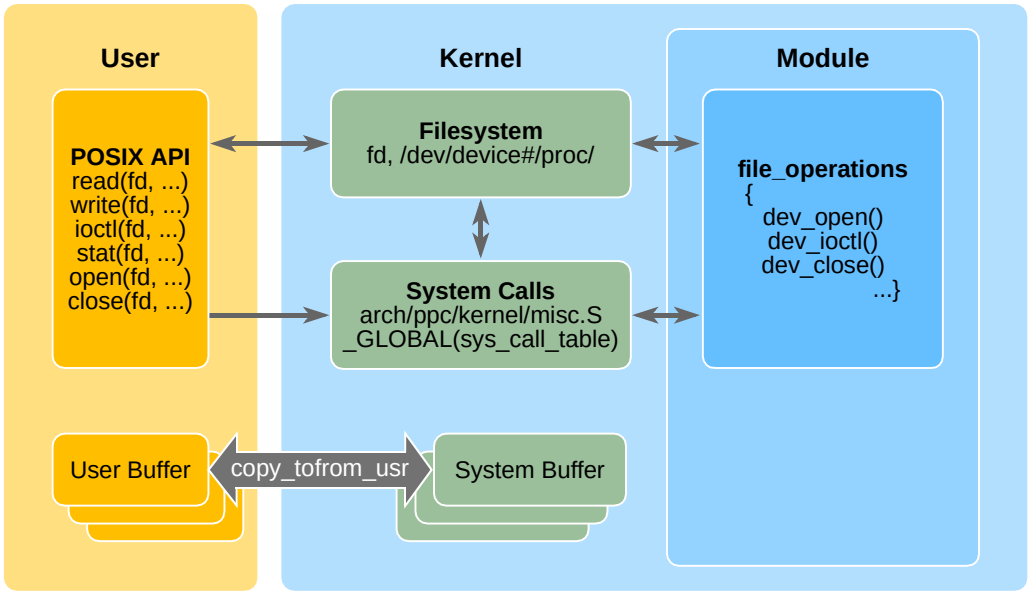


Figure 3. User Interaction to the Module

Figure 3 shows that once running, normal users can interface to the module with the standard POSIX API, using file commands, like open and read and write. This is because PMON instantiates itself as a char device and will get a device entry in /dev of /dev/pmon.

In summary, a root user must start PMON with the `insmod pmon26.ko` command. A shell script is available for this action, /root/ppcttools/pmon/install.sh If the system is rebooted, then PMON must be reinstalled.

7 More Advanced Examples

There are several more examples in the /guest/fae-training-04/library directory. We are going to discuss just one of them in detail, the Dot Product. Several other examples will be overviewed, the reader is encouraged to look at the other code. The mathlib directory has some AltiVec functions that can be used in your own programs.

7.1 Dot Product Example

In this example, the simg4plus facility is invoked with the -DTRACE=1 parameter as shown in the ,build.sh file. In order to use the PMON facility instead of simg4plus, use the Makefile, which does not define the TRACE macro.

True data dependency as well as some classical code optimization can often prevent vectorization. But in some cases the data dependency can be prevented and a gain in efficiency and speed can be obtained by using the AltiVec engine for vectorization. This example shows how to vectorize a classical data dependency problem.

Consider the dot product of two Matrixes, X and Y vectors of size N.

$$\text{Dot\_Product}(x[n], y[n]) = \sum_{i=1}^{n} x[i] * y[i]$$

The classic code solution:

```
float DotProduct( float *X, float *Y, int length ){
        int temp = 0;

        // N Iterations
        for( int i = 0; i < length; i++) {
            temp = X[i]*Y[i] + temp;
        }
        return temp;
}
```

This same function could be written in vector form, where each vector can contain 4 integers, thus v1 and v2 are size of N/4, thus we use four times fewer iterations. There is some set up time, but the dot product algorithm is operating four times faster.

```
float VectorDotProduct( vector float *v1, vector float *v2, int length ){
        vector float temp = (vector float) vec_splat_u32(0);
        float result;
        // Loop over the length of the vectors multiplying like terms and summing
        // Number of iterations is N/4
        for( int i = 0; i < length; i++)
                        temp = vec_madd( v1[i], v2[i], temp);
        // Still have four ints splat across a vector
        // Add across the vector
        temp = vec_add( temp, vec_sld( temp, temp, 4 ));     // Vector Shift Left Double
        temp = vec_add( temp, vec_sld( temp, temp, 8 ));
        vec_ste( temp, 0, &result );

        return result;
}
```

However, there is data dependency, only 1 madd can complete every 4 cycles.

```
float VectorDotProduct( vector float *v1, vector float *v2, int length ){
        vector float temp = (vector int) vec_splat_u32(0);
        float result;
        // Loop over the length of the vectors multiplying like terms and summing
        // Number of iterations is N/4
        for( int i = 0; i < length; i++)
                temp = vec_madd( v1[i], v2[i], temp); // true data dependency
                                                      // only 1 madd every 4 cycles

        temp = vec_add( temp, vec_sld( temp, temp, 4 ));      // Vector Shift Left Double
        temp = vec_add( temp, vec_sld( temp, temp, 8 ));
        vec_ste( temp, 0, &result );

        return result;
}
```

We can eliminate this dependency by performing 4 madd in a row, filling the pipeline, by doing 4 vectors at a time, incrementing our for loop by 4 each time, instead of once.

```
int FastVectorDotProduct( vector float *v1, vector float *v2, int length ){
   vector float temp = (vector float) vec_splat_s8(0);
   vector float temp2 = temp;    vector float temp3 = temp;
   vector float temp4 = temp;    vector float result;
   for( int i = 0; i < length; i += 4){                    //Loop over the length of the vectors,
      temp  = vec_madd( v1[i], v2[i], temp);               //this time doing 4 vectors in parallel
      temp2 = vec_madd( v1[i+1], v2[i+1], temp2);          // to fill the pipeline
      temp3 = vec_madd( v1[i+2], v2[i+2], temp3);
      temp4 = vec_madd( v1[i+3], v2[i+3], temp4);
   }
   //Sum our temp vectors
   temp       = vec_add( temp, temp2 );
   temp3      = vec_add( temp3, temp4 );
   temp       = vec_add( temp, temp3 );
   //Add across the vector
   temp       = vec_add( temp, vec_sld( temp, temp, 4 ));
   temp       = vec_add(temp, vec_sld( temp, temp, 8 ));
   //Copy the result to the stack so we can return it via the IPU
   vec_ste( temp, 0, &result );
   return result;
```

This code example, dot_product, procedes using these three methods, the classic method, the one madd at a time, and 4 madd at a time for one fourth the iterations. PMON is used to calculate the number of cycles and instructions used in each method. As we will see, the vector method is significantly better than the classic method, and the 4 madd at a time is again significantly better, i.e. more efficient and faster than either of the others.

7.1.1 Makefile

Line 2 in the Makefile below will compile and link our dot product example using the AltiVec intrinsics including our pmon.c interface to PMON, generating an elf executable whose name is test, because that is the name of the make target it line 1. Line 3 is the clean target, which invokes line 4 to remove all the objects and the elf file, test.

```
guest@debian:~/fae-training-04/library/dot_product$ cat -n Makefile
     1  test: dot_product.c pmon.c
     2          gcc -maltivec -mabi=altivec -O3 pmon.c dot_product.c -o  $@
```

```
     3  clean:
     4          rm -rf *.o test
guest@debian:~/fae-training-04/library/dot_product$
```

7.1.2 Code Listing and Explanation

The listing is the same as in the example in the Genesi Pegasos II directory and file, /home/guest/fae-training-04/library/dot_product/dot_product.c with the exception, that the printf statements at lines 177, 195, 197, 215, 217, and 235 have been changed to make the printing easier to discuss, and lines 184, 204, and 224 have been corrected to avoid the warning errors in the original.

```
/guest@debian:~/fae-training-04/library/dot_product$ cat -n dot_product.c
        1
     2  // Sergei Larin
     3  // Bitreversal example
     4  //
     5
     6  #include <altivec.h>
     7  #include <stdio.h>
     8
     9
    10  #define START_TIMER     \
    11     start_time      = read_744x_upmc1();\
    12     start_ins       = read_744x_upmc2();
    13
    14  #define STOP_TIMER \
    15     asm volatile("eieio"); \
    16     stop_time       = read_744x_upmc1(); \
    17     stop_ins        = read_744x_upmc2();
    18
    19  #if TRACE
    20  #define START_TRACING         asm (".long 0x14000001");
    21  #define STOP_TRACING          asm (".long 0x14000002");
    22  #define MAX_SIZE        64
```

```
23  #define REPEAT          1
24  #else
25  #define START_TRACING
26  #define STOP_TRACING
27  #define MAX_SIZE        4*1024
28  #define REPEAT          100
29  #endif
30
31  int start_pmon(int p1,int p2,int p3, int p4);
32  unsigned int read_744x_upmc1(void);
33  unsigned int read_744x_upmc2(void);
34  unsigned int read_744x_upmc3(void);
35  unsigned int read_744x_upmc4(void);
36
37  float aa[MAX_SIZE]      __attribute__ ((aligned (16)))  ;
38  float ab[MAX_SIZE]      __attribute__ ((aligned (16)))  ;
39
40  void print_int_vector(vector int *this_one){
41          printf("{%08x,%08x,%08x,%08x}\n",
42                  ((int *)this_one)[0],
43                  ((int *)this_one)[1],
44                  ((int *)this_one)[2],
45                  ((int *)this_one)[3]);
46  }
47
48  float dot_product(float *a,float *b, int num_elements){
49
50     int i;
51     float tmp    = 0;
52
53     for(i=0;i<num_elements;i++){
54             tmp += a[i] * b[i];
```

```
    55      }
    56      return tmp;
    57  }
    58
    59  float dot_p_vec_1(vector float *va,vector float *vb, int num_elements){
    60
    61      vector float temp = (vector float) vec_splat_u32(0);
    62      int i;
    63      float result;
    64
    65
    66      for(i = 0; i < num_elements/4; i++)
    67                      temp = vec_madd( va[i], vb[i], temp);
    68      temp = vec_add( temp, vec_sld( temp, temp, 4 ));     // Vector Shift Left
Double
    69      temp = vec_add( temp, vec_sld( temp, temp, 8 ));
    70      vec_ste( temp, 0, &result );
    71
    72      return result;
    73  }
    74
    75  float dot_p_vec_2(vector float *v1,vector float *v2, int num_elements){
    76
    77      vector float temp = (vector float) vec_splat_s8(0);
    78      vector float temp2 = temp;
    79      vector float temp3 = temp;
    80      vector float temp4 = temp;
    81      vector float *v1p    = v1;
    82      vector float *v2p    = v2;
    83      vector float t1,t2,t3,t4,t5,t6,t7,t8;
    84      float result;
    85      int i        = 0;
```

```
 86

 87     for(i = 0; i < num_elements/4; i += 4){                    //Loop over
the length of the vectors,

 88          temp  = vec_madd( v1[i], v2[i], temp);          //this time doing 4
vectors in parallel

 89         temp2 = vec_madd( v1[i+1], v2[i+1], temp2);     // to fill the pipeline

 90          temp3 = vec_madd( v1[i+2], v2[i+2], temp3);

 91          temp4 = vec_madd( v1[i+3], v2[i+3], temp4);

 92     }

 93  /*

 94     for(i = 0; i < num_elements/16; i++){                    //Loop over the
length of the vectors,

 95         temp  = vec_madd( *(v1p++), *(v2p++), temp);         //this time doing
4 vectors in parallel

 96        temp2 = vec_madd( *(v1p++), *(v2p++), temp2);     // to fill the pipeline

 97          temp3 = vec_madd( *(v1p++), *(v2p++), temp3);

 98          temp4 = vec_madd( *(v1p++), *(v2p++), temp4);

 99     }

100  */

101  /*

102     for(i = 0; i < num_elements/16; i++){                    //Loop over the
length of the vectors,

103          t1      = vec_ld(0,v1p);

104          t2      = vec_ld(0,v2p);

105          t3      = vec_ld(1,v1p);

106          t4      = vec_ld(1,v2p);

107         temp  = vec_madd( t1, t2, temp);          //this time doing 4 vectors
in parallel

108          t5      = vec_ld(2,v1p);

109          t6      = vec_ld(2,v2p);

110          temp2 = vec_madd( t3, t4, temp2);     // to fill the pipeline

111          t7      = vec_ld(3,v1p);

112          t8      = vec_ld(3,v2p);

113          temp3 = vec_madd( t5, t6, temp3);
```

```
114          v1p++;
115          v2p++;
116          temp4 = vec_madd( t7, t8, temp4);
117      }
118  */
119  /*
120      do{
121          temp  = vec_madd( v1[i], v2[i], temp);          //this time doing 4
vectors in parallel
122          temp2 = vec_madd( v1[i+1], v2[i+1], temp2);     // to fill the pipeline
123          temp3 = vec_madd( v1[i+2], v2[i+2], temp3);
124          temp4 = vec_madd( v1[i+3], v2[i+3], temp4);
125          i+=4;
126      }while(i<num_elements/4);
127  */
128  /*
129      while(1){
130          if(i>= num_elements/4)  break;
131          temp  = vec_madd( v1[i], v2[i], temp);          //this time doing 4
vectors in parallel
132          temp2 = vec_madd( v1[i+1], v2[i+1], temp2);     // to fill the pipeline
133          temp3 = vec_madd( v1[i+2], v2[i+2], temp3);
134          temp4 = vec_madd( v1[i+3], v2[i+3], temp4);
135          i+=4;
136      }
137  */
138      //Sum our temp vectors
139      temp          = vec_add( temp, temp2 );
140      temp3         = vec_add( temp3, temp4 );
141      temp  = vec_add( temp, temp3 );
142      //Add across the vector
143      temp  = vec_add( temp, vec_sld( temp, temp, 4 ));
144      temp          = vec_add(temp, vec_sld( temp, temp, 8 ));
```

```
145     //Copy the result to the stack so we can return it via the IPU
146     vec_ste( temp, 0, &result );
147     return result;
148  }
149
150  int main(){
151
152     int j,i;
153     int s1,s2,s3;
154     unsigned int    start_time, stop_time;
155     unsigned int    start_ins, stop_ins;
156     unsigned int    start_pc3,stop_pc3,start_pc4,stop_pc4;
157     float           result       = 0.0;
158
159     for(i=0;i<MAX_SIZE/2;i+=2){
160          aa[i]   = (float)i;
161          aa[i+1] = (float)-i;
162          ab[i]   = (float)i;
163          ab[i+1] = (float)i;
164     }
165
166  #if TRACE
167     START_TRACING;
168          //result  = dot_product(&aa,&ab,MAX_SIZE);
169          //result  = dot_p_vec_1(&aa,&ab,MAX_SIZE);
170          result  = dot_p_vec_2(&aa,&ab,MAX_SIZE);
171
172     STOP_TRACING;
173          return (int)result;
174  #else
175     start_pmon(1,2,1,2); // 1,2,1,15,  56,23
176
```

```
177     printf("Scalar function timing:\n \t");
178
179     start_pc4       = read_744x_upmc4();
180     start_pc3       = read_744x_upmc3();
181     START_TIMER;
182
183     for(i=0;i<REPEAT;i++)
184          result  = dot_product(aa,ab,MAX_SIZE);
185
186     STOP_TIMER;
187     stop_pc3       = read_744x_upmc3();
188     stop_pc4       = read_744x_upmc4();
189     printf("(%d),\t ins (%d),\t(%d)\t(%d)\n",
190          stop_time-start_time,
191          stop_ins-start_ins,
192          stop_pc3-start_pc3,
193          stop_pc4-start_pc4);
194
195     printf("   Output: (%f)\n======\n",result);
196
197     printf("Parallel version:      \n \t");
198
199     start_pc4       = read_744x_upmc4();
200     start_pc3       = read_744x_upmc3();
201     START_TIMER;
202
203     for(i=0;i<REPEAT;i++)
204        result  = dot_p_vec_1((vector float *) aa,(vector float *) ab,MAX_SIZE);
205
206     STOP_TIMER;
207     stop_pc3       = read_744x_upmc3();
208     stop_pc4       = read_744x_upmc4();
```

```
209     printf("(%d),\t ins (%d),\t(%d)\t(%d)\n",
210          stop_time-start_time,
211          stop_ins-start_ins,
212          stop_pc3-start_pc3,
213          stop_pc4-start_pc4);
214
215     printf("   Output: (%f)\n======\n",result);
216
217     printf("Parallel version 2:    \n \t");
218
219     start_pc4       = read_744x_upmc4();
220     start_pc3       = read_744x_upmc3();
221     START_TIMER;
222
223     for(i=0;i<REPEAT;i++)
224        result = dot_p_vec_2((vector float *) aa,(vector float *) ab, MAX_SIZE);
225
226     STOP_TIMER;
227     stop_pc3       = read_744x_upmc3();
228     stop_pc4       = read_744x_upmc4();
229     printf("(%d),\t ins (%d),\t(%d)\t(%d)\n",
230          stop_time-start_time,
231          stop_ins-start_ins,
232          stop_pc3-start_pc3,
233          stop_pc4-start_pc4);
234
235     printf("   Output: (%f)\n======\n",result);
236
237  #endif
238     return 0;
239  }
guest@debian:~/fae-training-04/library/dot_product$
```

Line explanation:

1 though 5 are comments

6 is the header to define the AltiVec intrinsics.

10 through 17 define macros for getting the start and stop time used in calculating the number of units used in a timing session, in this case cycles and instructions.

19 through 24 are used for simg4plus, which are not used here, since TRACE is not defined by the Makefile.

25 through 28 are used for PMON, however, 25 and 26 just shut off the simg4plus tracing facility.

31 through 35 are prototypes for the PMON functions, which are defined in pmon.c in this directory.

37 and 38 declare our float vectors, which are aligned to 16 bytes, a requirement for vector, AltiVec, operations.

40 through 46 is a function to print vectors, it is not used.

48 through57 is the scalar function to perform a dot product.

59 through 73 is a vectorization of the dot product algorithm, which can only perform 1 madd per 4 cycles, because of data dependency.

75 though 148 is the same vectorization, however, it can perform 4 madd per 4 cycles, i.e. 1 madd per cycle, by filling the pipe with four madd in a row. 93 through 137 are all commented out and therefore ignored. They do not participate in this algorithm.

150 through 157 is the beginning of the main function and the declaration of variables.

159 through 164 initialize the two arrays to values that will product a result of 0 in the dot product, no matter how many elements are in the array, as long as there are an even number of elements. This is described in Section 7.1.3, “Results and Explanation.”

166 through 173 are ignored because the macro TRACE is undefined.

175 initializes PMON to count cycles and instructions.

177 through 195 performs the scalar dot product algorithm many times and counts the cycles and instructions.

197 through 215 performs the 1 madd per 4 cycles vectorization algorithm many times and counts the cycles and instructions.

217 through 235 performs the 4 madd per 4 cycles vectorization algorithm many times and counts the cycles and instructions.

7.1.3 Results and Explanation

The following commands, will clean our directory, make the elf executable, test, and execute it with the `./test` command, since the local directory in not part of the PATH variable, finally, the result is stored in the file, j so that we can use the `cat -n` command to get line numbers for the code. The line numbers are not part of the output, and are used for this explanation. Of course, the code is supplied in the /home/guest/fae-training-04/library/dot_product/dot_product.c.

The two arrays, aa and ab, which are vectors, a special case of a matrix, which has one row, are filled with values that will result in a value of zero when applied to a dot vector.

aa[0] = 0, aa[1] =0, ab[0] = 0, ab[1] = 0

aa[2] = 2, aa[3] = -2, ab[2] = 2, ab[3] = 2

etc.

The sum of (aa[i] *ab[i]) + (aa[i+1] * ab[i+1]) = 0. Therefore all the sums of these alternate products will be zero for this example. We are not interested in the result of the dot product, only in the speed at which it completes. The code also repeats the operations many times, just to get enough cycles to make a comparison.

```
guest@debian:~/fae-training-04/library/dot_product$ make clean
rm -rf *.o test
guest@debian:~/fae-training-04/library/dot_product$ make
gcc -maltivec -mabi=altivec -O3 pmon.c dot_product.c -o  test
guest@debian:~/fae-training-04/library/dot_product$ ./test > j
guest@debian:~/fae-training-04/library/dot_product$ cat -n j
     1  CPU = 7457,
     2  CPU 7457, has 6 PMCs
     3  Monitoring events are PMC[0]:1
     4  Monitoring events are PMC[1]:2
     5  Monitoring events are PMC[2]:1
     6  Monitoring events are PMC[3]:2
     7  Monitoring events are PMC[4]:0
     8  Monitoring events are PMC[5]:0
     9  Scalar function timing:
    10          (2130188),       ins (2461537), (2130218)       (2461558)
    11     Output: (0.000000)
    12  ======
    13  Parallel version:
    14          (420495),        ins (618164),  (420513)        (618186)
    15     Output: (0.000000)
    16  ======
    17  Parallel version 2:
    18          (275311),        ins (439212),  (275329)        (439234)
    19     Output: (0.000000)
    20  ======
guest@debian:~/fae-training-04/library/dot_product$
```

Line explainations:

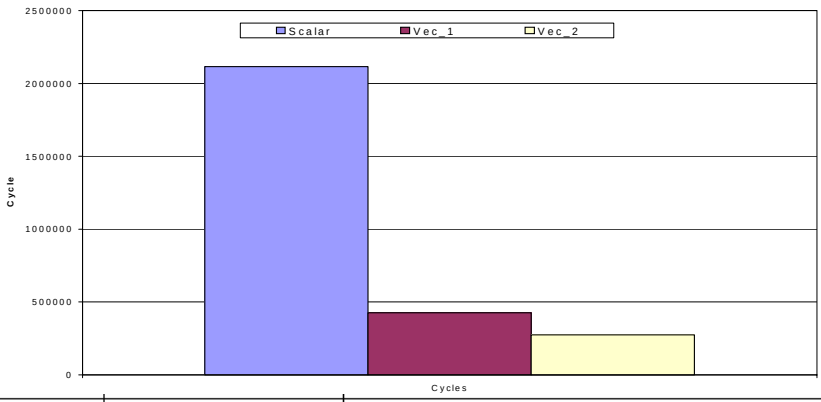
1 through 8 are the same as in Section 3.6, “An AltiVec Program Demonstrating the Use of PMON for Obtaining Performance Statistics” and in fact, we are using the same performance monitor counters, cycles and instructions.

9 through 12 is the result of the scalar dot product which took 2,130,188 cycles and 2,461,537 instructions.

13 through 16 is the result of the vectorization using only 1 madd per 4 cycles which took 420,495 cycles and 618,164 instructions. Thus this code is 5 times faster (2130188/420495) than the scalar case.

17 through 20 is the result of the vectorization using 4 madd per 4 cycles, i.e. 1 madd per cycle, which took 275,311 cycles and 439,212 instructions. Thus this code is 1.5 times faster (420495/275311) than the previous case and this code is 7.7 times faster (2130188/275311) than the scalar case.

Advanced example of PMON usage



15 (000_1111)	Cycles a VFPU instruction	Counts the cycles an AltiVec instruction in the VFPU reservation station is waiting for an operand.
32 (010_0000)	Cycles where no instructions completed	Counts the cycles where no instructions are completed.

	Cycles	Instructions	IPC	Speedup	VFPU wait	No Ins Completed
Scalar	2117088	2462236	1.16303	1	0	1645357
Vec_1	427256	617450	1.445152	4.955081	311620	219965
Vec_2	275730	439212	1.592906	7.67812	54147	56836

Figure 4. Comparison of Scalar Versus Vector Computations

Figure 4., “Comparison of Scalar Versus Vector Computations” graphically illustrates the value of vectorization.

7.2 Branching

The /home/guest/fae-training-04/library/branches is an example showing the increased efficiency of eliminating branches where possible. Again, it uses PMON to find the cycles and instructions used for various methods which eliminate branches. Rather than a detailed look at the code, this section will just present an overview of the code, which can then be looked at for the details.

The processor can not proceed at full speed unless it knows where it is going. Branches can not always be predicted correctly. The processor therefore guesses which branch will be taken and if wrong, must back track to the condition and then go off in the right direction.

- There are some general guidelines on how the processor will guess

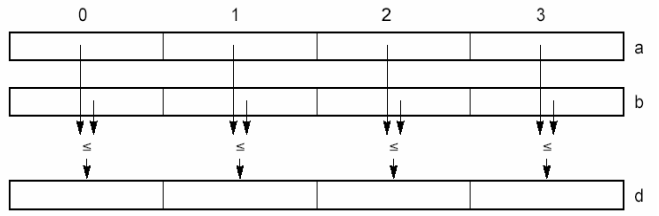
— Static branch prediction: Forward branch – not taken, backward branch is taken. Which means in an if-then-else decision the most likely section is “then”

- Dynamic branch prediction – after one or two invocations of the same branch instructions enough history is accumulated to make good predictions next time around.

However, branch prediction is vulnerable to aliasing. Try to avoid branches even if it means more computations – it is likely to be faster!

– Assuming it is used to compare two arrays

```
int Max( int a, int b ) {
   int result;
   if( a < b )    result = b;
   else           result = a;
   return result;
}
```




```
//Return the maximum of two vector integers:    result = (a & ~mask) | (b & mask);
vector signed int Max(vector signed int a, vector signed int b ){
    vector bool     int mask = vec_cmplt ( a, b );              //If ( a < b)...
    vector signed int result = vec_sel( a, b, mask );           //Select a or b
    return result;
}
```

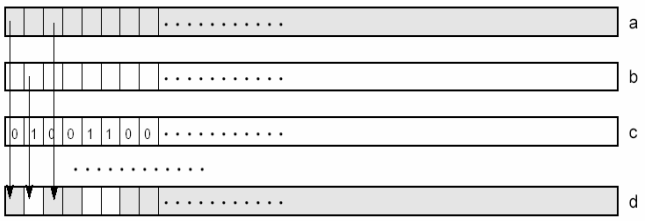


Figure 5. Scalar and Vector Method of Finding the Maximum of Two Numbers

Figure 5 is an example of finding the maximum of two numbers, which uses a simple comparison to determine which number is larger.

The code used for this example compares three methods for determining the maximum of two numbers. The major assumption is that eliminating the need for branches will improve our performance The fact that we process 4 times the data per iteration with AltiVec is a second order effect The difference between the two functions, Max and Max_p is mere semantics, even though meaning exactly the same thing it does cause different compiler analysis (for some compilers) resulting in a different code We used – GCC V3.3.3 with an optimization level of -O2. Max_vec is using AltiVec vector code.

```
guest@debian:~/fae-training-04/library/branches$ cat Makefile

test: branches.c pmon.c

        gcc -maltivec -mabi=altivec -O2 pmon.c branches.c -o  $@

clean:

        rm -rf *.o pmon_test
```

```
guest@debian:~/fae-training-04/library/branches$
```

The relevant code is shown here.

```
inline void Max( int *a, int *b, int *c, int elements) {
  int i;
  for(i=0;i<elements;i++){
     if(a[i]>b[i])   c[i] = a[i];
     else            c[i] = b[i];
  }
}

inline void Max_p( int *a, int *b, int *c, int elements) {
  int i;
  for(i=0;i<elements;i++)  ac[i] = (aa[i]>ab[i])?aa[i]:ab[i];
}

inline void Max_vec(int *a,int *b,int *c, int elements){
  vector int va,vb,vc;
  vector int *pva = (vector int *) a;
  vector int *pvb = (vector int *) b;
  vector int *pvc = (vector int *) c;
  vector bool int      mask;
  int         i;
  for (i=0; i<elements/4; i++){
     va      = vec_ld (0,pva++);
     vb      = vec_ld (0,pvb++);
     mask  = vec_cmplt (va,vb);
     vc      = vec_sel( va, vb, mask );
     vec_st(vc,0,pvc++);
  }
}
```

The code will run two different sets of data. One in a predictable fashion, the other in a random fashion.This arrangement should expose branch predictor behavior

```
#define ELEMENTS       4*1024

int    aa[ELEMENTS]    __attribute__ ((aligned (16))); // Input data set
int    ab[ELEMENTS]    __attribute__ ((aligned (16))); // Input data set
int    ac[ELEMENTS]    __attribute__ ((aligned (16))); // Output data set

 for(i=0;i<ELEMENTS;i++){
#if RANDOM
            aa[i]   = rand();
            ab[i]   = rand();
#else
            aa[i]   = i;
            ab[i]   = ELEMENTS - i;
#endif
            ac[i]   = 0;
}
```

We are using PMON to collect not only the cycles and instructions used, but also event 15, the number of cycles an AltiVec instruction in the VFPU reservation station is waiting for an operand, and event 26, the true branch target instruction hits for taken branches. The call to start_pmon determines which counters we are going to use. The performance monitor events are described in table 11-9 of *MPC7450 RISC Microprocessor Family User's Manual*.

```
start_pmon(1,2,26,15); // 1,2,26,15
```

The results of running this code dramatically shows the case for using the vector method which substitutes masking and vector operations for decision branching.

```
guest@debian:~/fae-training-04/library/branches$ ./test > j

guest@debian:~/fae-training-04/library/branches$ cat -n j

     1  CPU = 7457,

     2  CPU 7457, has 6 PMCs

     3  Monitoring events are PMC[0]:1

     4  Monitoring events are PMC[1]:2

     5  Monitoring events are PMC[2]:26

     6  Monitoring events are PMC[3]:15

     7  Monitoring events are PMC[4]:0

     8  Monitoring events are PMC[5]:0

     9  Scalar function timing:         (178432),        ins (106480),  (8219) (46771)

    10  Scalar predicate timing:       (379000),       ins (147392),  (4162) (267956)

    11  Vector timing:                  (22210),         ins (9364),    (0)    (17543)
```

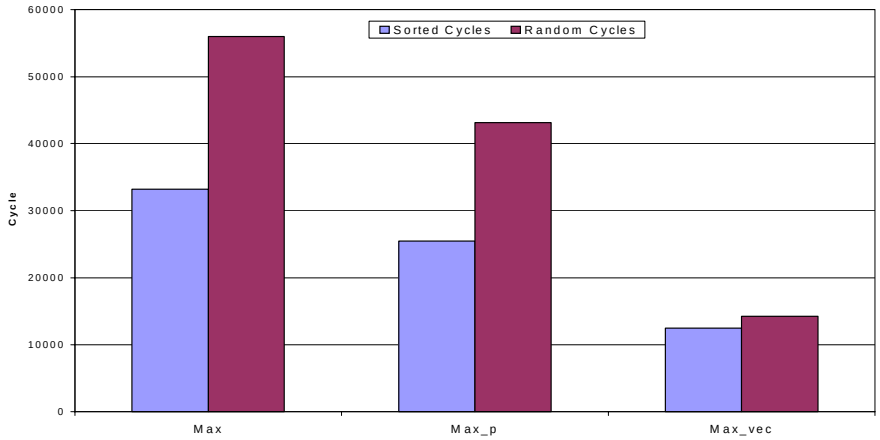
Line description:

1 through 9 is our all familiar introduction to these programs, describing the CPU type and the events to count, in this case, 1, 2, 26, and 15, cycles, instructions, branch target hits, and cycles for VFPU instructions. The performance monitor events are described in table 11-9 of *MPC7450 RISC Microprocessor Family User's Manual*.

9 shows the counters for a scalar branch

10 shows the counters for map_p

11 shows the counters for a vector solution. Clearly, the vector solution is superior. The next diagram graphically shows this conclusion.



Sorted Values						pmc1_4	pmc1_15			pmc2_26
	Cycles	Retired Instr	IPC	Ins/Iter	Speedup	Dispatched	Dispatch_0	Junk	Junk%	Br_Flush
Max	33174	30474	0.918611	7.4399414	1	30501	11423	27	0.000885	4
Max_p	25486	31773	1.246684	7.7570801	1.301656	31799	7096	26	0.000818	2
Max_vec	12508	9364	0.748641	2.2861328	2.652223	9385	8044	21	0.002238	0

Random Values						pmc1_4	pmc1_15			pmc2_26
	Cycles	Retired Instr	IPC	Ins/Iter	Speedup	Dispatched	Dispatch_0	Junk	Junk%	Br_Flush
Max	56000	30920	0.552143	7.5488281	1	53315	23751	22395	0.420051	2049
Max_p	43124	32410	0.751554	7.9125977	1.298581	58151	7053	25741	0.442658	2056
Max_vec	14238	9364	0.657677	2.2861328	3.933137	9388	9539	24	0.002556	0

7.3 Bit Reversal, Eliminating Computations.

This example, /home/guest/fae-training-04/library/bitreverse is an example of calculating the reverse of a bit pattern, or looking it up in a large table, or in a small table. PMON is used to collect just the cycles and instructions to determine which is more efficient, i.e. uses less cycles.

The technique of data slicing can be used in AltiVec to eliminate computation.

The first function is for a Byte-wise Bit Reversal Algorithm.

```
unsigned char reverse (unsigned char in){
  unsigned char out =  ((in & 0x01)<<7) |
                       ((in & 0x02) <<5) |
                       ((in & 0x04) <<3) |
                       ((in & 0x08) <<1) |
                       ((in & 0x10) >>1) |
                       ((in & 0x20) >>3) |
                       ((in & 0x40) >>5) |
                       ((in & 0x80) >>7);
  return out;
}
```

This straightforward method yields 0.10 Bytes/Cycle

An alternative implementation is a Big Lookup Table:

- 256 entry byte table holding the “reversed” values
- So, the computation for each byte is converted into a single “load”

```
reversed[j] = big_lookup[j];
```

```
unsigned char big_lookup[256] = {
  0x00,0x80,0x40,0xc0,0x20,0xa0,0x60,0xe0,0x10,0x90,0x50,0xd0,0x30,0xb0,0x70,0xf0,
  0x08,0x88,0x48,0xc8,0x28,0xa8,0x68,0xe8,0x18,0x98,0x58,0xd8,0x38,0xb8,0x78,0xf8,
  0x04,0x84,0x44,0xc4,0x24,0xa4,0x64,0xe4,0x14,0x94,0x54,0xd4,0x34,0xb4,0x74,0xf4,
  0x0c,0x8c,0x4c,0xcc,0x2c, 0xac,0x6c,0xec,0x1c,0x9c,0x5c,0xdc,0x3c,0xbc,0x7c,0xfc,
  0x02,0x82,0x42,0xc2,0x22,0xa2,0x62,0xe2,0x12,0x92,0x52,0xd2,0x32,0xb2,0x72,0xf2,
  0x0a,0x8a,0x4a,0xca,0x2a,0xaa,0x6a,0xea,0x1a,0x9a,0x5a,0xda,0x3a,0xba,0x7a,0xfa,
  0x06,0x86,0x46,0xc6,0x26,0xa6,0x66,0xe6,0x16,0x96,0x56,0xd6,0x36,0xb6,0x76,0xf6,
  0x0e,0x8e,0x4e,0xce,0x2e,0xae,0x6e,0xee,0x1e,0x9e,0x5e,0xde,0x3e,0xbe,0x7e,0xfe,
  0x01,0x81,0x41,0xc1,0x21,0xa1,0x61,0xe1,0x11,0x91,0x51,0xd1,0x31,0xb1,0x71,0xf1,
  0x09,0x89,0x49,0xc9,0x29,0xa9,0x69,0xe9,0x19,0x99,0x59,0xd9,0x39,0xb9,0x79,0xf9,
  0x05,0x85,0x45,0xc5,0x25,0xa5,0x65,0xe5,0x15,0x95,0x55,0xd5,0x35,0xb5,0x75,0xf5,
  0x0d,0x8d,0x4d,0xcd,0x2d,0xad,0x6d,0xed,0x1d,0x9d,0x5d,0xdd,0x3d,0xbd,0x7d,0xfd,
  0x03,0x83,0x43,0xc3,0x23,0xa3,0x63,0xe3,0x13,0x93,0x53,0xd3,0x33,0xb3,0x73,0xf3,
  0x0b,0x8b,0x4b,0xcb,0x2b,0xab,0x6b,0xeb,0x1b,0x9b,0x5b,0xdb,0x3b,0xbb,0x7b,0xfb,
  0x07,0x87,0x47,0xc7,0x27, 0xa7,0x67,0xe7,0x17,0x97,0x57,0xd7,0x37,0xb7,0x77,0xf7,
  0x0f, 0x8f,0x4f, 0xcf, 0x2f, 0xaf, 0x6f, 0xef, 0x1f, 0x9f, 0x5f, 0xdf,0x3f,0xbf,0x7f,0xff
};
```

This method yields 0.19 Bytes/Cycle or 2x faster then the original

Another method is using a Small Lookup Table

- based on splitting each byte into two nibbles
- looking up values for both of them independently, and merging result later

```
unsigned char small_lookup_l[16] __attribute__ ((aligned (16))) = {
  0x00,0x08,0x04,0x0c,0x02,0x0a,0x06,0x0e,0x01,0x09,0x05,0x0d,0x03,0x0b,0x07,0x0f
};

unsigned char small_lookup_h[16] __attribute__ ((aligned (16))) = {
  0x00,0x80,0x40,0xc0,0x20,0xa0,0x60,0xe0,0x10,0x90,0x50,0xd0,0x30,0xb0,0x70,0xf0
};


reversed[j] = small_lookup_l[(j&0xf0)>>4] | small_lookup_h[(j&0x0f)];
```

This method uses less memory, but runs a bit slower: yielding 0.11 Bytes/Cycle

The true advantages comes from observation that small tables will fit into two AltiVec registers and all lookups are completely independent, so 16 of them could be performed in parallel.

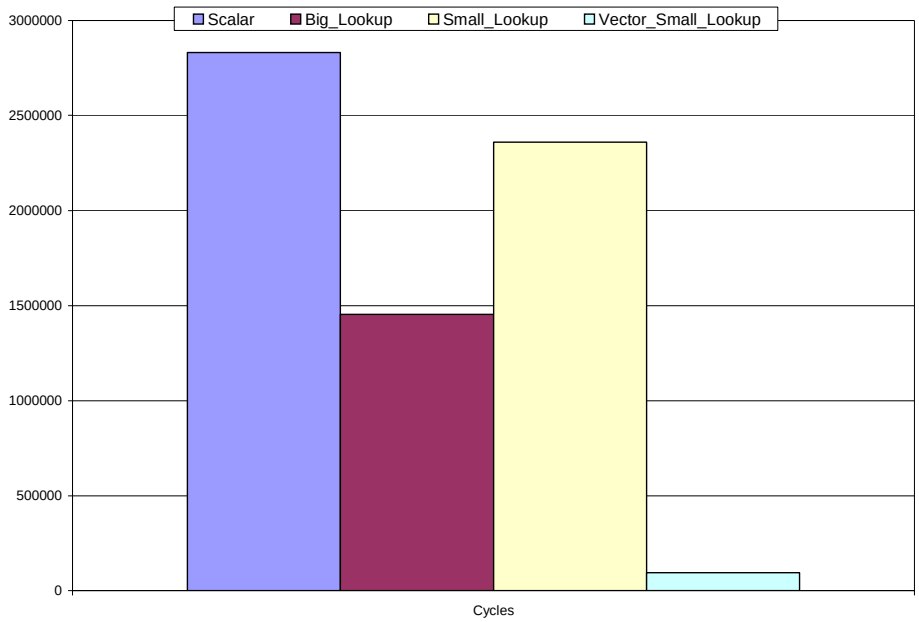
```
void reverse_vector(vector unsigned char *in,vector unsigned char *out, int num_elements){
  int i;
  vector unsigned char st_l, st_h;
  vector unsigned char four =  vec_splat_u8(4);
  vector unsigned char v_in,vl,vh, v_out;

  st_l = vec_ld (0,(vector unsigned char *) small_lookup_l);
  st_h = vec_ld (0,(vector unsigned char *) small_lookup_h);

  for(i=0; i<num_elements; i+=16){
      v_in    = vec_ld (i,in);
      vh      = vec_sr(v_in,four); // vl = vec_sl(v_in,four);
                                   // vl = vec_sr(vl,four);
      vh      = vec_perm(st_l,st_l,vh);
      vl      = vec_perm(st_h,st_h,v_in); // vl
      v_out  = vec_or(vh,vl);
      vec_st(v_out,i,out);
  }
```

This method for the same conditions yields 2.7 Bytes/Cycle. It is 30x faster then the original Scalar and 15x faster then the BigLookupTable

Compiling and running this program generates this output.

```
guest@debian:~/fae-training-04/library/bitreverse$ ./test > j

guest@debian:~/fae-training-04/library/bitreverse$ cat -n j

     1  CPU = 7457,

     2  CPU 7457, has 6 PMCs

     3  Monitoring events are PMC[0]:1

     4  Monitoring events are PMC[1]:2

     5  Monitoring events are PMC[2]:1

     6  Monitoring events are PMC[3]:2

     7  Monitoring events are PMC[4]:0

     8  Monitoring events are PMC[5]:0

     9  Scalar function timing:        (2844410),      ins (4872558), (2844465)
(4872576)

    10  Big Lookup:                   (1455586),      ins (1286521), (1455615)
(1286546)

    11  Small Lookup:                 (1793101),      ins (2309580), (1793279)
(2309602)

    12  Vector timing:                 (95939),        ins (156704),  (96188)(156732)

guest@debian:~/fae-training-04/library/bitreverse$
```

Line 9 uses 2,844,410 cycles versus line 10 uses 1455586 versus line 11, the vector solution, uses 95939 cycles. That is 2844410/95939 = 29.65, almost 30 times faster.



	Cycles	Instructions	IPC	Bytes/Cycle	Speedup	L1D_Hits	L1D_Miss	L1D_Access
Scalar	2830757	4871217	1.7208178	0.090435173	1	541155	1922	543077
Big_Lookup	1454818	1285741	0.8837813	0.175967028	1.945781	512332	30	512362
Small_Lookup	2360056	2403152	1.0182606	0.108472002	1.199445	768249	6	768255
Vector_Small_Lookup	95390	156143	1.6368907	2.683719467	29.67562	37047	34	37081

7.4 Constant Generations

/home/guest/fae-training-04/library/constant_gen is an example of generating constant data in vectors. The first case uses compiler generated code in a declaration, which generates code to replicate the constants, the other uses the vec_splat, splat, AltiVec instruction to replicate the same constant into an array.

```
vector unsigned char add_constants_1 (){
  vector unsigned char vec_a = { 5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5 };
  vector unsigned char vec_b = { 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7 };
  vector unsigned char vec_c = { 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1 };

  vec_c = vec_add(vec_a,vec_c);
  vec_c = vec_add(vec_b,vec_c);

  return vec_c;

}

vector unsigned char add_constants_2 (){
  vector unsigned char vec_a = vec_splat_u8(5);
  vector unsigned char vec_b = vec_splat_u8(7);
  vector unsigned char vec_c = vec_splat_u8(1);

  vec_c = vec_add(vec_a,vec_c);
  vec_c = vec_add(vec_b,vec_c);

  return vec_c;
}
```

In this case we are using PMON to collect our usual cycles and instructions, but also to collect event 64, AltiVec load instructions completed, and 41, L1 instruction cache accesses. The performance monitor events are described in table 11-9 of *MPC7450 RISC Microprocessor Family User's Manual*.

```
guest@debian:~/fae-training-04/library/constant_gen$ ./test > j

guest@debian:~/fae-training-04/library/constant_gen$ cat -n j

     1  CPU = 7457,

     2  CPU 7457, has 6 PMCs

     3  Monitoring events are PMC[0]:64

     4  Monitoring events are PMC[1]:41

     5  Monitoring events are PMC[2]:1

     6  Monitoring events are PMC[3]:2

     7  Monitoring events are PMC[4]:0

     8  Monitoring events are PMC[5]:0

     9  V1 function timing:     (3),     ins (9),       (20871) (10057)

    10  {0d,0d,0d,0d,0d,0d,0d,0d,0d,0d,0d,0d,0d,0d,0d,0d}

    11  V2 function timing:     (0),     ins (5),       (20302) (10054)

    12  {0d,0d,0d,0d,0d,0d,0d,0d,0d,0d,0d,0d,0d,0d,0d,0d}

guest@debian:~/fae-training-04/library/constant_gen$
```

Line 9, compiler declaration constants, takes 20871 cycles and line 11, AltiVec vector splat takes 20302 cycles, not a dramatic difference, but a little bit better. For large amounts of constant data, the vector splat could save significant amounts of computation time. Also, notice that declaration used 3 AltiVec load instruction, while splat took zero, and declaration used 9 L1 cache accesses, while splat used 5.

7.5 Step Back and Take a 10,000 Foot View

There is a logical sequence to be observed in implementation of these methods. One can look at the optimization process as moving the bottleneck around the processor

- if computation takes longer then anything else – speed them up
- if system bus is under utilized – use prefetching
- if bus is 100% full, computations are at the minimum… reduce the code and data size?
- But the truly superior goal is to reach computational entropy –
  - get rid of all the unnecessary computations through algorithm modifications
  - balance added memory bandwidth with real data I/O
  - use predictability of the data streams to the full extent

Concentrate your effort, in large applications work with 10% of the code which accounts for 90% of the execution time

8 Conclusion

- This application note has presented the information needed to use some AltiVec constructs, understand alignment, and use the PMON monitoring facility of the Performance Monitor Registers on the MPC74xx processors. Two detailed examples were presented, align.c and dot_product.c, which show how PMON can be used to determine which code is faster. Several other examples were overviewed. AltiVec Technology transparently adds SIMD functionality to a high speed RISC engine
- AltiVec enables a broad range of embedded and computing applications
- C level programming offers certain level of comfort while providing powerful way to extract parallelism from applications
- You must think in terms of Vector Processing throughout the design cycle of an application
- AltiVec is not pixie dust, to be sprinkled on an existing code, it takes foresight and design.
- With these techniques a 4x to 30x or more speedup is possible

AltiVec coding can speed up many common applications. CPD Applications has some AltiVec library applications that are available.

The items from the following categories are available at http://www.freescale.com/altivec:

- Telecomm
  - FFT/IFFT, FIR, Autocorrelation, Convolution Encoder/Viterbi Decoder (GSM)
- MultiMedia
  - DCT/IDCT, JPEG 2000, Quantization/Dequantization, SAD
- Networking
  - QOS, NAT, Route Lookup, IP Reassembly, TCP/IP,
  - Encryption (SHA)
- LibC (means could be “Linked” at compilation)
  - Link level support for standard C functions (memcpy, strcmp etc.)
- Mathematical primitives (Extension of LibC+)
  - Math.h - Log, Exp, Sin, Cos, Sqrt
- OS enablement
  - Linux (TCP/IP),

The items from these categories are available upon request:

- Telecomm
  - Convolution Encoder/Viterbi Decoder (3G), Error Correction Codes (CRC 8,12,16,24)
- MultiMedia
  - MPEG2, MP3, JPEG
- Printer
  - GhostScript Library elements, Color Conversion, FS Dithering
- Networking
  - Encryption (AES, DES, 3DES, MD5, Kasumi)
- OS enablement
  - VxWorks elements

These applications will be available soon:

- MultiMedia
  - MPEG4,
- Printer
  - Scaling/Rotation
- Networking
  - OSPF
  - Encryption (RSA)
  - Wireless network (802.11), LZO
- Mathematical primitives (Extension of LibC+)
  - Matrix math, LargeNumber Lib

For assistance or answers to any question on the information that is presented in this document, send an e-mail to risc10@freescale.com.

9 References

The following documents describe the various applications of the Genesi Pegasos II system.

1. Freescale application note AN2666, *Genesi Pegasos II Setup*
2. Freescale application note AN2736, *Genesi Pegasos II Boot Options*
3. Freescale application note AN2738, *Genesi Pegasos II Firmware*
4. Freescale application note AN2739, *Genesi Pegasos II Debian Linux*
5. Freescale application note AN2744, *PMON Module, an Example of Writing Kernel Module Code for Debian 2.6 on Genesi Pegasos II*
6. *AltiVec Technology Programming Environments Manual* (ALTIVECPEM/D Rev. 1)
7. *AltiVec Programming Interface Manual* (ALTIVECPIM/D Rev. 1)
8. *MPC7450 RISC Microprocessor Family User's Manual*

10 Revision History

Table 1 provides a revision history for this application note.

Table 1. Document Revision History

Rev. No.	Date	Substantive Change(s)
0	07/23/2004	Initial release.
0.1	08/31/2004	Minor editing.

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

How to Reach Us:

USA/Europe/Locations Not Listed:
Freescale Literature Distribution
P.O. Box 5405,
Denver, Colorado 80217
1-480-768-2130
(800)-521-6274

Japan:
Freescale Semiconductor Japan Ltd.
SPS, 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

AN2743
Rev. 0.1
08/2004