Notes on Using GCC IWMMXT Intrinsic

Table of Contents

List of Figures

List of Tables

1. Overview

The major benefit of using intrinsic is that you now have access to key features that are not available using conventional coding practices. Intrinsic enable you to code with the syntax of C function calls and variables instead of assembly language. Most Wireless MMX instructions have a corresponding C intrinsic that implements that instruction directly. This frees you from managing registers and enables the compiler to optimize the instruction scheduling [1].

This document is not intent to serve as the intrinsic function manual, but as a programming reference instead. We are trying to give some notes on how to write an efficient program with IWMMXT intrinsic. If you are looking for the full list of intrinsic functions and its usage syntax, please refer to the manuals in [1] or [2].

In the following sections, we will use invert DCT function in jpeg decoder [3] as the example to explain related techniques and their impact on the performance. Section 2 describes how to use IWMMXT intrinsics in C sources; Section 3 outlines related compiling options and debugging commands. Some other miscellaneous notes are listed in Section 4.

2. Using GCC IWMMXT intrinsics in C sources

2.1. Include files

To use Intel Wireless MMX technology intrinsics, the ***<mmintrin.h>*** file must be included. This file contains __m64 data type definitions and ANSI C prototypes for the Intel Wireless MMX technology intrinsic functions [2].

The syntax of Intel Wireless MMX technology intrinsic prototype is as follows:

Syntax

```
#include <mmintrin.h>
data_type   intrinsic_name (parameters);
```

- *data_type* is the return data type, which is usually void, int, or __m64. Intrinsics may return other data types that are described in the intrinsic syntax definitions.
- *intrinsic_name* is the name of the intrinsic, which behaves like a function that you can use in your C/C++ code instead of inlining the actual instruction.
- *parameters* represents the parameters required by each intrinsic.

2.2. Data types and Alignments

2.2.1. Data types

Marvell GCC provides two new 64-bit types for IWMMXT intrinsics; they are type defined as below:

```
typedef unsigned long long __m64;
typedef long long __int64;
```

Attention:
Previous version of GCCs (Official versions including CodeSourcery's) treat __int64 as **unsigned long long** which is different from our definition.

The __m64 data type can hold eight 8-bit values, four 16-bit values, two 32-bit values, or one 64-bit value. But since the new data type __m64 is not one of the basic ANSI C data types, you must observe the following usage restrictions:

- ✧ The __m64 data type can be used **only** on either side of an assignment, as parameters to a function call, and as a return value from a function call. **You cannot use it with other arithmetic expressions ("+", "-", etc.).**
- ✧ The __m64 data type can be used as objects in aggregates (such as unions) to access the byte elements and structures.

Conversions between __m64 and other types are supported by following intrinsics in Table 1.

Table 1 Data type conversion intrinsics

Intrinsics prototype		Note
__m64	_mm_cvtsi64_m64 (__int64 __i)	Convert from __int64 to __m64
__int64	_mm_cvtm64_si64 (__m64 __i)	Convert from __m64 to __int64
int	_mm_cvtsi64_si32 (__int64 __i)	Convert from __int64 to int
__int64	_mm_cvtsi32_si64 (int __i)	Convert from int to __int64

2.2.2. Alignments

As shown in general optimization guidelines, aligned data are preferred for SIMD computing. We can do it by specifying attributes for variables (GCC C Extensions).

For example, the following codes declare a 16-bytes aligned array:

```
int a[4] __attribute__((aligned(16))) = { 1, 3, 5, 7 };
```

You can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the default alignment for the target architecture

you are compiling for. The default alignment is sufficient for all scalar types, but may not be enough for all vector types on a target which supports vector operations. The default alignment is fixed for a particular target ABI. Please refer to ABIs' documents for details.

For example, Table 2 shows the default data type and alignment for AAPCS ABI [6].

GCC also provides a target specific macro __BIGGEST_ALIGNMENT__, which is the largest alignment ever used for any data type on the target machine you are compiling for. For example, you could write:

```
short array[3] __attribute__ ((aligned (__BIGGEST_ALIGNMENT__)));
```

This macro is also ABI dependent: for old ARM ABI (APCS and ATPCS), its value is 32, for the others (AAPCS, IWMMXT and AAPCS_LINUX), its value is 64.

Table 2 AAPCS ABI's default data type and alignment

Type Class	**Machine Type**	**Byte size**	**Byte alignment**	**Note**
Integral	Unsigned byte	1	1	Character
	Signed byte	1	1	
	Unsigned half-word	2	2	
	Signed half-word	2	2	
	Unsigned word	4	4	
	Signed word	4	4	
	Unsigned double-word	8	8	
	Signed double-word	8	8	
Floating Point	Half precision	2	2	Half-precision Floating Point.
	Single precision (IEEE 754)	4	4	The encoding of floating point numbers is described in [ARM ARM] chapter C2, VFP Programmer's Model, §2.1.1 Single-precision format, and §2.1.2 Double-precision format.
	Double precision (IEEE 754)	8	8	
Containerized vector	64-bit vector	8	8	Containerized Vectors.
	128-bit vector	16	8	
Pointer	Data pointer	4	4	Pointer arithmetic should be unsigned. Bit 0 of a code pointer indicates the target instruction set type (0 ARM, 1 Thumb).
	Code pointer	4	4	

2.3. Exploring SIMD Parallel Computing

Wireless MMX instructions are designed to do parallel computing in packed data, so in order to use IWMMXT intrinsics; we have to generate SIMD-Ready vectors first.

2.3.1. Generation of SIMD-Ready Vectors

✧ Loop unrolling

For some codes that apply same operations to different data (especially, within one loop), the most common technique to generate SIMD-Ready vectors is loop unrolling.

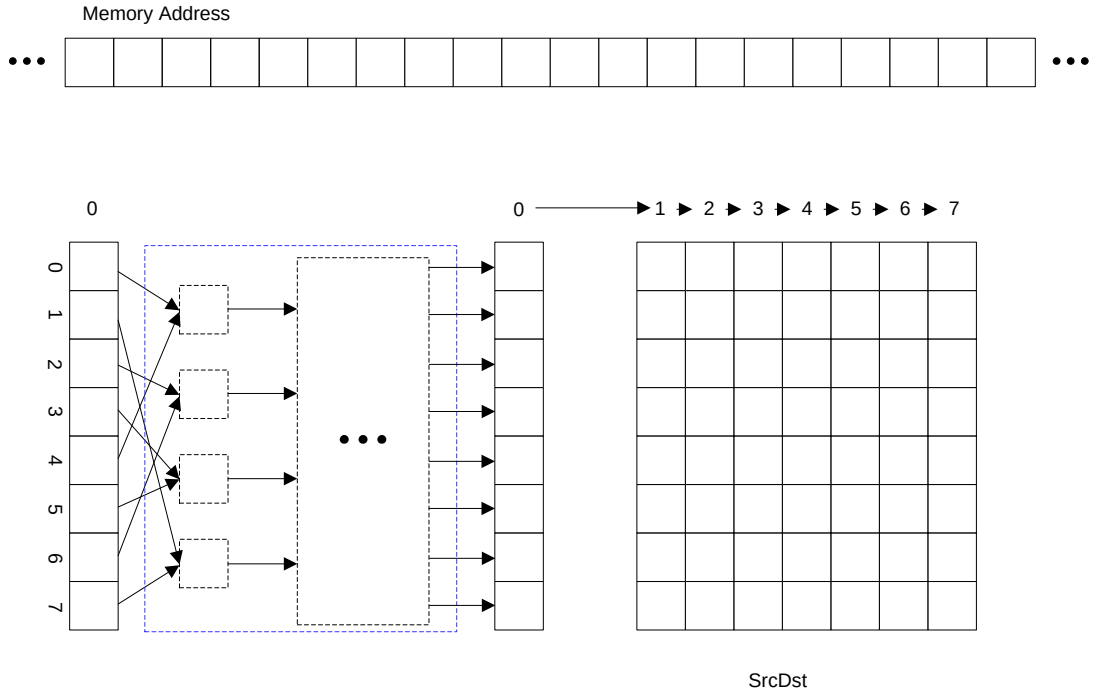

Figure 1 Scalar IDCT column transformations

For example, the invert DCT function in jpegdec (ippiDCTQuantInv_JPEG_16s_I) applies transformations on each column. As shown in Figure 1, scalar implementations usually do the transformations within one loop (8 times): the loop body loads eight elements from first column, applies calculations to them, and then store them back to memory.

Considering the data access pattern in the transformation and data arrangement in the memory, as there is not cross-referencing access to different columns' data, we can load contiguous data in a row together and apply the same transformations to them.

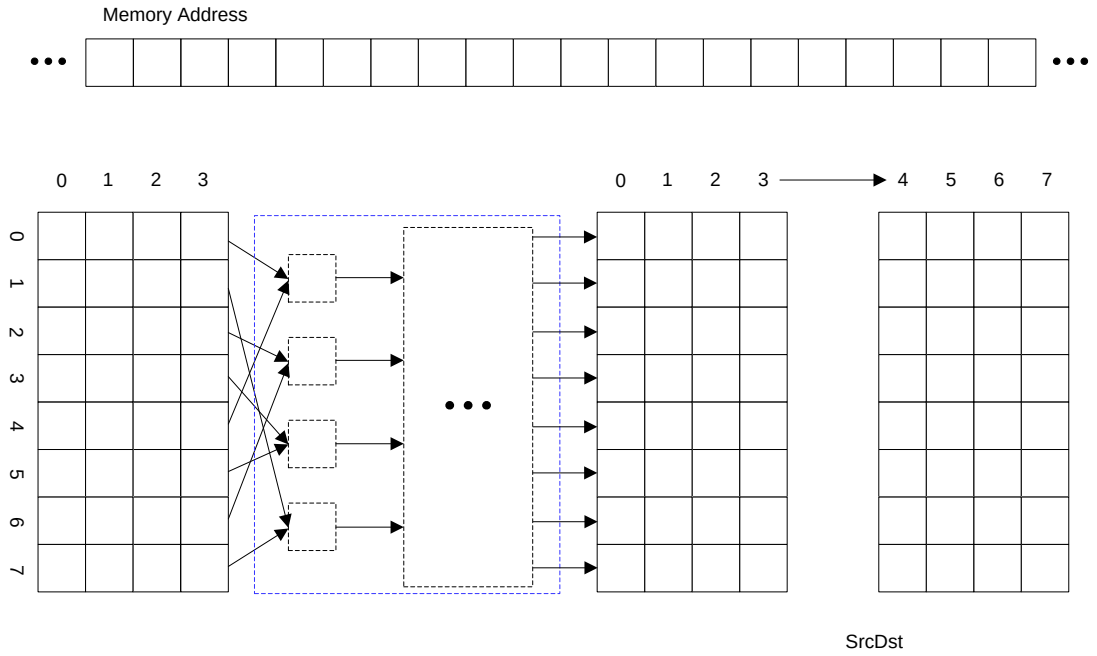

Figure 2 SIMD IDCT column transformations

The IWMMXT intrinsics could apply calculations on four packed bytes, so we can easily unroll the loop four times to utilize the SIMD parallel computing features. In this case, the whole transformation would need only twice calculation, as shown in Figure 2. Figure 3 contains some of the related code fragments.

```
for (i = 0; i < 8; i ++) {

    c0 = (Ipp16s)(pSrcDst[0*8]+pSrcDst[4*8]);
    c1 = (Ipp16s)(pSrcDst[0*8]-pSrcDst[4*8]);

      ...

    pSrcDst[0*8] = (Ipp16s)((e0+c7+8)>>4);
    pSrcDst[1*8] = (Ipp16s)((e1+c6+8)>>4);
    pSrcDst[2*8] = (Ipp16s)((e2+c5+8)>>4);
    pSrcDst[3*8] = (Ipp16s)((e3+c4+8)>>4);
    pSrcDst[4*8] = (Ipp16s)((e3-c4+8)>>4);
    pSrcDst[5*8] = (Ipp16s)((e2-c5+8)>>4);
    pSrcDst[6*8] = (Ipp16s)((e1-c6+8)>>4);
    pSrcDst[7*8] = (Ipp16s)((e0-c7+8)>>4);

    pSrcDst ++;
}
```

```
tmp1 = pSrcDstm64[0]; /*load 0-3 column to tmp1, first row*/
tmp2 = pSrcDstm64[4]; /*load 0-3 column to tmp2, third row*/
tmp3 = pSrcDstm64[8]; /*load 0-3 column to tmp3, fifth row*/
tmp4 = pSrcDstm64[12]; /*load 0-3 column to tmp4, seventh row*/

tmp5 = _mm_adds_pi16 (tmp1, tmp3); /*c0 = d0 + d4 */
tmp6 = _mm_subs_pi16 (tmp1, tmp3); /*c1 = d0 - d4 */
   ... ...
   ... ...
pSrcDstm64[8] = tmp12 ; /* pDst[4] = (e3-c4+8) >> 4 */
```

```
/* 0-3 column*/
idct_column(pSrcDstm64);

/* advance pSrcDstm64*/
pSrcDstm64++;

/* 4-7 column*/
idct_column(pSrcDstm64);
```

Figure 3 Code fragments of IDCT column transformations

✧ Data Realignment:

The loop unrolling is the most simple and common way to generate SIMD-Ready vectors, but such regular loop is usually rare. More often than not, the SIMD-Ready vectors come from manual data realignment.

For example, the row transformation in IDCT (Figure 4) also applies complicated calculation to row one by one. But considering the data arrangement in memory: the data for different loops are not contiguous; we could not unroll the loop as in column transformation.

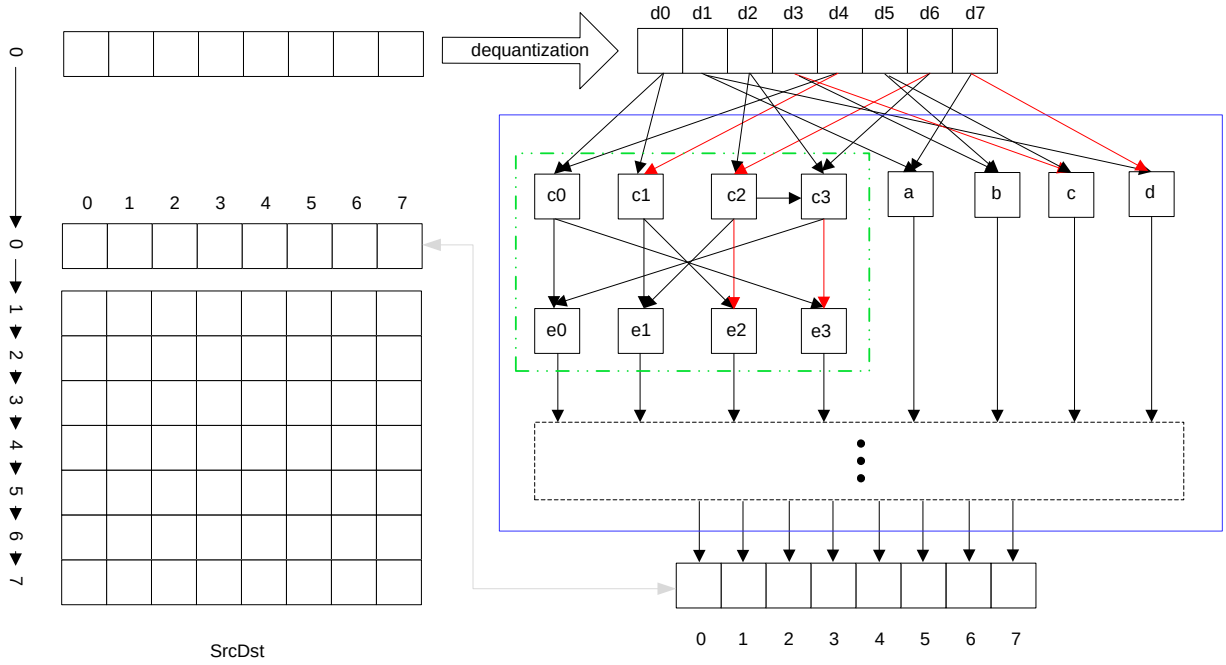

Figure 4 Scalar IDCT row transformations

After carefully investigation, we still can dig some SIMD computing. As shown in Figure 4, we could find some regular patterns in the computation, e.g. calculation of e0~e3. But there is another problem: the data path of the calculation is too complicated, there is not such IWMMXT intrinsics to calculate them directly. To utilize the intrinsics, we need to realign the data.

The calculation of e0~e3 contains two 32-bit adds and two 32-bit subs, so one possible way to use IWMMXT intrinsics is to do the add within one SIMD operation and sub within another. Figure 5 presents one possible realignment solution: just exchange the position of c1 and e1, and then we get regular pattern in data paths too.

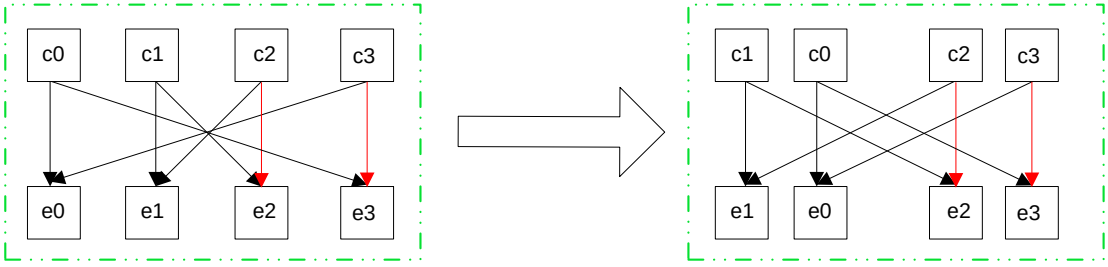

Figure 5 Data realignment for IDCT row transformations

There are a lot of different ways to realign data in IWMMXT: merge, shuffle, walign, insert, unpack. Detail intrinsics could be found in Table 3. One possible choice for previous example is using _mm_merge_si64, as in Figure 6.

Table 3 Data realignment intrinsics

Intrinsics prototype	Note
__m64 _mm_merge_si64 (__m64 a, __m64 b, const int n)	Take n bytes from b, the others from a
__m64 _mm_shuffle_pi16(__m64 a, int n)	Returns a combination of the four half words of a specified by the selector n.
__m64 _mm_align_si64(__m64 m1, __m64 m2, int count)	Extracts a 64-bit value from m1, m2 with count byte offset.
__m64 _mm_insert_pi8(__m64 a, int d, int n)	Inserts byte d into one of eight bytes of a specified by the selector n.
__m64 _mm_insert_pi16(__m64 a, int d, int n)	Inserts half word d into one of four half words of a specified by n.
__m64 _mm_insert_pi32(__m64 a, int d, int n)	Inserts word d into one of two words of a specified by n.
__m64 _mm_unpackhi_pi8 (__m64 m1, __m64 m2)	Interleaves the eight 8-bit values from the upper half of m1 and m2.
__m64 _mm_unpackhi_pi16 (__m64 m1, __m64 m2)	Interleaves the four 16-bit values from the upper half of m1 and m2.
__m64 _mm_unpackhi_pi32 (__m64 m1, __m64 m2)	Interleaves the two 32-bit values from the upper half of m1 and m2.
__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)	Interleaves the eight 8-bit values from the lower half of m1 and m2.
__m64 _mm_unpacklo_pi16 (__m64 m1, __m64 m2)	Interleaves the four 16-bit values from the lower half of m1 and m2.
__m64 _mm_unpacklo_pi32 (__m64 m1, __m64 m2)	Interleaves the two 32-bit values from the lower half of m1 and m2.

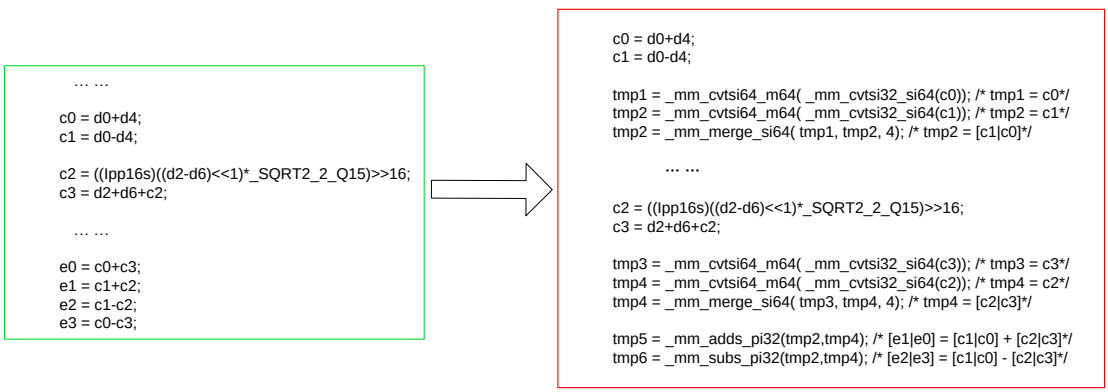

Figure 6 Code fragments of IDCT data realignment

2.3.2. Conditional Branches

With scalar operations, a conditional branch is used to alter the processing flow when a specified condition is met. The use of branch instructions, however, affects the efficiency of SIMD operations because it requires decomposing each vector into elements and processing them sequentially. Some SIMD processors therefore employ another method to obtain the same result as conditional branching [4]. But IWMMXT doesn't provide such mechanism (there is not vector selection instruction), so rewriting conditional branches with IWMMXT intrinsics is not recommended. If it is inevitable, we still have to decompose vector into elements.

3. Compiling and Debugging

3.1. Compiling Options

To use IWMMXT intrinsics, some compiling options are required, while some other options are recommended for better performance. Such options are listed in Table 4.

Table 4 IWMMXT Compiling Options

Options		Notes
-mmrvl-use-iwmmxt	**Required**	Turns on IWMMXT support in Marvell GCC
-O2	Optional	Recommended, it will turn on IWMMXT scheduler.
-fno-schedule-insns	Optional	Recommended if the calculation required a lot of temp variables (>8) at the same time. This will disable **pre-register allocation scheduler** to lower register pressure for better performance.
-fno-schedule-insns2[1]	Optional	Not recommended, it will disable **after register allocation scheduler**.

[1] Use this option only when you know the difference between pre-register and after-register allocation scheduler.

3.2. Debugging IWMMXT with GDB

3.2.1. Enable IWMMXT support in GDB

GDB 6.6 supports examining IWMMXT disassembly, but doesn't support examine IWMMXT registers. To enable IWMMXT support in GDB 6.6, you can do it as below:

```
(gdb) set arch iwmmxt2
```

GDB 6.8 or later supports both disassembly and registers. To enable the feature, you may do it like this:

```
(gdb) set tdesc filename gdb_src_path/gdb-6.8/gdb/features/arm-with-iwmmxt.xml
```

The xml files are provided in gdb's source code, in gdb-6.8/gdb/features directory.

3.2.2. Examine IWMMXT registers

After enabling IWMMXT support, you can view registers separately or with all-registers.

```
(gdb) info registers wR0
        wR0            {u8 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
                        u16 = {0x0,0x0, 0x0, 0x0},
                        u32 = {0x0, 0x0},
                        u64 = 0x0}
(gdb) info all-registers
```

3.2.3. Examine IWMMXT disassembly

If you have done Step 3.2.1, you should be able to see them correctly.

```
(gdb) x /20i 0xf600
 0xf600 <test+596>: wldrd wr0, [r11, #-108]
 0xf604 <test+600>: wmiattn wr0, wr2, wr1
 0xf608 <test+604>: wstrd wr0, [r11, #-108]
 0xf60c <test+608>: ldrd r2, [r11, #-108]
 0xf610 <test+612>: ldr r1, [pc, #1880] ; 0xfd70 <test+2500>
```

3.2.4. Enable IWMMXT support in gdbserver

Gdbserver in GDB 6.8 or later support IWMMXT registers too. To enable it, we have to export definition of **__IWMMXT__** before configure.

A simple script to compile it maybe as below:

```
#!/usr/bin/env bash
echo "Setting environments.."
export CC=arm-marvell-linux-gnueabi-gcc
export CFLAGS=" -D __IWMMXT__ "
echo "Configuring"
./configure --host=arm-marvell-linux-gnueabi \
            --target=arm-marvell-linux-gnueabi
echo "Make"
make
```

4. Miscellaneous

✧ SIMD flags intrinsics

Though IWMMXT provides intrinsics to access SIMD flags (_mm_getwcx, _mm_tandcx, _mm_torcx, _mm_textrcx, _mm_torvscx et al), they are intent to be used only with _mm_setwcx. They still can't be used to access SIMD flags set by other intrinsics.

Attention:

Use those intrinsics to access SIMD flags set by other intrinsics may produce unpredictable results.

✧ Fighting with register pressures

Intensive computation may require a lot of temp variables which may increase register pressure in GCC. This will cause GCC to spill instructions to memory, which will decrease application performance.

One recommended solution is to disable pre register allocation scheduler while compiling. See Section 3.1 for option details.

Another solution is to interleave ordinary computing with SIMD computing. Some times the SIMD computing itself may require more instructions or latency, but interleave them may lower the register pressure, prevent spilling memory access instructions, and hence increase performance. Code translation in Figure 6 is one of such examples.

✧ Use of inline asm

For some special patterns that are still not supported in GCC, inline asm could still be used. But the IWMMXT scheduler could not get enough information for instructions in inline asm, so usage of IWMMXT instructions in inline asm may affect the effect of instruction scheduling, it is not recommended.

References

[1] *Intel Wireless MMX Technology Developer Guide* (Order Number: 251793-001), Intel, August, 2002

[2] Intel Wireless MMX Technology Intrinsic Support (Chapter 20 of *Marvell C++ Compiler Users' Manual*), Marvell, December, 2008

[3] Jpegdec in IPP Codecs with Manchac, Marvell, 2009

[4] Basics of SIMD Programming (Chapter 2 of *Cell Programming Primer*), Geoff Levand, Sony, 2008

[5] Using the GNU Compiler Collection (GCC) (http://gcc.gnu.org/onlinedocs/gcc/), 2009

[6] Procedure Call Standard for the ARM Architecture (ARM IHI 0042C, ABI 2.07), Oct 10th, 2008.