MARVELL®

Notes on Using GCC IWMMXT

Table of Contents

1 NOTES ON USING GCC IWMMXT INTRINSIC

1.1 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.

1.2 Using GCC IWMMXT intrinsics in C sources

1.2.1 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.

1.2.2 Data types and Alignments

1.2.2.1 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

1.2.2.2 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	

1.3 Compiling and Debugging

1.3.1 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 3**.

Table 3 IWMMXT Compiling Options

Options		Notes
-mwmmxt	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.

1.3.2 Debugging IWMMXT with GDB

1.3.2.1 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.

1.3.2.2 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
```

1.3.2.3 Examine IWMMXT disassembly

If you have done Step 1.3.2.1, 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>
```

1.3.2.4 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
```

1.4 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 1.3.1 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.

✧ 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.

1.5 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.

2 NOTES ON USING GCC IWMMXT VECTOR CLASS HEADER FILE

2.1 Overview

The IWMMXT vector class header file contains functions abstracted from the IWMXMT instruction set. By providing a convenient interface to access the underlying IWMMXT instructions through intrinsics, the classes enable Single-Instruction, Multiple-Data (SIMD) operations. The objective is to improve application performance of complex and computation-intensive audio, video, and graphical data bit streams.

IWMMXT SIMD instructions can help improve efficiency of vector operations. These instructions can be implemented using inline asm, intrinsics or the C++ vector classes. The C++ vector class reuse the standard notation in C++, makes it much easier to implement SIMD operations over other methods.

This documentation is intended for programmers writing code for the IWMMXT coprocessor, particularly code that would benefit from the use of SIMD instructions. You should be familiar with C++ and the use of C++ classes.

2.2 Using GCC IWMMXT Vector Class in C++ sources

2.2.1 Include files and naming conventions

To use IWMMXT vector class, the ***<mmclass.h>*** file must be included.

The name of each class denotes the data type, signedness, bit size, number of elements using the following generic format:

<type>	<signedness>	<bits>	vec	<elements>
{ I }	{s | u }	{64 | 32 | 16 | 8 }		{1 | 2 | 4 | 8}

For example:

Is32vec2 denotes the class supporting operations on **two signed 32-bit integer** vector.

Note that not all the combinations of signedness, bits and elements are supported. Please refer to Section 2.3 for more details.

2.2.2 Rules for Operators

To use operators with the Ivec classes you must use one of the following three syntax conventions:

[Ivec_Class] R = [Ivec_Class] A [operator][Ivec_Class] B

Example 1: I64vec1 R = I64vec1 A & I64vec1 B;

[Ivec_Class] R = [operator] ([Ivec_Class] A,[Ivec_Class] B)

Example 2: I64vec1 R = andnot(I64vec1 A, I64vec1 B);

[Ivec_Class] R [operator]= [Ivec_Class] A

Example 3: I64vec1 R &= I64vec1 A;

[operator] an operator (for example, &, |, or ^)
[Ivec_Class] an Ivec class
R, A, B variables declared using the corresponding Ivec classes

2.2.3 Example

A simple example usage comparison is shown in **Table 4**. In the example, we add four 8-bit integer values twice, one with signed saturation the other with unsigned saturation. You can see how much easier it is to code with the vector class. Besides using fewer keystrokes and fewer lines of code, you can use standard C++ operators and don't have to remember the different intrinsic names or complicated inline asm syntax.

Table 4 Comparison among Inline ASM, Intrinsics, and Vector Class

Inline ASM	`__m64 op1, op2, op3, op4;` `__asm volatile(` `"waddbss %0, %1, %2"\` `: "=y" (chks), "y" (op1), "y" (op2));` `"waddbus %0, %1, %2"\` `: "=y" (chku), "y" (op3), "y" (op4));` `)`
Intrinsics	`#include <mmintrin.h>` `....` `__m64 op1, op2, op3, op4;` `chks = _mm_adds_pi8( op1, op2 );` `chku = _mm_adds_pu8( op1, op2 );` `....`
Vector Class	`#include <mmclass.h>` `....` `Is8vec8 op1, op2;` `Iu8vec8 op3, op4;` `chks = sat_add(op1,op2);` `chku = sat_add(op3,op4);` `....`

2.3 Vector Classes details

2.3.1 Overview

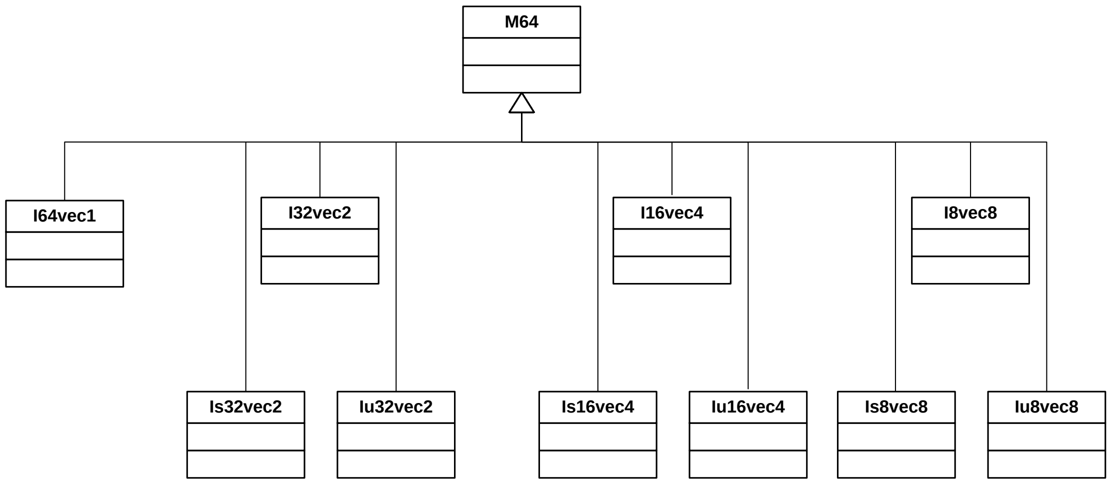

Figure 1 Vector Class Hierarchy

All supported vector classes are shown in **Figure 1**. The M64 class defines the __m64 data types from which the rest of the vector classes are derived. The first four child classes are derived based solely on bit sizes of 64, 32, 16 and 8 respectively for the I64vec1, I32vec2, I16vec4 and I8vec8 classes. The latter six of the classes require specification of signedness and saturation. Detail class names and their data type are listed in **Table 5**.

Table 5 Vector Classes Data Types

Class	Signedness	Data Type	Size	Elements
I64vec1	unspecified	__m64	64	1
I32vec2	unspecified	int	32	2
Is32vec2	signed	int	32	2
Iu32vec2	unsigned	int	32	2
I16vec4	unspecified	short	16	4
Is16vec4	signed	short	16	4
Iu16vec4	unsigned	short	16	4
I8vec8	unspecified	char	8	8
Is8vec8	signed	char	8	8
Iu8vec8	unsigned	char	8	8

The vector class supports almost all the regular operators, but it varies a little for different class. The supported operators are: Assignment, Logical, Addition and Subtraction, Multiplication, Shift, Comparison. Details about the supported operators could be found in **Table 6**.

Table 6 Operators Support Matrix

		I8vec8	Iu8vec8	Is8vec8	I16vec4	Iu16vec4	Is16vec4	I32vec2	Iu32vec2	Is32vec2	I64vec1
Assign	=	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Logical	&	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	&=	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	|	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	|=	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	^	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	^=	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	andnot	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Arith	+	Y	Y	Y	Y	Y	Y	Y	Y	Y	-
	-	Y	Y	Y	Y	Y	Y	Y	Y	Y	-
	+=	Y	Y	Y	Y	Y	Y	Y	Y	Y	-
	-=	Y	Y	Y	Y	Y	Y	Y	Y	Y	-
	sat_add	-	Y	Y	-	Y	Y	-	Y	Y	-
	sat_sub	-	Y	Y	-	Y	Y	-	Y	Y	-
Multiply	*	-	-	-	Y	Y	Y	Y	Y	Y	-
	*=	-	-	-	Y	Y	Y	Y	Y	Y	-
	mul_high	-	-	-	-	Y	Y	-	Y	Y	-
	mul_add	-	-	-	-	Y	Y	-	-	-	-
Shift	<<	-	-	-	Y	Y	Y	Y	Y	Y	Y
	<<=	-	-	-	Y	Y	Y	Y	Y	Y	Y
	>>	-	-	-	-	Y	Y	-	Y	Y	-
	>>=	-	-	-	-	Y	Y	-	Y	Y	-
Compare	==	Y	Y	Y	Y	Y	Y	Y	Y	Y	-
	!=	Y	Y	Y	Y	Y	Y	Y	Y	Y	-
	>	-	Y	Y	-	Y	Y	-	Y	Y	-
	>=	-	Y	Y	-	Y	Y	-	Y	Y	-
	<	-	Y	Y	-	Y	Y	-	Y	Y	-
	<=	-	Y	Y	-	Y	Y	-	Y	Y	-
Pack	unpack_low	Y	Y	Y	Y	Y	Y	Y	Y	Y	-
	unpack_high	Y	Y	Y	Y	Y	Y	Y	Y	Y	-
	pack_sat	-	-	-	-	-	Y	-	-	Y	-
	packu_sat	-	-	-	-	Y	-	-	Y	-	-

Note: All operators are restricted within same class only, except the assign operator.

2.3.2 Initialization and Data access

All vector classes could be initialized by three different values in examples below. All values are initialized with the most significant element on the left and the least significant to the right.

Example 1: __m64 initialization

I64vec1 A(__m64 m); Iu8vec8 A(__m64 m);

Example 2: __int64 initialization

I64vec1 A = __int64 m; Iu8vec8 A = __int64 m;

Example 2: Corresponding data type initialization

I32vec2 A (int A1, int A0);

Is32vec2 A (signed int A1, signed int A0);

Iu32vec2 A (unsigned int A1, unsigned int A0);

I16vec4 A (short A3, short A2, short A1, short A0);

The value of the class could be accessed by operator __m64().

For example, __m64 (I8vec8 A) would return the value of class A (of the type __m64).

2.3.3 Assign Operators

Any Ivec object can be assigned to any other Ivec object; conversion on assignment from one Ivec object to another is automatic. **Table 7** shows the corresponding intrinsics used to implement the assign operator.

Table 7 Corresponding Intrinsics for Assign Operators

Assign

	=
I8vec8	_mm_set_pi32, _mm_cvtsi32_si64
Iu8vec8	
Is8vec8	
I16vec4	
Iu16vec4	
Is16vec4	
I32vec2	
Iu32vec2	
Is32vec2	
I64vec1	

2.3.4 Logical Operators

The logical operators use the intrinsics listed in **Table 8**.

Table 8 Corresponding Intrinsics for Logical Operators

Logicals							
	&	**&=**	|	|=	^	^=	**andnot**
I8vec8	_mm_and_si64		_mm_or_si64		_mm_xor_si64		_mm_andnot_si64
Iu8vec8							
Is8vec8							
I16vec4							
Iu16vec4							
Is16vec4							
I32vec2							
Iu32vec2							
Is32vec2							
I64vec1							

2.3.5 Arithmetic Operators

Default behavior is no saturation operation. The arithmetic operators use the intrinsics listed in **Table 9**.

Table 9 Corresponding Intrinsics for Arithmetic Operators

Arithmetic						
	+	+=	**sat_add**	-	-=	**sat_sub**
I8vec8	_mm_add_pi8		-	_mm_sub_pi8		-
Iu8vec8			_mm_adds_pu8			_mm_subs_pu8
Is8vec8			_mm_adds_pi8			_mm_subs_pi8
I16vec4	_mm_add_pi16		-	_mm_sub_pi16		-
Iu16vec4			_mm_adds_pu16			_mm_subs_pu16
Is16vec4			_mm_adds_pi16			_mm_subs_pi16
I32vec2	_mm_add_pi32		-	_mm_sub_pi32		-
Iu32vec2			_mm_adds_pu32			_mm_subs_pu32
Is32vec2			_mm_adds_pi32			_mm_subs_pi32
I64vec1	-			-		

2.3.6 Multiplication Operators

Default is to return the lower 32-bits of result. The multiplication operators use the intrinsics listed in **Table 10**.

Table 10 Corresponding Intrinsics for Multiplication Operators

Multiply				
	*	*=	**mul_high**	**mul_add**
I8vec8	-			
Iu8vec8				
Is8vec8				
I16vec4	_mm_mullo_pi16		-	-
Iu16vec4			_mm_mulhi_pu16	_mm_madd_pu16
Is16vec4			_mm_mulhi_pi16	_mm_madd_pi16
I32vec2	_mm_mullo_pi32		-	-
Iu32vec2			_mm_mulhi_pu32	-
Is32vec2			_mm_mulhi_pi32	
I64vec1	-			

2.3.7 Shift Operators

The right shift argument can be any integer or Ivec value. The shift operators use the intrinsics listed in **Table 11**.

Table 11 Corresponding Intrinsics for Shift Operators

Shift				
	<<	<<=	>>	>>=
I8vec8	-	-	-	-
Iu8vec8	-	-	-	-
Is8vec8	-	-	-	-
I16vec4	_mm_sll_pi16, _mm_slli_pi16		-	
Iu16vec4			_mm_srl_pi16,_mm_srli_pi16	
Is16vec4			_mm_sra_pi16,_mm_srai_pi16	
I32vec2	_mm_sll_pi32, _mm_slli_pi32		-	
Iu32vec2			_mm_srl_pi32,_mm_srli_pi32	
Is32vec2			_mm_sra_pi32,_mm_srai_pi32	
I64vec1	_mm_sll_pi64, _mm_slli_pi64		-	

2.3.8 Pack/Unpack Operators

The pack/unpack operators use the intrinsics listed in **Table 12**.

Table 12 Corresponding Intrinsics for Pack/Unpack Operators

Pack/Unpack				
	unpack_low	**unpack_high**	**pack_sat**	**packu_sat**
I8vec8	_mm_unpacklo_pi8	_mm_unpackhi_pi8	-	-
Iu8vec8			-	-
Is8vec8			-	-
I16vec4	_mm_unpacklo_pi16	_mm_unpackhi_pi16	-	-
Iu16vec4			-	_mm_packs_pu16
Is16vec4			_mm_packs_pi16	-
I32vec2	_mm_unpacklo_pi32	_mm_unpackhi_pi32	-	-
Iu32vec2			-	_mm_packs_pu32
Is32vec2			_mm_packs_pi32	-
I64vec1	-			

2.3.9 Comparison Operators

The equality and inequality comparison operands can have mixed signedness, but they must be of the same size. The comparison operators for “less-than” and “greater-than” must be of the same sign and size. The comparison operators use the intrinsics listed in **Table 13**.

Table 13 Corresponding Intrinsics for Comparison Operators

Compare						
	==	!=	>	<	>=	<=
I8vec8	_mm_cmpeq_pi8	_mm_andnot_si64, _mm_cmpeq_pi8	-		-	
Iu8vec8			_mm_cmpgt_pu8		_mm_andnot_si64, _mm_cmpgt_pu8	
Is8vec8			_mm_cmpgt_pi8		_mm_andnot_si64, _mm_cmpgt_pi8	
I16vec4	_mm_cmpeq_pi16	_mm_andnot_si64, _mm_cmpeq_pi16	-		-	
Iu16vec4			_mm_cmpgt_pu16		_mm_andnot_si64, _mm_cmpgt_pu16	
Is16vec4			_mm_cmpgt_pi16		_mm_andnot_si64, _mm_cmpgt_pi16	
I32vec2	_mm_cmpeq_pi32	_mm_andnot_si64, _mm_cmpeq_pi32	-		-	
Iu32vec2			_mm_cmpgt_pu32		_mm_andnot_si64, _mm_cmpgt_pu32	
Is32vec2			_mm_cmpgt_pi32		_mm_andnot_si64, _mm_cmpgt_pi32	
I64vec1	-					

2.4 References

[1] *Intel® C++ Compiler for Linux Compiler Reference* (Document Number: 307777 - 002US), Intel

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

3 NOTES ON USING GCC IWMMXT AUTO-VECTORIZATION

3.1 Turning on auto vectorization

To switch on auto vec for the compiler, the following options should be given:

-O3 (or -ftree-vectorize -O2) and -mwmmxt

To view the detailed optimization information, use the additional options:

-ftree-vectorizer-verbose=3

3.2 Hint: use a same or wider data type for the target in a loop

For the following codes:

```
>>>   while (p<i) {
>>>         pTable->huffVal[p] = pHuffValue[p];
>>>         p++;
>>>   }
```

If the data type of pTable->huffVal[p] is same as or wider than pHuffValue[p], this loop could be vectorized. For example, the type of pTable->huffVal[p] is int while the type of pHuffValue[p] is short.

3.3 Hint: use "aligned" attribute for array when possible

For the following simplest code:

```
>>>   extern unsigned char arr1[100];
>>>
```

```
>>>   extern a;
>>>
>>>   int foo()
>>>   {
>>>       int i;
>>>       for( i = 0; i < 100; ++i )
>>>           {
>>>               arr1[i] = 0;
>>>           }
>>>   }
```

Loop peeling (or loop versioning) will be used by auto vectorizer, which will bloat code size, to overcome such problems, try to add "aligned" attribute to the external declaration:

```
>>>   extern unsigned char __attribute__((aligned(8))) arr1[100];
>>>
>>>   extern a;
>>>
>>>   int foo()
>>>   {
>>>       int i;
>>>       for( i = 0; i < 100; ++i )
>>>           {
>>>               arr1[i] = 0;
>>>           }
>>>   }
```

3.4 Issue: autovected cases for pointers with loop peeling

```
>>> IPPCODECFUN(IppCodecStatus, _ijxSetZero_16s) (Ipp16s * pDst, int len)
>>> {
>>>         int    i;
>>>         Ipp16s pp[len];
>>>
>>>     //_IPP_CHECK_ARG(pDst && len > 0);
>>>
>>>         for ( i = 0; i < len; i ++ ) {
>>>                 pDst[i] = 0;
>>>     }
>>>
>>>     return IPP_STATUS_NOERR;
>>> }
```

For the above code snippet, the vectorizer will use loop peeling to enable loop vectorizing. The side effect is that code size is bloated.

Currently there is no way to specify pointer alignment in GCC, the following is a description from the mailing list of one of the GCC developers:

"Unfortunately there's no way to specify alignment attribute of pointers in GCC - the syntax was allowed in the past but not really supported correctly, and then entirely disallowed (by this patch http://gcc.gnu.org/ml/gcc-patches/2005-04/msg02284.html). This issue was discussed in details in these threads:
http://gcc.gnu.org/bugzilla/show_bug.cgi?id=20794
http://gcc.gnu.org/ml/gcc/2005-03/msg00483.html (and recently came up again also in
http://gcc.gnu.org/bugzilla/show_bug.cgi?id=27827#c56).
The problem is that "We don't yet implement either attributes on array parameters applying to the array not the pointer, or attributes inside the [] of the array parameter applying to the pointer. (This is documented in "Attribute Syntax".)" (from the above thread)."

So there seems no workaround for this loop peeling problem on pointers now.

3.5 Issue: straight lines of statements will not be vectorized.

For the following example:

```
>>>  int foo(int i)
>>>  {
>>>     ...
>>>     a1[ i ] = 0
>>>     a1[ i + 1 ] = 0;
>>>        a1[ i + 2 ] = 0;
>>>        a1[ i + 3 ] = 0;
>>>        ...
>>>    }
```

The assignment to a1 array will not be vectorized, because all of these statements belongs to a same iteration vector. In other words, vectorizer will only try to combine statements from different but consecutive loop iteration vectors, statements that belong to same iteration vector will not be combined.