Intel® Wireless MMX™ Technology

Developer Guide

August, 2002

Order Number: 251793-001

Contents

Figures

Tables

Introduction 1

The past decade has seen the ever-increasing popularity of desktop applications that incorporate, and rely heavily on, powerful multimedia capabilities. These capabilities include video and audio playback, 3D graphics, and digital photography, among others. Early on, Intel recognized the growing trend towards rich multimedia and realized that the inherent complexity of the algorithms required for multimedia, coupled with consumer's expectations for ever richer and more complex data streams to feed those algorithms, would require significant new processor capabilities. Through the efforts of Intel's processor research and development, technologies such as Intel® MMX™ technology, and Streaming SIMD Extensions (SSE) were created. These meet not only the specific needs of multimedia acceleration, but also provide a powerful set of processor tools that enable a wide range of multimedia application capabilities for the future.

The trend for richer multimedia capabilities is not unique to the desktop. As the mobile-device market segment matures (cell phones, smartphones, PDAs, etc.) the end-users are demanding similar, and in many cases the same, multimedia experiences that they enjoy on their desktop. For example:

- Consumers are playing 2D and 3D games on PDAs and watching movie clips while sitting on airplanes.
- Business users are reading their e-mail, viewing the latest news clips, and updating presentation materials while away from their offices.
- Car drivers are using speech recognition to dial phone numbers hands-free.
- Parents are taking pictures of kids with their PDAs and uploading it to their desktops.
- Everyone is listening to MP3s.

However, one of the biggest challenges for multimedia on mobile devices is to provide high-performance, low-power consumption. Playing a richly detailed 3D game on a PDA can be highly enjoyable until the PDA runs out of power. What is needed is a processor technology that, when utilized with the Intel® XScale™ microarchitecture, provides:

- A familiar development environment to encourage the porting of desktop applications to mobile devices.
- Fast multimedia acceleration that accomplishes the same task in less time (power savings).
- Fast multimedia acceleration that accomplished the same task in less power.
- Fast multimedia acceleration that accomplishes more tasks in the same amount of time (performance enhancement).

The Wireless MMX™ technology integrates the high performance of MMX™ technology and the integer functions from SSE to the Intel® XScale™ microarchitecture. Wireless MMX™ technology brings the power of a 64-bit, parallel, multimedia architecture to handheld systems in a programmer's model that is already familiar to thousands of developers. Like MMX™ technology and SSE, Wireless MMX™ technology utilizes 64-bit wide Single Instruction Multiple Data (SIMD) instructions which allow it to concurrently process up to eight data elements in a single cycle.

The Wireless MMX™ technology gives a large performance boost to many multimedia applications, such as motion video, combined graphics with video, image processing, audio synthesis, speech synthesis and compression, telephony, conferencing, and 2D and 3D graphics.

1.1 How Wireless MMX™ Technology Works

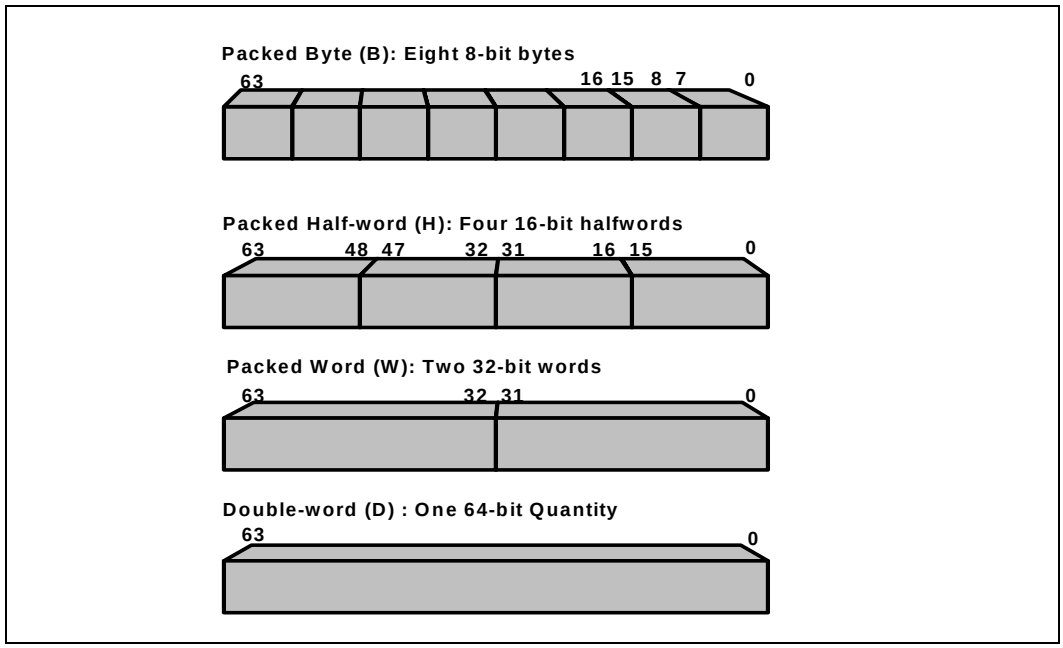
Wireless MMX™ technology exploits the data parallelism present in a large number of multimedia algorithms by executing the same operation on different data elements in parallel. This is accomplished by packing data elements into a single register and introducing new types of instruction to operate on packed data. All Wireless MMX™ technology data types are 64-bits wide.

The Wireless MMX™ technology defines three packed data types and the 64-bit double word in a manner similar to the data types found in MMX™ technology. The elements within the packed data type are fixed-point integers which may be either signed or unsigned. In the context of the ARM* architecture, a word is 32 bits, Figure 1-1 on page 2 illustrates the naming strategy that applies to the different data types in the Wireless MMX™ technology.

The instructions created by Wireless MMX™ technology to operate on these data types are:

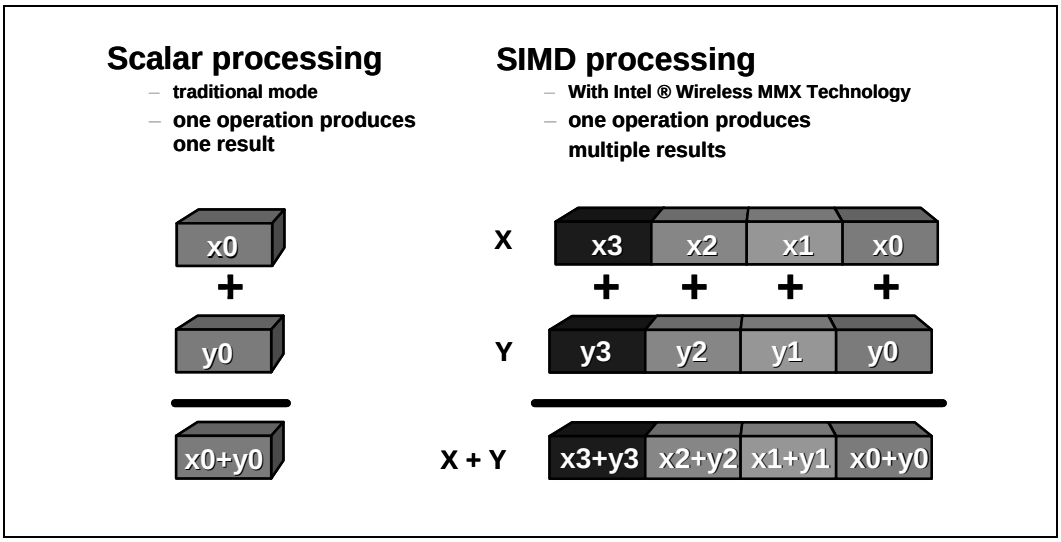
- Arithmetic and logical instructions
- Data reformatting instructions
- Multiplication instructions
- Data comparison
- Data transfer

Figure 1-1. Wireless MMX™ Technology Data Types



Using these new instructions it is possible to operate on data elements in the packed format. Figure 1-2 on page 3 shows an example of how a packed addition instruction operates on the half word data in a packed register. Here a single instruction operates on multiple data elements. This is often referred to as SIMD processing (single instruction multiple data). In SIMD processing each packed register is considered to be four independent data values. The packed add executes four additions on the four pairs of data values. The result of the operation of one element of the data does not affect any of the other results; thus, four operations are executed in parallel, with four independent results being produced. Similarly, eight operations are executed in parallel when the operands are packed bytes, and two operations are executed in parallel when the operands are packed words. This can be compared to the traditional scalar approach used for standard Intel® XScale™ microarchitecture data operations where an instruction operates on one set of word data operands and produces one result.

Figure 1-2. Packed Addition Operation



1.1.1 Exploiting Parallelism

Many multimedia algorithms execute the same set of operations on a large number of data elements. This creates an opportunity for parallel processing, referred to as data parallelism in this book, which microprocessors that operate only on one set of data at a time do not exploit. For example, when processing still images or video frames, the same operation is most often done over all of the pixels of the image or of the frame. Pixels are usually represented using 8-bit or 16-bit data elements. Executing the same operations on 8-bit, 16-bit or even 32-bit data is what Wireless MMX™ technology is all about. By executing these operations two, four, or eight at a time, Wireless MMX™ technology speeds up applications that exhibit data parallelism.

1.2 Wireless MMX™ Technology Web Resources

The following online resources are available:

- Intel® Wireless MMX™ technology home page:*http://www.intel.com/go/wirelessMMX*
- Intel® PCA Developer Network: *http://www.intel.com/pca/developernetwork*
- Intel® PCA processors: *http://www.intel.com/design/pca/applicationsprocessors/*

1.3 About This Document

This document describes the software programming optimizations and considerations for the Wireless MMX™ technology. Additionally, it covers coding techniques that will help you get started in coding your application. This documentis organized into four chapters, including this chapter, and three appendices.

- **Chapter 1 “Introduction”**
- **Chapter 2 “Programmers Model”**

  This chapter provides an overview of the programming model and includes detailed description of the data, status, and control registers.
- **Chapter 3“Optimization Techniques”**

  This chapter provides a list of rules and guidelines that will help you develop fast and efficient code. Additionally, it provides information on general optimization, instruction scheduling and selection, and cache and memory optimization.
- **Chapter 4“Porting Existing Code”**

  This chapter reviews the steps for porting existing MMX™ technology and SSE code sequences to Wireless MMX™ technology.
- **Appendix A “Instruction Set”**

  This appendix summarizes the Wireless MMX™ technology instructions.
- **Appendix B “Instruction Encoding Summary”**

  This appendix summarizes the Wireless MMX™ technology instruction encoding.
- **Appendix C “Intrinsic Support”**

  This appendix summarizes the Wireless MMX™ technology C callable intrinsics.

It is assumed that the reader is familiar with the Intel® XScale™ microarchitecture.

Programmers Model 2

The Wireless MMX™ technology unit is a tightly coupled coprocessor of the Intel® XScale™ microarchitecture. The programmers model is an extension of the Intel® XScale™ microarchitecture programming model and comprises of the following components:

- Mapping onto the ARM* architecture
- New state introduced in the form of registers files
- New instructions for processing 64-bit packed data formats
- New arithmetic Flags
- Control and status flags

In common with the Intel® XScale™ microarchitecture programing model, Wireless MMX™ technology supports the conditional execution primitives of the ARM* architecture.

The Wireless MMX™ technology programming model allows Wireless MMX™ instructions to be interleaved with Intel® XScale™ microarchitecture instructions in a single thread of control, with no complex synchronization or data coherency issues.

2.1 Mapping Wireless MMX™ technology onto the ARM* architecture

The ARM* architecture includes a framework for the addition of coprocessors to the main core. The architecture allows up to 16 coprocessors in the standard coprocessor space, the Wireless MMX™ technology utilizes two ARM* coprocessors; **coprocessor 0** and **coprocessor 1**. Two coprocessors are used to support the range of instructions and data types. These coprocessors support Wireless MMX™ technology data and control registers using standard coprocessor transfer instructions (MCR/MRC & MCRR/MRRC). The coprocessor interface supports both a 32- and 64-bit interface to the Intel® XScale™ microarchitecture (to support MRC and MRCC class of coprocessor data transfer instructions)

Wireless MMX™ instructions map onto one of three classes of ARM* coprocessor instruction formats:

- Coprocessor data transfers
- Coprocessor data operations
- Coprocessor register load and store

These instructions can be executed conditionally, depending on the state of the ARM* flags (note the exception to this is Load and Stores to Control (wCx) registers).

Note: Some data operations can also be performed while transferring data between the Intel® XScale™ microarchitecture and the Wireless MMX™ technology register files.

2.2 Wireless MMX™ technology Instruction Set Summary

The Wireless MMX™ technology programming model provides a rich set of instructions that perform parallel operations on multiple data elements packed into 64-bit words. In addition to providing operations on the Wireless MMX™ technology registers, several parallel operations are available on data elements residing in the Intel® XScale™ microarchitecture register file. The instruction set can be functionally partitioned as follows:

- Compatibility instructions
- New Wireless MMX™ instructions
- Coprocessor data transfer instructions
- Coprocessor register load and store

The following sections give an overview of each functional group of instructions.

2.2.1 Compatibility Instructions

The main class of instructions in the Wireless MMX™ technology programmers model are the compatibility instructions. These instructions are added to ease code portability between desktop and mobile platforms. In particular they provide equivalent functionality to:

- MMX™ technology
- Integer Intel Streaming SIMD Extensions (SSE)
- Intel® XScale™ microarchitecture media instructions

Wireless MMX™ technology provides equivalent functionality to all the MMX™ technology instructions and integer instructions from SSE instruction group. This group of instructions is targeted to accelerate software performance in applications that use video and 3D rasterization, video encoding and decoding on blocks of data, as well as a wide variety of signal processing algorithms.

The Intel® 80200 processor introduced a DSP coprocessor to the Intel® XScale™ microarchitecture for the purpose of increasing performance and the precision of audio processing algorithms. This coprocessor introduced several new instructions which use a 40-bit accumulator. The Wireless MMX™ technology provides equivalent functionality in a binary compatible form, but with the extended precision of a 64-bit accumulator.

Table 2-1 summarizes the instructions in this class.

Table 2-1. Compatibility Instruction Summary (Sheet 1 of 2)

Instruction	Description
WADD WSUB	Add or subtract eight bytes, four 16-bit half-words, or two 32-bit words
WCMPEQ WCMPGT	Compare eight bytes, four 16-bit half-words, or two 32-bit elements in parallel. Result is mask of 1's if true or 0's if false.
WMULL WMULM	Multiply four signed16-bit words in parallel. Low-order or high order 16-bits of 32-bit result are produced

Table 2-1. Compatibility Instruction Summary (Sheet 2 of 2)

Instruction	Description
WMADD	Multiply four 16-bit words in parallel and add
WROR WSRA WSLL WSRL	Rotate right, shift arithmetic right, logical right and left of 4 half-words, 2 words, or the full 64-bit double-word, in parallel
WUNPCK WUNPCK	Unpack eight bytes, four 16-bit halfwords, words, or two 32-bit words
WPACK WPACK	Pack double-word to words or words to bytes
WAND, WANDN WOR, WXOR	64-bit logical operations
WMAX WMIN	Vector maximum/minimum selection between elements in two registers.
TMOVMSK	Transfers the most significant bit of each SIMD field of a Wireless MMX™ technology register to an ARM* register
WAVG2	Two element average on unsigned vectors of 8-or 16-bit data
TINSR	Transfers and inserts 8-, 16-, 32-bit data from an ARM* register to a field in a Wireless MMX™ technology data register
TEXTRM	Extracts and transfers an 8-, 16-, 32-bit field from a Wireless MMX™ technology data register to an ARM* register
WSAD	Performs sum-of-absolute-differences on unsigned vectors of 8- or 16-bit data.
WSHUFH	Shuffles 16-bit data values specified by an 8-bit immediate
TMIA	Performs multiply-accumulate using signed 32-bit operands from the two source ARM* Core registers
TMIAxy	Performs a 16-bit multiply-accumulate using two signed operands from the ARM* Core registers
TMIAph	Performs multiply-accumulate using signed 16-bit operands from the two source ARM* Core registers

2.2.2 New Wireless MMX™ Instructions

The new instructions are designed to accelerate critical software kernels found in a number of multimedia applications. Special instruction support is provided to integrate the SIMD programing model into the ARM* architecture. These include: management of the SIMD status flags and providing alignment of operands. The new Wireless MMX™ instructions are summarized in Table 2-2 on page 8

Table 2-2. Wireless MMX™ Instructions

Instruction	Comments
WMAC	Multiply four signed or unsigned 16-bit half-words in parallel and accumulate with a 64-bit register.
WACC	Unsigned addition of eight bytes, four half-words, or two words.
WALIGN WALIGNI	Performs alignment on byte boundaries between two registers
TANDC TORC	Performs logical operations across the fields of the SIMD PSR (wCASF) and sends the result to the ARM* CPSR
TEXTRC	Extracts a 4-, 8-, 16-bit field specified by the 3-bit Immediate field data from the SIMD PSR register (wCASF)
TBCST	Broadcasts a value from the ARM* source register, Rn, or to every SIMD position in the Shortened Product Name Destination register, wRd; can operate on 8-, 16-, and 32-bit data values

2.2.3 Transfers to and from Coprocessor Register

The transfer instructions for moving data between the Wireless MMX™ technology control and data registers and the Intel® XScale™ microarchitecture registers are summarized in Table 2-3.

Table 2-3. Transfer Instructions to/from Wireless MMX™ technology and Intel® XScale™ microarchitecture Registers

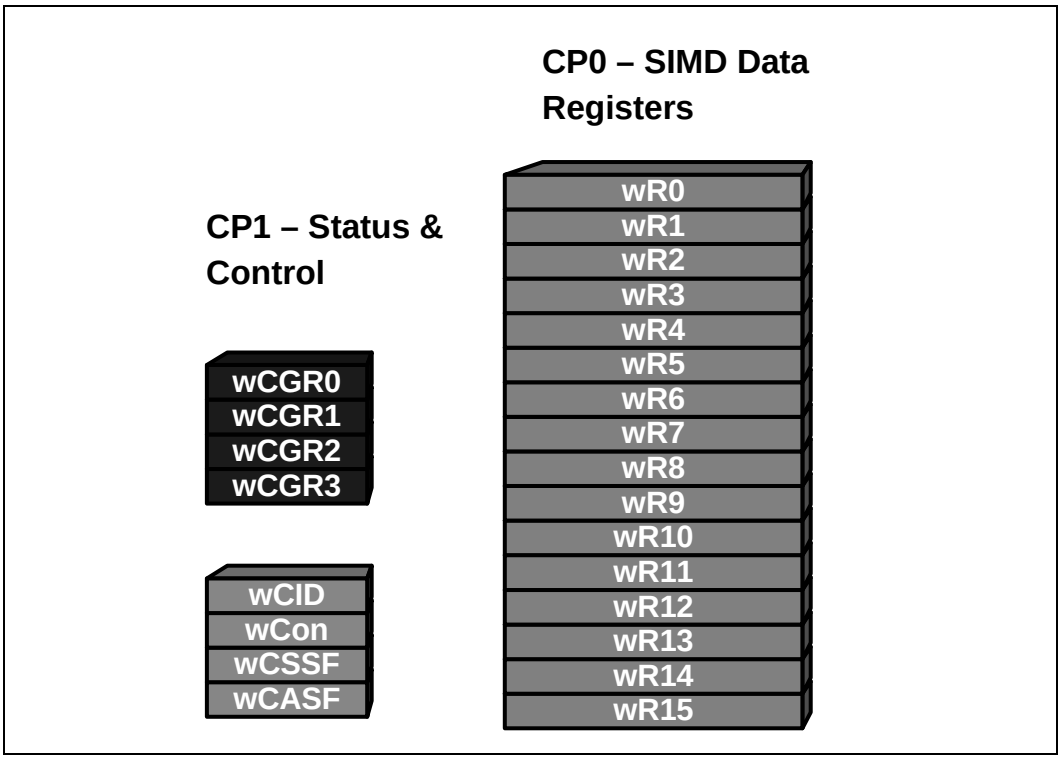
Instruction	Comments
TMRRC	MRRC transfers the contents of the wRn 64-bit Wireless MMX™ technology data register to two ARM* core Destination registers (RdHi, RdLo)
TMCRR	TMCRR transfers the contents of the two ARM* registers (RdHi and RdLo) to wRd (the 64-bit Wireless MMX™ technology destination register.)
TMRC	TMRC transfers the contents of the 32-bit Wireless MMX™ technology control register (wCx) to the ARM* core Destination register
TMCR	TMCR transfers the contents of the Rn ARM* Core registers to a 32-bit wCx Wireless MMX™ technology control register
WLDR/WSTR	Data Load Store operation

2.3 Wireless MMX™ Technology Registers State

The Wireless MMX™ technology programmers model supports two register files. A main register file, mapped onto coprocessor 0 space, is provided for holding 16x64-bit packed data and a 32-bit control register file, mapped onto coprocessor one space, is provided for auxiliary support functions. The register file organization is shown in Figure 2-1 on page 9.

When transferring data between the main packed data registers (wRx) and the ARM* core register file use the TMRCC and TMCRR instructions. If only part of the Wireless MMX™ technology register is to be transferred, use the TEXTRM and TINSTR instructions to extract or insert sub-fields of the register.

Figure 2-1. Wireless MMX™ Technology Register File Organization



2.3.1 Status and Control Registers

For ARM* coprocessor transfers, the Status and Control registers of Wireless MMX™ technology map into the registers of coprocessor 1, which allows the standard MRC and MCR ARM* instructions to read and write to these registers.

The Status registers include:

- SIMD arithmetic flags and saturation flags.
- Saturation status (wCSSF)

The Control registers include the following:

- wCGR0-wCGR3 (32-bit General-Purpose registers used for alignment, and shift.)
- Coprocessor ID (wCID)
- Coprocessor Control register (wCon)

2.3.2 Register Map

Table 2-4 on page 10 shows how the Control and Status registers map into coprocessor 1 register addresses.

Table 2-4. Status and Control Register Mappings

Mnemonic	Register Name	Coprocessor Register Number
wCID	Coprocessor ID, Revision, Status	0
wCon	Control	1
wCSSF	Saturation SIMD Flags	2
wCASF	Arithmetic SIMD Flags	3
	Reserved	4-7
wCGR0	General Purpose register 0	8
wCGR1	General Purpose register 1	9
wCGR2	General Purpose register 2	10
wCGR3	General Purpose register 3	11
	Reserved	12-15

2.3.3 General-Purpose Registers

There are four 32-bit, General-Purpose registers defined in the Wireless MMX™ technology coprocessor that store constants for use in memory alignment and shifting. The registers are addressed by the mnemonic wCGR0-wCGR3. Table 2-5 shows the format and accessibility of the General-Purpose registers.

Table 2-5. Register Bitmap for wGRn Registers

Registers 8-11 — **wCGRn** — **Coproc 1**

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	GPData																															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	Access	Name	Description
31:0	Read/Write	GPData	General-Purpose Data register

This register can also be used as temporary storage.

2.3.3.1 Data Transfer

To transfer data between the ARM* and the General-Purpose registers use the TMRC and TMCR instructions. The General-Purpose registers are used for constants and address offsets. As most of the data to be loaded into these registers comes from the ARM* registers (address offsets, constants), there is no direct path between the wCx registers and the wRx SIMD registers.

2.3.3.2 Memory Alignment

The General-Purpose registers can contain the offset from a word boundary when unaligned data is loaded using a standard ARM*-aligned load. This offset lets the WALIGN instruction extract the correct 64-bit double word from two aligned double words that have already been loaded. The

memory-alignment value is set by extracting the lower order address bits from the ARM* Base register and transferring these to the Wireless MMX™ technology Destination General-Purpose register.

2.3.3.3 Shift Value

A shift instruction with the G-qualifier specified uses the shift value stored in the General-Purpose register (specified in the wRm field).

2.3.4 Coprocessor ID

The read-only, coprocessor ID register contains the connected coprocessor type and its revision number. For coprocessor ID register information, refer to Table 2-6.

Table 2-6. Register Bitmap for wCID – Coprocessor ID

	Register 0																													
	wCID																								Coproc 1					
	Intel Vendor ID								Architecture Version								Coprocessor Type								Revision					
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	1	0	1	0	0	1	0	0	0	0	0	1	0	1	0	0	0	1	0	0	0	0	Major				Minor			
Reset	0	1	1	0	1	0	0	1	0	0	0	0	0	1	0	1	0	0	0	1	0	0	0	0	Implementation **Dependant**				Implementation **Dependant**			

Bits	Access	Name	Description
31:24	Read Only	Vendor	Intel® Vendor ID 0x69 – Intel ID
23:16	Read Only	Arch	Architecture Version 0x05 – ARM* Architecture V5 Compliant
15:8	Read Only	CPTyper	Coprocessor Type 0x10 – Wireless MMX™ technology
7:4	Read Only	Major	Major revision number –implementation dependant
3:0	Read Only	Minor	Minor revision number – implementation dependant

Note: Writes to this register are ignored and do not effect the CUP bit in the coprocessor control register.

2.3.5 Coprocessor Control Register

The coprocessor Control register is used to set all user-configuration variables. This register provides a way of detecting if the coprocessor is currently in use (See Table 2-7 on page 12.)

Table 2-7. Register Bitmap for wCon – Coprocessor Control

Register 1	wCon	Coproc 1
	Alias	

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Reserved																														MUP	CUP
Reset	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	0	0

Bits	Access	Name	Description
31:2	N/A	Reserved	Reserved for future use – Write as zero, reads as unknown.
1	Read/Write 1 to clear	MUP	One of the Main registers (wRn) has been updated 0 – Not Updated 1 – Updated
0	Read/Write 1 to clear	CUP	One of the Control Registers (wCx) has been updated.| 0 – Not Updated 1 – Updated

The *CUP/MUP* bits are set if the Control (wCx)/Data registers (wRn) contents have been updated since the last time this bit was cleared. This is typically used by the operating system (OS) to determine whether the registers need saving out to memory on a context switch (if the registers are unchanged then the values already stored in memory can be used instead). Note writing or loading this register does not set the CUP bit.

2.4 Arithmetic and Saturation Flags

The Wireless MMX™ technology provides a set of SIMD Flags to indicate the arithmetic status of the result and whether the result saturated. These flags can be combined and transferred back to the Intel® XScale™ microarchitecture CPSR.

2.4.1 Arithmetic Flags

The wCASF register contains arithmetic flags at each of the SIMD field positions. The information contained in the PSR therefore reflects the last arithmetic operation that set the SIMD arithmetic flags. Table 2-8 on page 13 shows the overall register format and accessibility for wCASF, the detailed format after each type of data operations is shown later in this section.

Table 2-8. Register Bitmap for wCASF

	Register 3													wCASF										Coproc 1									
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	SIMD Flags																															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	Access	Name	Description
31:0	Read/Write	SIMD Flags	Contains updated SIMD PSR flags following SIMD operation, format is either SIMD8,SIMD16,SIMD32 or SIMD64 as shown below.

After a byte-wide SIMD operation, the PSR would look like that shown in Table 2-9.

Table 2-9. Register Format of wCASF After a Byte Operation – SIMD8 PSR

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	N	Z	C	V	N	Z	C	V	N	Z	C	V	N	Z	C	V	N	Z	C	V	N	Z	C	V	N	Z	C	V	N	Z	C	V

	Name	Description
	N	Result in this byte field was **N**egative
	Z	Result in this byte field was **Z**ero
	C	Result in this byte field has a **C**arry out
	V	Result in this byte field o**V**erflowed

If a wider SIMD field operation is performed, the lower order unused bits are set to zero. Therefore, for a 16-bit SIMD operation, the PSR would contain the data shown in Table 2-10.

Table 2-10. Register Format for wCASF After a Half-Word Operation – SIMD16 PSR

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	N	Z	C	V	0	0	0	0	N	Z	C	V	0	0	0	0	N	Z	C	V	0	0	0	0	N	Z	C	V	0	0	0	0

	Name	Description
	N	Result in this half word field was **N**egative
	Z	Result in this half word field was **Z**ero
	C	Result in this half word field has a **C**arry out
	V	Result in this half word field o**V**erflowed

For 32-bit SIMD format the results are shown in Table 2-11.

Table 2-11. Register Format for wCASF After a Word Operation – SIMD32 PSR

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	N	Z	C	V	0	0	0	0	0	0	0	0	0	0	0	0	N	Z	C	V	0	0	0	0	0	0	0	0	0	0	0	0

	Name	Description
	N	Result in this word field was **N**egative
	Z	Result in this word field was **Z**ero
	C	Result in this word field has a **C**arry out
	V	Result in this word field o**V**erflowed

And finally, for a 64-bit operation (see Table 2-12).

Table 2-12. Register Bitmap for wCASF After a Double-Word Operation – SIMD64 PSR

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	N	Z	C	V	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	Name	Description
	N	Result in this word field was **N**egative
	Z	Result in this word field was **Z**ero
	C	Result in this word field has a **C**arry out
	V	Result in this word field o**V**erflowed

2.4.2 Saturation Flags

Certain instructions force the results to saturate to the maximum or minimum value rather than overflow (See Section A.18 as an example of an instruction which can saturate). The Saturation register indicates that saturation has occurred. The saturation flags indicate that saturation occurred at that particular byte, half word, or word position in the 64-bit double word. The *saturation* bit remains set until explicitly cleared by writing to the register. Additionally, a 16- or 32-bit SIMD operation does not clear the lower unused bits to zero. Instead, the lower unused bits remain at their previous value. The format is as shown in Table 2-13 on page 15.

Table 2-13. Register Bitmap for wCSSF – SIMD Saturation Flags

Register 2																wCSSF												Coproc 1							
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Reserved																								B7	B6	B5	B4	B3	B2	B1	B0
Reset	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	0	0	0	0	0	0	0	0

Bits	Access	Name	Description
31:7	N/A	Reserved	Write as Zero, Read as unknown.
7	Read/Write	B7	Saturation flag for Byte 7 or Half Word 3 or Word 1 or Double Word 0
6	Read/Write	B6	Saturation flag for Byte 6
5	Read/Write	B5	Saturation flag for Byte 5 or Half Word 2
4	Read/Write	B4	Saturation flag for Byte 4
3	Read/Write	B3	Saturation flag for Byte 3 or Half Word 1 or Word 0
2	Read/Write	B2	Saturation flag for Byte 2
1	Read/Write	B1	Saturation flag for Byte 1 or Half Word 0
0	Read/Write	B0	Saturation flag for Byte 0

- For byte SIMD operation, bits B0-B7 can be set if saturation occurs.
- For half-word SIMD operation, bits B1, B3, B5, and B7 can be set if saturation occurs.
- For word SIMD operation, bits B3 or B7 may be set.
- For 64-bit operation, only B7 is set on saturation.

Note: The saturation flags remain set until cleared by explicitly writing zeros into the register. The flags are cleared at reset.

2.5 Conditional Instruction Execution

The Wireless MMX™ technology programming model follows the ARM* conditional execution mechanism. The ARM* architecture specifies that instructions can be conditionally executed if the main ARM* arithmetic flags in the CPSR match the condition specified. All Wireless MMX™ technology data operations can be conditionally executed and have a “{Cond}” qualifier shown in the usage field of each instruction.

The conditional execution behavior is not shown explicitly in the operation of each instruction in this documentbut it is the same as defined in the ARM* V5 specification.

2.6 Group Conditional Execution

A unique feature of the Wireless MMX™ technology programming model is the ability to logically combine the SIMD arithmetic flags and use these in conjunction with the ARM* conditional execution mechanism to provide new execution predicates. This technique is known as **Group Conditional Execution**.

The arithmetic flags can be transferred to the ARM* Status register by performing an AND/OR operation across the SIMD PSR word (See TANDC/TORC), and sending the flags to the ARM* CPSR. Here, subsequent conditional instructions can use them (both Intel® XScale™ microarchitecture and coprocessor conditional instructions). Individual fields can also be extracted into ARM* registers using the TEXTRC instruction. Use of TANDC and TORC supports conditional instructions that operate on conditions such as “if *any* SIMD field is zero” (using TORC) and “if *all* SIMD fields are zero” (using TANDC). These new forms of conditions are known as **Group Condition Tests**.

2.6.1 Group Conditions

This list is the new group conditions tests that can be specified:

- If any field has overflowed
- If any field has not overflowed
- If any field is positive
- If any field is negative
- If any field is zero
- If any field is not zero
- If any field has a carry out
- If any field does not have a carry out
- If all fields have overflowed
- If all fields have not overflowed
- If all fields are positive
- If all fields are negative
- If all fields are zero
- If all fields are non-zero
- If all fields have a carry out
- If all fields do not have a carry out.

2.7 Enabling the Wireless MMX™ Technology Coprocessor

To use the Wireless MMX™ technology coprocessor, the coprocessors need to be enabled in the “Coprocessor Access Register” which is located in Register 15 of coprocessor 15. Both bits 0 and 1 of this register (corresponding to coprocessor 0 and 1) should be set to a 1 to enable access to the Wireless MMX™ technology coprocessor. If the Wireless MMX™ technology coprocessor is not enabled in this way, all Wireless MMX™ instructions will take the undefined instruction trap. See the *Intel® XScale™ Microarchitecture Megacell Users Guide* for more information about how to enable coprocessors.

Optimization Techniques 3

This chapter describes the creation of highly optimized code for the Wireless MMX™ technology coprocessor. This chapter provides the details of generic optimizations such as software pipelining and the multi-sample calculations as well as implementation specific optimizations associated with instruction latencies, instruction scheduling. In addition, data alignment and mixing Intel® XScale™ microarchitecture and Wireless MMX™ instructions are discussed. The application of these optimization techniques can lead to substantial increases in the execution speed for target code sequences.

Note: The implementation specific optimizations relating to instruction scheduling and instruction latencies are designed for use on Wireless MMX™ technology 1.0 implementations.

3.1 Optimizing with Multiple Pipelines

In order to write fully optimized Wireless MMX™ technology code, it is necessary to know the latency properties of the instructions. The instruction result latency defines the number of cycles you should wait after issuing an operation before using the result and the issue latency is the number of cycle it takes an instruction to leave the register file. On Wireless MMX™ technology violating the latency will not cause the program to function incorrectly but means that a stall is introduced until the latency is satisfied

After the register file, there are four concurrent pipelines to which an instruction can be dispatched. An instruction can be issued to a pipeline if the resource is available and there are no unresolved data dependencies. For example, a load instruction that uses the Memory pipeline can be issued while a multiply instruction is completing in the Multiply pipeline (assuming there are no data hazards.)

The performance effect of resource contention can be quantified by examining the delay taken for a particular instruction to start execution and release the resource. The definition of “release the resource” is the point at which a resource can accept another instruction. (Note: the resource may still be processing the previous instruction further down its internal pipeline.) A delay of one clock cycle indicates that the resource is available immediately to the next instruction. A delay greater than one clock cycle stalls the next instruction if the same resource is required. The following sections examine the resource-usage delays for the four pipelines, and how these map onto the instruction set

3.2 Issue Cycle and Result Latency

The issue cycle and result latency of all the Wireless MMX™ instructions is shown in Table 3-1 on page 18. In this table, the issue cycle is the number of cycles that an instruction takes to leave the register file. The **result latency** is the number of cycles required to calculate the result and make it available to the bypassing logic. A result latency of *1* indicates that the value is available immediately to the next instruction. This table shows the best case result latency that can be degraded by data or resource hazards.

Table 3-1. Issue Cycle and Result Latency of the Wireless MMX™ Instructions

Instructions	Issue Cycle	Result Latency
WADD, WSUB, WAVG2	1	1
WCMPGT, WCMPGT	1	2
WAND, WANDN, WOR, WXOR	1	1
WMAX, WMIN	1	2
WSAD, WACC	1	1
WMUL, WMADD, WMAC, TMIA	1	2
TMIAPH, TMIAxy	1	1
WSLL, WSRA, WSRL, WROR	1	1
WPACK	1	2
WUNPCKEH, WUNPCKEL WUNPCKIH, WUNPCKIL	1	1
WALIGNI, WALIGNR, WSHUFH	1	1
TANDC, TORC, TEXTRC	1	1
TEXTRM, TMOVMSK	1	2
TMCR	1	3
TMCRR	1	1
TMRC	1	2
TMRRC	1	3
TINSTR, TBCST	1	1
WLDR (BHW) to main regfile	1	4 (3)†, ††
WLDRW to control regfile	1	4††
WSTR	1	na††

† WLDRD is 4 cycles WLDR<B,H,W> is 3 cycles

†† Base address register update for WLDR and WSTR is the same as the core load/ store operation.

3.3 Performance Hazards

The basic performance of the system can be effected by stalls caused by data or resource hazards. This section describes the factors effecting each type of hazard and the implications for performance.

3.3.1 Data Hazards

A data hazard occurs when an instruction requires data that cannot be provided by the register file or the data-forwarding mechanism or if two instructions update the same destination register in an out of order fashion. The first hazard is termed as *Read-After-Write (RAW)* and the second hazard is termed as *Write-After-Write (WAW)*. The processing of the new instruction is stalled until the data becomes available for RAW hazards, and until it can be guaranteed that the new instruction will update the register file after previous instruction has updated the same destination register for

WAW hazards. The Wireless MMX™ technology device contains a bypassing mechanism for ensuring that data and different stages of the pipeline are forwarded to the correct instructions. There are, however, certain combinations of instructions where it is not possible to forward directly between instructions in the Wireless MMX™ technology 1.0 implementation.

The result latency shown in Table 3-1 on page 18 and best-case result latency are generally achievable. However there are certain instruction combinations where these result latencies do not hold because not all combinations of bypassing logic exist in the hardware, and some instructions require more time to calculate the result when certain qualifiers are specified. The following list describes the data hazards for the Wireless MMX™ technology 1.0 implementation:

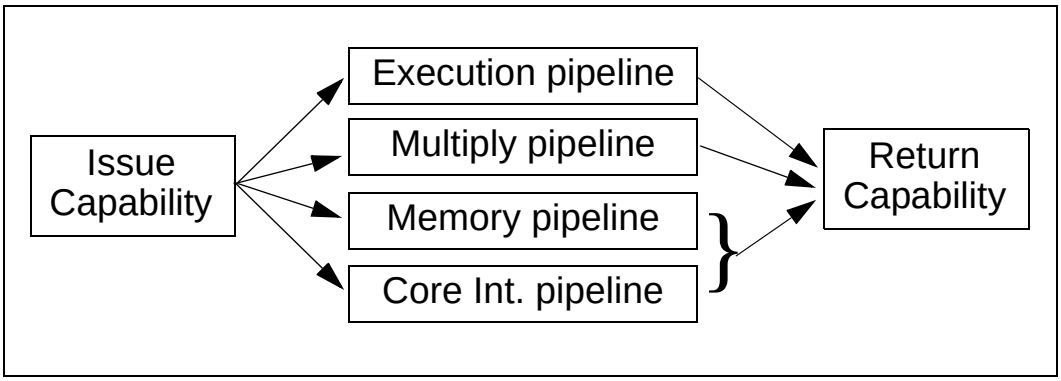
- When saturation is specified for WADD or WSUB, the result latency is increased to two cycles.
- The destination register (accumulator) for certain multiplier instructions (WMAC, WSAD, TMIA, TMIAPH, TMIAxy) can be forwarded for accumulation to the same destination register only. If the destination register results are needed by another instruction as source operands, there will be an additional result latency as the result is available from the regular forwarding paths, external to the multiplier.
  - — For WMAC, WACC, WMUL, and WMADD there is an additional result latency of 2 cycles.
  - — For WSAD, TMIA, TMIAPH, and TMIAxy there is an additional result latency of 3 cycles.
- If an instruction is updating a destination register from the multiply pipeline, the next instruction in the execute, memory or core interface pipelines updating the same destination register is stalled until it can guarantee that the following instruction will update the register file after the previous instruction in the multiply pipe has updated the register file
- If an instruction is updating a destination register from the memory pipeline, the next instruction updating the same destination register is stalled until it can be guaranteed that the next instruction will update the register file after the previous instruction in the memory pipe, has updated the register file.
- If the Intel® XScale™ microarchitecture MAC unit is in use, the resulting latency of a TMRC, TMRRC, and TEXRM increases accordingly.

3.3.2 Resource Hazard

A **resource hazard** is caused when an instruction requires a resource that is already in use. When this condition is detected, the processing of the new instruction is stalled at the register file stage.

Figure 3-1 on page 20 shows a high-level representation of the operation of the Wireless MMX™ unit.

Figure 3-1. Instruction Issue and Return Capability



3.3.2.1 Execution Pipeline

An instruction can be accepted into the Execution pipeline when the first stage of the pipeline is empty. Table 3-2 shows the instructions that execute in the main Execution pipeline. All these instructions have a resource usage delay of one clock cycle. Therefore, the Execution pipeline will always be available to the next instruction.

Table 3-2. Resource Availability Delay for the Execution Pipeline

Instructions	Delay (Clocks)
WADD, WSUB	1
WCMPEQ, WCMPGT	1
WAND,WANDN,WOR,WXOR	1
WAVG2	1
WMAX, WMIN	1
WSAD	1†
WSLL, WSRA, WSRL, WROR	1
WPACK	1
WUNPCKEH, WUNPCKEL WUNPCKEL, WUNPCKIL	1
WALIGNI, WALIGNR, WSHUFH	1
TMIA	1†
TMIAPH	1†
TMIAxy	1†
TMCR, TMCRR	1
TINSR, TBCST, TANDC, TORC	1
TEXTRC	1

† The WSAD, TMIA, TMIAPH, TMIAxy execute in both the main Execution pipeline and the Multiplier pipeline. They execute for one cycle in the Execution pipeline and the rest in the Multiplier pipeline. See Section 3.3.2.5 for more details.

3.3.2.2 Multiply Pipeline

Instructions issued to the Multiply pipeline may take up to two cycles before another instruction can be issued to the pipeline. The instructions in the multiply pipe can be categorized into 4 classes shown in Table 3-3. The resource-availability delay for the instructions that are mapped onto the multiplier pipeline depend upon the class of the multiply instruction that subsequently wants to use the multiply resource. These delays are shown below in Table 3-4. For example if a TMIA instruction is followed by a TMIAPH (class3) instruction, then the TMIAPH sees a resource availability of 2 cycles.

Table 3-3. Multiply pipe instruction classes

Instructions	Class
WACC	1
WMAC, WMUL, WMADD	2
WSAD, TMIAPH, TMIAxy	3
TMIA	4

Table 3-4. Resource Availability Delay for the Multiplier Pipeline

Instruc-tions	Delay(Clocks) for a subsequent class 1 multiply pipe instruction	Delay(Clocks) for a subsequent class 2 multiply pipe instruction	Delay(Clocks) for a subsequent class 3 multiply pipe instruction	Delay(Clocks) for a subsequent class 4 multiply pipe instruction
WSAD	2	2	1	1
WACC	1	1	1	1
WMUL	2	2	1	1
WMADD	2	2	1	1
WMAC	2	2	1	1
TMIA	3	3	2	2
TMIAPH	2	2	1	1
TMIAxy	2	2	1	1

WSAD, TMIA, TMIAxy, TMIAPH execute in both the Main Execution pipeline and the Multiplier pipeline. See Section 3.3.2.5 for more details

3.3.2.3 Memory Control Pipeline

The Memory Control pipeline is responsible for coordinating the Load/Store activity with the Intel® XScale™ microarchitecture. The external interface to memory is 32-bits so the 64-bit Load/Store issued by the Wireless MMX™ technology device are sequenced as two 32-bit Load/Stores to memory. This is transparent to end users and is already factored into the result latencies show in Table 3-1 on page 18. After the Wireless MMX™ technology device issues the 64-bit memory transaction, it must buffer the data until the two 32-bit half transactions are complete. Currently, there are two 64-bit buffer slots for Load operations and one 64-bit buffer slot available for Store transactions. If the memory buffer is currently empty, the Memory pipeline resource-availability delay is only one clock. However, if the buffer is currently full due to a sequence of memory transactions, the next instruction must wait for space in the buffer. The resource availability delay in this case is two cycles. This is summarized in Table 3-5 on page 22.

Table 3-5. Resource Availability Delay for the Memory Pipeline

Instructions	Delay(Clocks)	Condition
WLDRD	1	Two loads not already outstanding
WSTRD	2	
WLDRD	3+M	Two loads already outstanding (M is delay for main memory if cache miss)

The buffering in the Memory pipeline allows two Load transactions to be issued sequentially without incurring a penalty (stall). More than two outstanding Load transactions causes a stall and loss in performance.

3.3.2.4 Coprocessor Interface Pipeline

The coprocessor Interface pipeline also contains buffering to allow multiple outstanding MRC/ MRRC operations. The coprocessor interface pipeline can continue to accept MRC and MRRC instructions every cycle until its buffers are full. Currently there is sufficient storage in the buffer for either four MRC data values (32-bit) or two MRRC data values (64-bit). Table 3-6 shows a summary of the resource availability delay for the coprocessor interface.

Table 3-6. Resource Availability Delay for the Coprocessor Interface Pipeline

Instructions	Delay(Clocks)	Condition
TMRC, TMRRC	1	Buffer Empty
TMRC, TMRRC	2	Buffer Full

There is also an interaction between TMRC/TMRRC and any instructions in the core that utilize the MAC unit of the core. For optimum performance, the MAC unit in the core should not be used adjacent to TMRC instructions as they both share the route back to the core register file.

3.3.2.5 Multiple Pipeline Instructions

The WSAD, TMIA, TMIAPH and TMIAxy instructions execute in both the main execution pipeline and the multiplier pipeline. The instruction executes one cycle in the execution pipeline and the rest in the multiplier pipeline. The WSAD, TMIA, TMIAPH, TMIAxy instructions will always issue without stalls to the Execution pipeline (see Section). The availability of the multiplier pipeline depends on a previous instruction that was using the multiply resource. If the previous instruction was a TMIA, there is an effective resource availability of two cycles.

3.4 Instruction Scheduling

Figure 3-2 on page 23 summarizes how the instruction execution maps onto hardware resource. The figure shows an ***X*** where an instruction executes in a particular pipeline stage for 1 cycle. The diagram shows an ***X'*** where it occupies the pipeline stage for multiple cycles. An ***R*** in the diagram indicates the position in the pipeline where the result from that instruction becomes available for forwarding to other instructions and an ***R'*** indicates that the result is available for limited forwarding inside the multiplier. This summary used in conjunction with the latency information and scheduling detail described in this section lets you write optimal Wireless MMX™ technology code sequences.

Figure 3-2. Instruction Schedule Summary

Trace of ALU Operations :

Instructions	ID	RF	X1	X2	XWB
WADD	X	X	X/R	R	R
WSUB	X	X	X/R	R	R
WCMPEQ	X	X	X	R	R
WCMPGT	X	X	X	R	R
WAND	X	X	R	R	R
WANDN	X	X	R	R	R
WOR	X	X	R	R	R
WXOR	X	X	R	R	R
WAVG2	X	X	R	R	R
WMAX	X	X	X	R	R
WMIN	X	X	X	R	R
WSLL	X	X	R	R	R
WSRA	X	X	R	R	R
WSRL	X	X	R	R	R
WROR	X	X	R	R	R
WPACK	X	X	X	R	R
WUNPCKEH	X	X	R	R	R
WUNPCKEL	X	X	R	R	R
WUNPCKIH	X	X	R	R	R
WUNPCKIL	X	X	R	R	R
WALIGNI	X	X	R	R	R
WALIGNR	X	X	R	R	R
WSHUF	X	X	R	R	R

Trace of CIU Operations :

Instructions	ID	RF	X1	X2	XWB
TANDC	X	X	R	R	R
TORC	X	X	R	R	R
TEXTRC	X	X	R	R	R
TMOVSK	X	X	X	R	R
TINSR	X	X	R	R	R
TBCST	X	X	R	R	R
TMRRC	X	X	X	X	R
TMRC	X	X	X	R	R
TMCR	X	X	X	X	X
TMCRR	X	X	R	R	R
TEXTRM	X	X	X	R	R

Trace of load and store instructions :

Instructions	ID	RF	X1	D1	D2	DWB
WLDR	X	X	X	X	R	R
WSTR	X	X	X			

Trace of MAU instructions :

Instructions	ID	RF	X1	X2	XWB	M1	M2	M3	MWB
WSAD	X	X	X			X	X/R’	R	R
WACC	X	X				X	X/R’	R	R
WMUL	X	X				X/X’	X/R’	R	R
WMADD	X	X				X/X’	X/R’	R	R
WMAC	X	X				X/X’	X/R’	R	R
TMIA	X	X	X			X/X’	X/R’	R	R
TMIAPH	X	X	X			X	X/R’	R	R
TMIAxy	X	X	X			X	X/R’	R	R

An instruction in MAC can forward to another instruction within MAC. (in M2)

Forwarding to other functional unit takes place in M3

X = Execute

R = Ready for forwarding

R’= Forwarding within the functional unit

X’= Spends additional cycle

3.4.1 Increasing Load Throughput on Wireless MMX™ Technology

The same load transactions constraints in Intel® XScale™ microarchitecture are in the Wireless MMX™ technology. The considerations reviewed using the Intel® XScale™ microarchitecture instructions are re-illustrated in this section using the Wireless MMX™ instruction set. The primary observations with load transactions are:

- The buffering in the Memory pipeline allows two Load Double transactions to be outstanding without incurring a penalty (stall).
- Back-to-Back WLDRD instructions will incur a stall, Back-to-Back WLDR(BHW) instructions will not incur a stall
- The WLDRD requires 4 cycles to return the DWORD assuming a cache hit, Back-to-Back WLDR (BHW) require 3 cycles to return the data.

The overhead on issuing load transactions can be minimized by instruction scheduling and load pipelining. In most cases it is straightforward to interleave other operation to avoid the penalty with Back-to-Back LDRD instructions. In the following code sequence three WLDRD instructions are issued back-to-back incurring a stall on the second and third instruction.

```
WLDRD wR3,[r4],#8
WLDRD wR5,[r4],#8       @ STALL
WLDRD wR4,[r4],#8       @ STALL
WADDB wR0,wR1,wR2
WADDB wR0,wR0,wR6
WADDB wR0,wR0,wR7
```

The same code sequence is reorganized to avoid a back-to-back issue of WLDRD instructions.

```
WLDRD wR3,[r4],#8
WADDB wR0,wR1,wR2
WLDRD wR4,[r4],#8
WADDB wR0,wR0,wR6
WLDRD wR5,[r4],#8
WADDB wR0,wR0,wR7
```

Always try to separate 3 consecutive WLDRD instructions so that only 2 are outstanding at any one time and the loads are always interleaved with other instructions.

```
WLDRD wR0, [r2] , #8
WZERO wR15
WLDRD wR1, [r4] , #8
SUBS  r3, r3, #8
WLDRD wR3, [r4] , #8
```

Always try to interleave additional operations between the load instruction and the instruction which will first use the data.

```
WLDRD wR0, [r2] , #8
WZERO wR15
WLDRD wR1, [r4] , #8
SUBS  r3, r3, #8
WLDRD wR3, [r4] , #8
SUBS  r4, r4, #1
WMACS wR15, wR1, wR0
```

3.4.2 Scheduling the WMAC Instructions

The issue latency of the WMAC instruction is one cycle and the result and resource latency is two cycles. The second WMAC instruction in the following example stalls for one cycle due to the two cycle resource latency.

```
WMACS wR0, wR2, wR3
WMACS wR1, wR4, wR5
```

The WADD instruction in the following example stalls for three cycles due to the dependency of the source operands. (refer to section 6.3.1 Data Hazards)

```
WMACS wR0, wR2, wR3
WADDHSS wR1, wR0, wR2
```

It is often possible to interleave instructions and effectively overlap their execution with multi-cycle instructions that utilize the multiply pipe-line. The 2-cycle WMAC instruction may be easily interleaved with operations which do not utilize the same resources:

```
WMACS wR14, wR2, wR3
WLDRD wR3, [r4] , #8
WMACS wR15, wR1, wR0
WALIGNI wR5, wR6, wR7, #4
WMACS wR15, wR5, wR0
WLDRD wR0, [r3], #8
```

In the above example, the WLDRD and WALIGNI instructions do not incur a stall since they are utilizing the memory and execution pipelines respectively and there are no data dependencies.

When utilizing both Intel® XScale™ microarchitecture and Wireless MMX™ technology execution resources, it is also possible to overlap the multi-cycle instructions. The ADD instruction in the following example executes with no stalls.

```
WMACS wR14, wR1, wR2
ADD   R1, R2, R3
```

Refer to the *Intel® XScaleTM Microarchitecture Programmers Reference Manual* for more information on instruction latencies for various multiply instructions. The multiply instructions should be scheduled taking into consideration their respective instruction latencies.

3.4.3 Scheduling the TMIA Instruction

The issue latency of the TMIA instruction is one cycle and the result and resource latency are two cycles. The second TMIA instruction in the following example stalls for one cycle due to the two cycle resource latency.

```
TMIA wR0, r2, r3
TMIA wR1, r4, r5
```

The WADD instruction in the following example stalls for four cycles due to the dependency of the source operands. (refer to section 6.3.1 Data Hazards)

```
TMIA    wR0, r2, r3
WADDHUS wR1, wR0, wR2
```

Refer to the *Intel® XScaleTM Microarchitecture Programmers Reference Manual* for more information on instruction latencies for various multiply instructions. The multiply instructions should be scheduled taking into consideration their respective instruction latencies.

3.4.4 Scheduling the WMUL and WMADD Instructions

The issue latency of the WMUL and WMADD instructions is one cycle and the result and resource latency are two cycles. The second WMUL instruction in the following example stalls for one cycle due to the two cycle resource latency.

```
WMULUM wR0, wR1, wR2
WMULSL wR3, wR4, wR5
```

The WADD instruction in the following example stalls for three cycles due to the dependency of the source operands. (refer to section 6.3.1 Data Hazards)

```
WMULUM wR0, wR1, wR2
WADDHUS wR1, wR0, wR2
```

3.4.5 Use of Auxiliary Registers

The auxiliary registers are designed to hold constants that are invariant across the life-time of an inner loop calculation. For this reason values loaded into the auxiliary registers are not forwarded to data operations. The intended use of the registers is that the shift or alignment offset is loaded into a wCGRn register before the main loop is entered and then the shift or alignment offset is used repeatedly inside the loop without it changing. If the value in a wCGRx register is changed and an instruction immediately afterwards tries to use the loaded value then the coprocessor stalls until the loaded value has reached the control register file.

3.5 SIMD Optimization Techniques

The Single Instruction Multiple Data, (SIMD) architecture provided by the Wireless MMX™ technology enables us to exploit the inherent parallelism found in the wide domain of multimedia and communication applications. The most time-consuming code sequences have certain characteristics in common:

- Operations are performed on small native data types (8-bit pixels, 16-bit voice, 32-bit audio)
- Regular and recurring memory access patterns, usually data independent
- Localized, recurring computations performed on the data
- Compute-intensive processing

The following sections illustrate how the rules for writing fast sequences of Wireless MMX™ instructions that can be applied to the optimization of short loops.

3.5.1 Software Pipelining

Software pipelining or loop unrolling is a well known optimization technique where multiple calculations are executed with each loop iteration. The disadvantages of applying this technique include: Increases in code size for critical loops, and restrictions on the minimum and multiples of taps or samples

The obvious advantage is in reduced cycle consumption. Overhead from loop exit testing may be reduced, load-use stalls may be minimized (and in some cases eliminated completely), and instruction scheduling opportunities may be created and exploited.

To illustrate the need for software pipe-lining, lets consider a key kernel of Wireless MMX™ code that is central to many signal-processing algorithms, the real block Finite-Impulse-Response (FIR) filter. A real block FIR filter operates on two real vectors ***c(i)*** and ***x(i)*** and produces and output vector ***y(n)***. The vectors are represented for Wireless MMX™ technology programming as arrays of 16-bit integers of some length N. The real FIR filter is represented by the equation,

$$y(n) = \sum_{i=0}^{L-1} c_i \cdot x(n-i), \qquad \vee\, 0 \le n \le N-1$$

or, in C-code,

```
for (i = 0; i < N; i++) {
   s = 0;
   for (j = 0; j < T; j++) {
   s += a[j]*x[i-j]);
   }
   y[i] = round (s);
}
```

The WMAC instruction is utilized for this calculation and provides for four parallel 16-bit by 16-bit multiplications with accumulation. The first level of unrolling is a direct function of the four-way SIMD instruction that is used to implement the filter.

The C-code for the real block FIR filter is re-written to illustrate that 4-taps are computed for each loop iteration (refer to Example 3-1).

Example 3-1. C-Code for FIR Filter with Loop-Unrolling for 8-taps per iteration.

```
for (i = 0; i < N; i++) {
   s0= 0;
   for (j = 0; j < T/4; j++4) {
      s0 += a[j]*x[i+j];
      s0 += a[j+1]*x[i+j+1];
      s0 += a[j+2]*x[i+j+2];
      s0 += a[j+3]*x[i+j+3];
   }
   y[i] = round (s0);
}
```

The direct assembly code implementation of the inner loop illustrates clearly that optimum execution has not been accomplished. In Example 3-2 the code sequence has several undesirable stalls. The back-to-back LDRD instructions incur a 1 cycle stall, the load-to-use penalty incurs a 3 cycle stall. In addition, the loop overhead is high with 2 cycles being consumed for every 4-taps.

Example 3-2. Assembly Code for FIR Filter for 4-taps per iteration.

```
; Pointers r0 -> val , r1 -> pResult, r2 -> pTapsQ15 r3 -> tapsLen

   WZERO wR15
Loop_Begin:
   WLDRD wR0, [r0], #8
   WLDRD wR1, [r1], #8
   WMACS wR2, wR1, wR0
   SUBS r3, r3, #4
   BNE Loop_Begin
```

The parallelism of the filter may be exposed further by unrolling the loop to provide for 8-taps per iteration. In the following code sequence, the loop has been unrolled once allowing several load-to-use stalls to be eliminated. The loop overhead has also been further amortized reducing it from two cycles for every four taps to 2 cycles for every 8 taps (see Example 3-3 on page 29). There is still a single load-to-use stall present between the second WLDRD instruction and the second WMACS instruction within the inner loop.

Example 3-3. FIR Filter with Loop-Unrolling for 8-taps per iteration.

```
    @Pointers r0 -> val , r1 -> pResult, r2 -> pTapsQ15 r3 -> tapsLen

       WLDRD     wR0, [r0], #8   @ Load first 4 input samples
       WLDRD     wR1, [r2], #8   @ Load first 4 coefficients
       WZERO     wR15
       WLDRD     wR2, [r0], #8   @ Load next 4 input samples
Loop_Begin:
       WLDRD     wR3, [r2], #8   @ Load next 4 coefficients
       WMACS     wR15,wR0,wR1
       WLDRDNE   wR0, [r0], #8   @ Load 4 input samples
       SUBS      r3,r3,#8
       WLDRDHS   wR2, [r2], #8
       WMACS     wr8, wr1, wr3
       WLDRDHS   wr1, [r1], #8
       BHS       Loop_Begin
```

3.5.1.1 General Remarks on Software Pipelining

In the example for the real block FIR filter, two copies of the basic sequence of code were interleaved eliminating all but one of the stalls. The throughput for the sequence went from 9 cycles for every four taps to 8 cycles for every 8 taps. This corresponds to a throughput of 1 cycle per tap represents a 2X throughput improvement.

It is useful to define a metric to describe the number of copies of a basic sequence of instructions which need to be interleaved in order to remove all stalls. We can call this the interleave factor, k. The real block FIR filter requires k=2 to eliminate all possible stalls primarily because it is a small sequence which must take into account the long load-to-use latency. In practice, k=2 is sufficient for most loops encountered in real applications. This is fortunate because each interleaving requires its own set of temporary registers and with some algorithms interleaving with k=3 is not possible. A good rule of thumb is to try k=2 first, as it will usually be the right choice.

3.5.2 Multi-Sample Technique

The Multi-sample optimization technique provides for calculating multiple outputs with each loop iteration similar to loop unrolling. The disadvantages of applying this technique include, increases in code size for critical loops. Restrictions on the minimum and multiples of taps or samples are also imposed. The obvious advantage is in reduced cycle consumption.

- Memory bandwidth is reduced by data re-use.
- Load-to-Use stalls may be easily eliminated with scheduling.
- Multi-cycles may be interleaved with other instructions

The C-code for the N-Sample, T-Tap block FIR filter is also used to illustrate the multi-sample technique.

Example 3-4. C-Code for FIR Filter with Multiple Samples for 8-taps per iteration

```
for (i = 0; i < N; i++4) {
   s0=s1=s2=s3=0;
   for (j = 0; j < T/4; j++4) {
      s0 += a[j]*x[i-j];
      s1 += a[j]*x[i-j+1];
      s2 += a[j]*x[i-j+2];
      s3 += a[j]*x[i-j+3]);
   }
   y[i] = round (s0);
   y[i+1] = round (s1);
   y[i+2] = round (s2);
   y[i+3] = round (s3);
}
```

In the inner loop, we are calculating four output samples using the adjacent data samples *x(n-i)*, *x(n-i+1)*, *x(n-i+2)* and *x(n-i+3)*. The output samples *y(n)*, *y(n+1)*, *y(n+2)*, and *y(n+3)* are assigned to four 64-bit Wireless MMX™ technology registers. In order to obtain near ideal throughput, the inner loop is unrolled to provide for 8 taps for each of the four output samples per loops iteration.

Example 3-5. Assembly for FIR Filter with Multiple Samples and 8-taps per iteration

```
    ; ** Update pointers,
Outer_Loop:
    ; ** Update pointers,zero accumulators and prime the loop with DWORD loads

   WLDRD    wR1, [R1], #8    ; Load first 4 input samples
   WZERO    wR15
   WLDRD    wR0, [R1], #8    ; Load even groups of 4
                             ; input samples
   WZERO    wR14
   WLDRD    wR8, [R2], #8; Load first 4 coefficients
   WZERO    wR13
   WZERO    wR12

InnerLoop:
   ; ** Executes 8-Taps for each four outputs samples
   ; y(n),y(n+1), y(n+2),y(n+3)

   SUBS     R0  ,R0  , #8       ; Decrement loop counter
   WMACS    wR15,wR8 , wR0      ; y(n)+=
   WALIGNI  wR3 ,wR1 , wR0, #2
   WMACS    wR14,wR8 , wR3      ; y(n+1) +=
   WALIGNI  wR3 ,wR1 , wR0, #4
```

```
        WMACS    wR13,wR8 , wR3      ; y(n+2) +=
        WLDRD    wR0, [R1], #8       ;next 4 input samples
        WALIGNI  wR3 ,wR1 , wR0, #6
        WLDRD    wR9, [R2], #8       ; odd groups of 4 coeff.
        WMACS    wR12,wR8 , wR3      ; y(n+3) +=
        WALIGNI  wR3 ,wR0 , wR1, #2
        WALIGNI  wR4 ,wR0 , wR1, #4
        WMACS    wR15,wR9 , wR1      ; y(n) +=
        WALIGNI  wR5 ,wR0 , wR1, #6
        WMACS    wR14,wR9 , wR3      ; y(n+1) +=
        WLDRD    wR1, [R1], #8       ; even groups of 4 inputs
        WMACS    wR13,wR9 , wR4      ; y(n+2) +=
        WLDRD    wR8, [R2], #8       ; even groups of 4 coeff.
        WMACS    wR12,wR8 , wR5      ; y(n+3) +=
        BNE      Inner_Loop

        ; ** Outer loop code calculates the last four taps for
        ; y(n), y(n+1), y(n+2), y(n+3)**
        ; ** Store results

BNE Outer_Loop
```

3.5.2.1 General Remarks on Multi-sample Technique

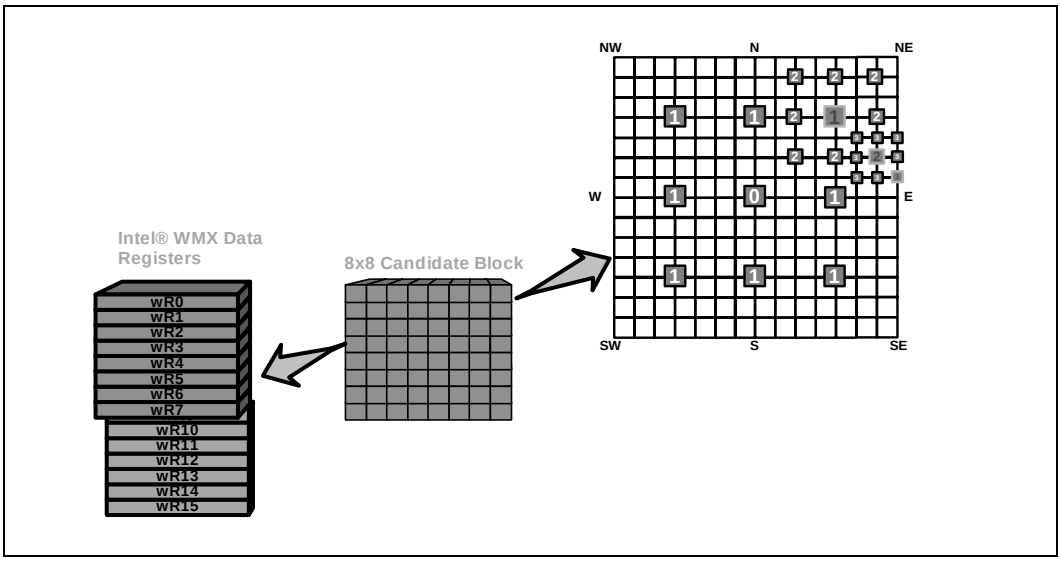
In the example for the real block FIR filter, four outputs are computed simultaneously in the same inner loop. This has allowed the re-use of coefficients and sample data loaded into the register for computation of the first output to be used for the computation of the next three outputs. The interleave factor is set at k=2, which results in the elimination of load-to-use stalls. The throughput for the sequence is 20 cycles for every 32 taps, or 0.625 cycles per tap. This represents near ideal utilization of the execution resources.

The multi-sample technique may be applied whenever the same data is being utilized for multiple calculations. The large register file on Wireless MMX™ technology facilitates this approach and a number of variations are possible.

3.5.3 Register File Usage

With the large register file of the Wireless MMX™ technology it is possible to store large data structures in the register file and reduce memory load traffic. For example, in Figure 3-3 on page 32 the register file is used to store the 8x8 pixel macroblock during a video encode motion search. Even with the macroblock loaded into the register file there are still eight 64-bit registers still available for use in the motion search.

Figure 3-3. Loading an 8x8 Macro Block in the Register File During Video Motion Estimation



3.5.4 Data Alignment Techniques

The exploitation of the data parallelism present in multimedia algorithms is accomplished by executing the same operation on different elements in parallel. This is accomplished by packing several data elements into a single register and using the packed data instructions provided by the Wireless MMX™ technology.

An important guideline for achieving optimum performance is always to align memory references. This means that an N-byte memory read or write should always be on an N-byte boundary. In some applications it is easy to align data so that all of the reads and writes are aligned. In other cases it is more difficult because an algorithm naturally reads data in a misaligned fashion. A couple of examples of this include the single-sample FIR and video motion estimation.

The Wireless MMX™ technology provides a mechanism for reducing the overhead associated with the classes of algorithms which require data to be accessed on 32-bit, 16-bit, or 8-bit boundaries. The WALIGNI instruction is useful when the sequence of alignment is known beforehand as with the single-sample FIR filter. The WALIGNR instruction is useful when sequence of alignments are calculated when the algorithm executes as with the fast motion search algorithms used in video compression. Both of these instructions operate on register pairs which may be effectively ping-ponged with alternate loads reducing the alignment overhead significantly.

The following code sequence illustrates how to set-up an unaligned array access. The procedure involves loading one of the general purpose registers on Wireless MMX™ technology with the lower 3-bits of the address pointer and then clearing the LSB's so that future accesses to the array are on a 64-bit boundary.

```
;r0 -> pointer to misaligned array.
AND r7,r0,#7
TMCR wCGR1, r7
BIC r1, r1, #7 @ r1 psrc 64 bit aligned
```

Following the initial setup for alignment, the data can now be accessed, aligned, and presented to the execution resources.

```
WLDRD wR0, [r0]
WLDRD wR1, [r0,#8]
WALIGNR1 wR4, wR0, wR1
```

In the above code sequence it is also necessary to interleave additional operations to avoid the back-to-back WLDRD and load-to-use penalties

Porting Existing Code 4

The re-use of existing MMX™ technology and integer SSE code is encouraged since algorithm mapping to Wireless MMX™ technology may be significantly accelerated. This section provides guidelines and examples which highlights some of the differences to watch for when porting existing code sequences.

4.1 Instruction set Mapping

Wireless MMX™ technology provides equivalent functionality to all the Intel® MMX™ instructions and integer instructions from SSE instruction group. This group of instructions is targeted to accelerate software performance in applications that use video and 3D rasterization, video encoding, and decoding on blocks of data, as well as a wide variety of signal processing algorithms. Table 4-1summarizes the MMX™ technology instructions mapping for Wireless MMX™ technology and the types of operations they perform. The Wireless MMX™ technology mapping of the integer SSE instructions are summarized in Table 4-2 on page 36.

Table 4-1. Providing the functionality of Wireless MMX™ technology

Wireless MMX™ technology	MMX™ technology	Options	Description
WADD{b/h/w} WSUB{b/h/w}	PADD{b/w/d} PSUB{b/w/d}	Wraparound, and saturate.	Add or subtract eight bytes, four 16-bit half-words, or two 32-bit words
WCMPEQ{b/h/w} WCMPGT{b/h/w}	PCMPEQ{b/w/d} PCMPGT{b/w/d}	Equal to or greater than	Compare eight bytes, four 16-bit half-words, or two 32-bit elements in parallel. Result is mask of 1's if true or 0's if false.
WMUL{L} WMUL{H}	PMULLW PMULHW	Result is high or low order bits	Multiply four signed 16-bit words in parallel. Low-order or high order 16-bits of 32-bit result are produced.
WMADD	PMADDWD	Half-word to word conversion	Multiply add four 16-bit words in parallel and add
WSRA{h/w} WSLL{h/w/d} WSRL{h/w/d}	PSRA{w/d} PSLL{w/d/q} PSRL{w/d/q}	Shift count in data or auxiliary register	Shift arithmetic right, logical right and left of 4 half-words, 2 words, or the full 64-bit double-word, in parallel
WUNPCKIL{b/h/w} WUNPCKIH{b/h/w}	PUNPCKL{bw/wd/dq} PUNPCKH{bw/wd/dq}		Interleave and merge eight bytes, four 16-bit halfwords, words, or two 32-bit words
WPACK{h/w}{SS} WPACK{h/w}{US}	PACKSS{wb/dw} PACKUS{wb/dw}	Always saturate	Pack double-words to words or words to bytes
WAND, WANDN WOR, WXOR	PAND POR		64-bit logical operations
WMOV/WLDR	MOV{d/q}		Move or load a 64-bit register.

Table 4-2. Providing the Functionality of SSE Technology

Wireless MMX™ technology	MMX™ technology	Options	Comments
WMAX{b,h,w} WMIN{b,h,w}	PMAX{b,w} PMIN{b,w}	Signed or unsigned operands	Vector maximum/minimum selection between elements in two registers.
TMOVMSK{b,h,w}	PMOVMSKB	Wireless MMX™ technology provides 8-bit and 32-bit options	Transfers the most significant bit of each SIMD field of a Wireless MMX™ technology register to an ARM* register
WAVG2{b,h}	PAVG{b,w}	Rounding option by +1 or +2.	Two element average on eight unsigned vectors of 8-or 16-bit data
TINSR{b,h,w}	PINSRW	Wireless MMX™ technology provides 8-bit and 32-bit options	Transfers and inserts 8-/16-/32-bit data from an ARM* register to a field in a Wireless MMX™ technology data register
TEXTRM{b,h,w}	PEXTRW	Wireless MMX™ technology provides 8-bit and 32-bit options	Extracts and transfers an 8-/16-/32-bit field from a Wireless MMX™ technology data register to an ARM* register
WROR{h,w,d}	PROR{w,d,q}	Shift count in data or auxiliary register	Rotate right of 4 half-words, 2 words, or the full 64-bit double-word, in parallel
WSAD{B,W}	PSAD{b,w}		Performs sum-of-absolute-differences on unsigned vectors of 8- or 16-bit data.
WSHUFH	PSHUFW		Shuffles 16-bit data values specified by an 8-bit immediate

4.1.1 Intel® XScale™ microarchitecture Media Instructions

The Intel® 80200 processor introduced a DSP coprocessor to the architecture for the purpose of increasing performance and the precision of audio processing algorithms. This coprocessor introduced several new instructions which use a 40-bit accumulator. The Wireless MMX™ technology mapping of the multiply with internal accumulate instruction provides the same functionality but extends the precision by using a 64-bit accumulator. The MIA instruction group all use the ARM* core registers for source operands and a Wireless MMX™ technology data register for the accumulation. The Wireless MMX™ technology mapping of the multiply accumulate instructions are summarized in Table 4-3 on page 37.

Table 4-3. Wireless MMX™ technology Mapping of Intel® XScale™ microarchitecture Instructions

Wireless MMX™ technology	Intel® XScale™ microarchitecture	Differences	Comments
TMIA	MIA	Wireless MMX™ technology provides a 64-bit accumulator	Performs multiply-accumulate using signed 32-bit operands from the two source ARM* Core registers
TMIAxy	MIAxy	Wireless MMX™ technology provides a 64-bit accumulator	Performs a 16-bit multiply-accumulate using two signed operands from the ARM* Core registers
TMIAPH	TMIAPH	Wireless MMX™ technology provides a 64-bit accumulator	Performs multiply-accumulate using signed 16-bit operands from the two source ARM* Core registers

4.2 General Guidelines for Porting MMX™ technology and Integer SSE Code to Wireless MMX™ technology

The target pipeline and the instruction set architecture is different for Wireless MMX™ technology, so changes may be required for optimal mapping. Considerations while porting code include:

- Wireless MMX™ technology provides 3 operand instructions – 2 source operand and 1 destination operand so destructive write operations are not necessary.
- Wireless MMX™ technology provides a larger 16x64-bit register file rather than the 8 registers supplied by MMX™ technology. This allows more intermediate values to be stored in the register file.
- Instruction latencies – Instruction latencies are different with Wireless MMX™ technology Always look at the scheduling of instructions, especially the load/store and multi-cycle instructions.
- Instruction Pairing – MMX™ technology interleaves with x86 to reduce stalls. May need to alter the pairing of instructions in some cases on Wireless MMX™ technology and provide appropriate interleaving with Intel® XScale™ microarchitecture instructions
- Operand Alignment – DWORD load/store requires 64-bit alignment. The pointers must be on a 64-bit boundary to avoid an exception, become familiar with Wireless MMX™ technology alignment capability to handle algorithms which naturally operate on 8-bit, 16-bit, or 32-bit boundaries.
- Memory Latency – Software Pipelining is influenced by the specific product.

The following sections address some of the aspects of porting code sequences which are directly related to the differences in the instruction set architecture.

4.3 Wireless MMX™ technology and MMX™ technology/SSE Programming models

The Wireless MMX™ technology implements a register-to-register or load-store architecture in which memory can only be accessed with load and store instructions. In contrast, the x86 implements a register-memory architecture with memory references in the instructions being permitted. This restricts the number of registers used in the encoding for the integer MMX™ technology and SSE technology instructions and often requires a register move instruction to be used since a source operand is destroyed as a result of the operation.

The following sections address some of the aspects of porting code sequences which are directly related to the differences in the instruction set architecture.

4.3.1 Instruction Syntax Differences

The MMX™ technology and SSE instructions use different designators for data types and also have some variation in the suffixes. A typical MMX™ technology instruction has this syntax:

Prefix: **P** for Packed

Instruction operation: for example – ADD, CMP, or XOR

Suffix:

--**US** for Unsigned Saturation
--**S** for Signed saturation
--**B, W, D, Q** for the data type: packed byte, packed word, packed double-word, or quadword.

Instructions that have different input and output data elements have two data-type suffixes. For example, the conversion instruction converts from one data type to another. It has two suffixes: one for the original data type and the second for the converted data type.

A typical Wireless MMX™ instruction has this syntax:

Prefix: **W** for Wireless MMX™ technology data operations

Instruction operation: for example – ADD, CMP, or XOR

Suffix:

--**US** for Unsigned Saturation
--**SS** for Signed saturation
--**B, H, W, D** for the data type: packed byte, packed word, packed double-word, or quadword.

This is an example of an instruction mnemonic syntax for a packed addition with saturation for 16-bit signed elements:

Wireless MMX™ Instructions	**MMX™ technology Instructions**
WADDHUS	**PADDUSW**

W = Packed
ADD = the instruction operation
H = Half-word (16-bit)
US = Unsigned Saturation

P = Packed
ADD = the instruction operation
US = Unsigned Saturation
W = Word (16-bit)

Table 4-4 shows the different data type naming conventions used for Wireless MMX™ technology and MMX™ technology.

Table 4-4. ARM* to IA-32 Data Type Name Mapping

Data Size	Wireless MMX™ technologyName	MMX™ technology/SSE Name
8	Byte	Byte
16	Half-Word	Word
32	Word	Double Word
64	Double Word	Quad Word

4.3.2 Instruction Format Differences

The Wireless MMX™ technology provides encoding for three registers unlike the MMX™ technology instructions which provide for only two registers. The general format for Intel Architecture (IA) instructions is as follows:

```
OPERATION operand1, operand2
```

Operand1 is always the destination operand in addition to being a source operand. For example, in

```
PADDSW mm0, mm1
```

the contents of mm0 are added to mm1 and placed in mm0. The Wireless MMX™ technology format is different in that three operands are available. The general format for Wireless MMX™ instructions is as follows:

```
OPERATION operand3, operand1, operand2
```

Operand3 is always the destination operand and may be different from the source operands. For example, with the equivalent packed add instruction with Wireless MMX™ technology,

```
WADDHSS  wR2,   wR0,  wR1
```

the contents of wR0 are added to wR1 and placed into wR2.

4.3.2.1 Eliminate MOV Instructions

Since the destination registers may be different from the source registers when converting MMX™ technology code to Wireless MMX™ technology, most MOV instructions used to duplicate operands can be eliminated from the instruction sequence. For example, the MMX™ technology code sequence,

```
MOVQ   mm2, mm0
PADDSW mm0, mm1
```

copies the contents of mm0 prior to doing the addition operation. The equivalent Wireless MMX™ instructions allows the contents of the source operand registers to be preserved.

```
WADDHSS  wR2,   wR0,  wR1
```

4.3.2.2 Separate Load & Store Instructions from Operations

One of the most critical items to notice when converting existing code sequences is that the MMX™ technology and SSE instruction set is that Wireless MMX™ technology is register-memory architecture where Wireless MMX™ technology has three registers and separate load and store instructions.

Sample Wireless MMX™ Instructions	**Sample MMX™ technology Instructions**
WADDHSS wRd, wRn, wRm	PADDSSW mm, mm/mm64
WSUBHSS wRd, wRn, wRm	PSUBSSW mm, mm/mm64

Code sequences in MMX™ technology need to have MOV instructions associated with the destructive register behavior removed to improve throughput, and any instruction using memory references in MMX™ technology with require an explicit load in Wireless MMX™ technology.

4.3.3 Shift Instruction Differences

The Wireless MMX™ technology shift instructions do not support a shift by immediate form. The logical and arithmetic shift instructions provide two options for implementation. They may either use a data or auxiliary register to hold the shift operand. The MMX™ technology shift instructions provide options to have the shift count in either a register or an immediate.

Sample Wireless MMX™ Instruction	**Sample MMX™ technology Instruction**
WSRL <H,W,D> wRd, wRn, wRm	PSRA[w/d/q] mm,mm/mm64

Code sequences in MMX™ technology need to have any shift by immediate instructions altered to provide the shift count in either a data or general purpose register.

4.3.4 Multiply-Accumulate Instructions

The Wireless MMX™ technology instruction set includes additional useful processing capabilities beyond the MMX™ technology and SSE integer instructions. The most common instructions to become familiar with are the high precision multiply-accumulate group. The WMAC, TMIA, TMIAxy, and TMIAPH instructions should always be evaluated for replacement of critical inner loops for image, graphics, audio, voice and typical signal processing functions.

The typical MMX™ technology sequence for implementing a multiply-accumulate is the combination of the PMADD and PADD instructions where Wireless MMX™ technology combines these with higher precision into a single instruction.

Sample Wireless MMX™ Instructions	Sample MMX™ technology Instructions	
WMAC wRd, wRn, wRm	PMADDWD	mm, mm/mm64
	PADDD	mm, mm/mm64

Code sequences converted to Wireless MMX™ technology benefits from the increased precision and in some cases allow you to eliminate any pre-scaling sequences otherwise needed to perform work within the limits of 32-bit arithmetic.

4.4 Examples of Porting MMX™ Technology & SSE Code Sequences

The following sections provide several examples of converting exiting code sequences.

4.4.1 Unsigned Unpack Example

The Wireless MMX™ technology provides instructions for unpacking 8 bit, 16 bit, or 32 bit data and either sign-extending or zero extending. The MMX™ technology provides several instructions that pack and unpack data in the MMX™ technology registers. The unpack instructions can be used to zero-extend an unsigned number. Example 4-1 assumes the source is a packed-halfword (16-bit) data type.

The unsigned unpack replaces the MMX™ technology sequence, shown in Example 4-1.

Example 4-1. Unsigned Unpack MMX™ technology to Wireless MMX™ Technology Conversion

Wireless MMX™ Instructions	MMX™ technology Instructions	
Input: wR0 : Source Value	Input: mm0	: Source Value
	mm7	: 0
WUNPCKELUH wR1 , wR0	MOVQ	mm1, mm0
WUNPCKEHUH wR2 , wR0	PUNPCKLWD	mm0, mm7
	PUNPCKHWD	mm1, mm7

4.4.2 Signed Unpack Example

Signed numbers should be sign-extended when unpacking the values. This is done differently than the zero-extend shown in Example 4-1. The following example assumes the source is a packed-halfword (16-bit) data type.

The signed unpack replaces the MMX™ technology sequence, shown in Example 4-2 on page 42.

Example 4-2. Signed Unpack MMX™ technology to Wireless MMX™ Technology Conversion

Wireless MMX™ Instructions	MMX™ technology Instructions
Input: wR0 : Source Value	Input: mm0 : Source Value
WUNPCKELSH wR1 , wR0	PUNPCKHWD mm1, mm0
WUNPCKEHSH wR2 , wR0	PUNPCKLWD mm0, mm0
	PSRAD mm0, 16
	PSRAD mm1, 16

4.4.3 Complex Multiply by a Constant

Complex multiplication is an operation which requires four multiplications and two additions. This is exactly how the PMADDWD instruction operates. In order to use this instruction you need only format the data into four 16-bit values. The real and imaginary components should be 16-bits each. Let the input data be Dr and Di

where Dr = real component of the data
Di = imaginary component of the data

Format the constant complex coefficients in memory as four 16-bit values [Cr -Ci Ci Cr]. Example 4-3 illustrates a complex multiplication by a constant.

Example 4-3. Complex Multiply by Constant MMX™ technology to Wireless MMX™ Technology conversion

Wireless MMX™ Instructions	MMX™ technology Instructions
Input: wR0 : Complex number [Dr,Di]	mm0 : Complex number [Dr,Di]
wR1: Complex coeff. [Cr -Ci Ci Cr]	mm1 : Complex coeff. [Cr -Ci Ci Cr]
WUNPCKILW wR2, wR0, wR0	PUNPCKLDQ MM0,MM0
WMADDS wR3, wR2, wR1	PMADDWD MM0, MM1

Note that the output is a packed word. If needed, a pack instruction can be used to convert the result to 16-bit (thereby matching the format of the input).

4.4.4 Absolute Difference of Unsigned Numbers

Example 4-4 computes the absolute difference of two unsigned numbers. It assumes an unsigned packed-byte data type. Here, the subtract instruction is used with unsigned saturation. This instruction receives UNSIGNED operands and subtracts them with UNSIGNED saturation.

Example 4-4. Absolute Difference of Unsigned Numbers MMX™ technology to Wireless

MMX™ Technology Conversion

Wireless MMX™ Instructions		MMX™ technology Instructions	
Input: wR0 : Source Value 1		Input: mm0 : Source Value 1	
wR1 : Source Value 2		mm1 : Source Value 2	
WSUBBUS	wR3, wR1, wR2	MOVQ	MM2, MM0
WSUBBUS	wR4, wR2, wR1	PSUBUSB	MM0, MM1
WOR	wR5, wR3, wR4	PSUBUSB	MM1, MM2
		POR	MM0, MM1

Example 4-4 will not work if the operands are signed. See the Example 4-5 for signed absolute differences.

4.4.5 Absolute Difference of Signed Numbers

Example 4-5 computes the absolute difference of two signed numbers. There is no Wireless MMX™ instruction subtract which receives SIGNED operands and subtracts them with UNSIGNED saturation. The technique used here is to first sort the corresponding elements of the input operands into packed-words of the maxima values, and packed-words of the minima values. Then the minima values are subtracted from the maxima values to generate the required absolute difference. The key is a fast sorting technique which uses the fact that B = XOR(A, XOR(A,B)) and A = XOR(A,0). Thus in a packed data type, having some elements being XOR(A,B) and some being 0, you could XOR such an operand with A and receive in some places values of A and in some values of B. Example 4-5 assume a packed-word data type, each element being a signed value.

Example 4-5. Absolute Difference of Unsigned Numbers MMX™ technology to Wireless MMX™ Technology Conversion

Wireless MMX™ Instructions		MMX™ technology Instructions	
Input: wR0 : Source Value 1		Input: mm0 : Source Value 1	
wR1 : Source Value 2		mm1 : Source Value 2	
WCMPGTSH	wR2, wR1, wR0	MOVQ	MM2, MM0
WXOR	wR3, wR0, wR1	PCMPGTW	MM0, MM1
WAND	wR4, wR3, wR0	MOVQ	MM4, MM2
WXOR	wR5, wR0, wR4	PXOR	MM2, MM1
WXOR	wR6, wR1, wR4	PAND	MM2, MM0
WSUBH	wR2, wR5, wR6	MOVQ	MM3, MM2
		PXOR	MM4, MM2
		PXOR	MM1, MM3
		PSUBW	MM1, MM4

Instruction Set A

This chapter presents the Wireless MMX™ instructions in alphabetical order, with a full description of each instruction. Appendix B, “Instruction Encoding Summary”provides instruction formats and encodings, and Appendix C, “Intrinsic Support” provides an alphabetical list of the related C intrinsic functions for each instruction.

A.1 Instruction Syntax

This section describes the syntax that is used in the Wireless MMX™ instruction set.

All Wireless MMX™ instructions are prefixed with either a “**T**” or “**W**”:

- A Prefix of **T** signifies that this is a transfer instruction. Data is being transferred between the Intel® XScale™ microarchitecture and the Wireless MMX™ technology. Some instructions perform a data operation on the data as it is transferred. e.g. TBCSTB
- A prefix of **W** signifies that this is a Wireless MMX™ technology data operation that operates on packed data from the Wireless MMX™ technology register file. e.g. WADDB

Where appropriate the instruction format also contains a suffix to indicate the data type that is being processed. The suffixes in Table A-1 indicate the size of the data.

Table A-1. ARM* to IA-32 Data-Type Name Mapping

Data Size	Qualifier	Description
8	**B**	Byte
16	**H**	Half-Word
32	**W**	Word
64	**D**	Double Word

The instruction set also contains suffixes for certain qualifiers, such as US for unsigned saturation. Some of these are mandatory and some are optional:

- Optional qualifiers to instructions contained in “{“ “}” need not be specified, if not required. For example, WADD <B,H,W>{US,SS}{Cond} wRd, wRn, wRm. In this case US, SS are optional qualifiers; B, H, and W are compulsory.
- Required qualifiers are contained in “<“ “>”. One of the qualifiers must be chosen. For example, for WACC<B,H,W> Rd, wRn the B, H, or W flag must be specified; that is, valid mnemonics are WACCB, WACCH, or WACCW. WACC alone is not valid.

When specifying operands to Wireless MMX™ instructions this syntax is used for register names:

- Rn – ARM* Primary Source register
- Rm – ARM* Secondary Source register
- Rd – ARM* Destination register

Wireless MMX™ technology Data registers are prefixed by “wR”:

- wRn – Wireless MMX™ technology Primary Source register
- wRm – Wireless MMX™ technology Secondary Source register
- wRd – Wireless MMX™ technology Destination register

Wireless MMX™ technology Control registers are prefixed by “wC”:

- wCx – Refers to any Control register
- wCGRn – Refers to any of the General-Purpose Control registers

Immediate operands are prefixed by a “#”, see the appropriate instruction definition to determine the valid range of values for Immediate operands.

In Wireless MMX™ instructions the syntax of Conditional Execution Specifiers is the same as in the ARM* architecture.

A.1.1 Reserved Instruction Fields

An invalid instruction is defined to be an instruction that contains any field encoding that is indicated as reserved in this book. Invalid instructions either take an undefined instruction exception or execute without taking the exception and the result is defined to be unpredictable. Table A-2 describes the exact definition of each type.

Table A-2. Effect of Reserved Fields on Execution

Type	Condition
Unpredictable Result	The result of the operation is defined to be unpredictable if the instruction decodes to one of instruction types defined in this book, but one or more of the qualifier fields decode to a reserved value. The instruction still executes, but the results are not defined. No exception is caused.
Undefined Exception	An undefined exception results if the instruction does not decode to one of the instruction types defined in this document or the coprocessor number does not match those of the Wireless MMX™ technology coprocessor

A.1.2 Number Representation

All numbers in this document are base 10, unless designated otherwise. In text and pseudo-code descriptions, hexadecimal numbers have a prefix of 0x and binary numbers have a prefix of 0b. For example, 107 would be represented as 0x6B in hexadecimal and 0b1101011 in binary.

A.1.3 Terminology and Acronyms

SIMD	Single Instruction Multiple Data
PSR	Processor Status Register.
Reserved	A **reserved** field may be used by an implementation. If software provides the initial value of a reserved field, this value must be zero. Software should not modify reserved fields or depend on any values in reserved fields.

A.1.4 Symbols Glossary

This section describes the functions and symbols (shown in Table A-3) used to describe the operation of each instruction in Section A.2.2, “Instruction set List” on page 49. This document does not describe basic symbols.

Table A-3. Symbol and Function Glossary

Symbol	Description
>>	Shift Right
<<	Shift Left
Nibble x	Refers to Nibble x, counting from 0 at the LSB
Byte x	Refers to Byte x, counting from 0 at the LSB
Half x	Refers to Half Word x, counting from 0 at the LSB
Word X	Refers to Word x, counting from 0 at the LSB
signReplicate(X,Y)	Replicate Sign bit of X across the width (Y) of the specifed field
signExtend(X,Y)	Sign Extend X to the width of the result field
saturate(X,Y,Z)	Saturate x to the maximum or minimum value, with the output result width specified by Z and using Y to determine whether to use the signed or unsigned maximum and minimum values
Low_DB_word	Use the lower double word of the result
(Condition)? X: Y	If condition is true then X else Y
abs(X)	The result is the absolute value of X

A.2 How To Read The Instruction Set Pages

Section A.2.1 is an example of the format used for each Wireless MMX™ instruction description in this appendix.

A.2.1 Example Instruction Format: WSLL

Overview Performs vector logical shift-left of wRn by wRm for vectors of 16-, 32-, or 64-bit data and places the result in wRd.

*The **Overview** section briefly describes the behavior of the instruction.*

Usage
WSLL<H,W,D>{Cond} wRd, wRn, wRm
WSLL<H,W,D>G {Cond} wRd, wRn, wCGRn

*The **Usage** section describes the legal syntax of the instruction.*

Qualifiers
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD
D – 64-bit (Double)

G – wRm field specifies general purpose register to use for shift value

*The **Qualifiers** section describes all available qualifiers for this instruction*

Operation

```
if (H Specified)then
  wRd[half 3] = wRn[half 3] << ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[half 2] = wRn[half 2] << ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[half 1] = wRn[half 1] << ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[half 0] = wRn[half 0] << ((G Specified) ? wCGRm[7:0]:wRm[7:0])
else if (W Specified) then
  wRd[word 1] = wRn[word 1] << ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[word 0] = wRn[word 0] << ((G Specified) ? wCGRm[7:0]:wRm[7:0])
else if (D Specified) then
  wRd = wRn << ((G Specified) ? wCGRm[7:0]:wRm[7:0])
```

*The **Operation** section describes the behavior of the instruction in psuedo code*

Exceptions None

*The **Exceptions** section describes any exceptions that the instruction can cause*

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ww01				wRn				wRd				000g				010			0	wRm/wCGRn			

*The **Encoding** section describes the instruction and encoding used and where the qualifiers are encoded. The detailed encoding of the qualifers is described below in the sub-field encoding section*

Sub-Field Encoding

Qualifier	Field	Value
H	ww	01
W	ww	10
D	ww	11
Reserved	ww	00
G	g	1
No G	g	0

*The **Sub-Field Encoding** section describes the binary encoding of the qualifiers*

SIMD Flags The V, C flags are cleared; the Z and N flag are set as a result of the operation,; does not effect the Saturation flags (wCSSF).

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

*The **SIMD Flags** section describes which of the SIM arithmetic and saturation flags are effected by the instructions*

Note Zero is shifted in from the right; if shift value > 63 (for D qualifier), 31 (for W qualifier) or 15 (for H qualifier), the Destination register contains all zeros. Specifying an illegal value for wCGRm when the G qualifier is used causes an undefined instruction exception.

*The **Note** section adds clarifying notes to the description of the instruction*

A.2.2 Instruction set List

The rest of the appendix describes the instruction set in alphabetical order:

A.3 TANDC

Overview Performs “AND” across the fields of the SIMD PSR register (wCASF) and sends the result to the ARM* CPSR; can be performed after a Byte, Half-word or Word operation that sets the flags.

Usage TANDC<B,H,W>{Cond} R15

Qualifiers
B – Transfer Flags after a Byte Operation
H – Transfer Flags after a Half-word Operation
W – Transfer Flags after a Word Operation

Operation

```
if (B Specified) then
  CPSR[31:28] = wCASF[31:28] & wCASF[27:24] & wCASF[23:20] & wCASF[19:16]
   & wCASF[15:12] & wCASF[11:8] & wCASF[7:4] & wCASF[3:0]

if(H Specified) then
  CPSR[31:28] = wCASF[31:28] & wCASF[23:20] & wCASF[15:12] & wCASF[7:4]

else if (W Specified)
  CPSR[31:28] = wCASF[31:28] & wCASF[15:12]
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ww0			1	0011				1111				0001				001			1	0000			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note Specifying any destination register other than R15 causes a undefined instruction exception.

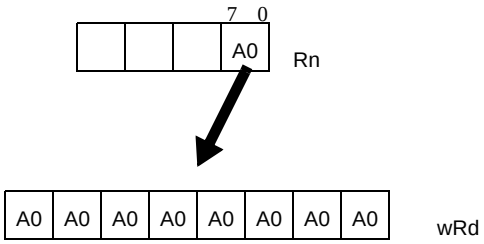
A.4 TBCST

Overview Broadcasts a value from the ARM* Source register, Rn, or to every SIMD position in the Wireless MMX™ technology Destination register, wRd; can operate on 8-, 16-, and 32-bit data values.

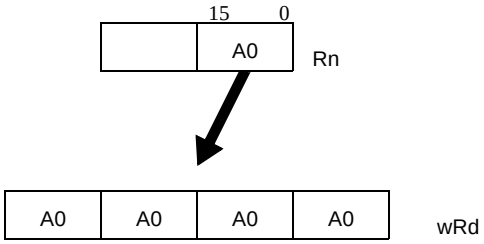
Usage TBCST<B,H,W>{Cond} wRd, Rn

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD

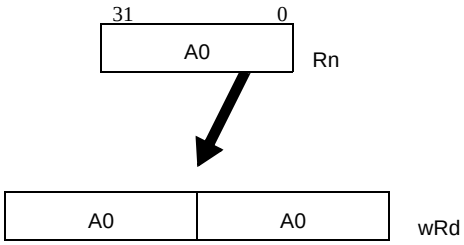
Operation `TBCSTB wRd,Rn`



`TBCSTH wRd,Rn`



`TBCSTW wRd,Rn`



Exceptions None

Encoding

TBCST

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				010			0	wRd				Rn				0000				ww0			1	0000			

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note

A.5 TEXTRC

Overview Extracts 4-/8-/16-bit field specified by the 3-bit Immediate field data from the SIMD PSR register (wCASF), and transfers to the ARM* CPSR.

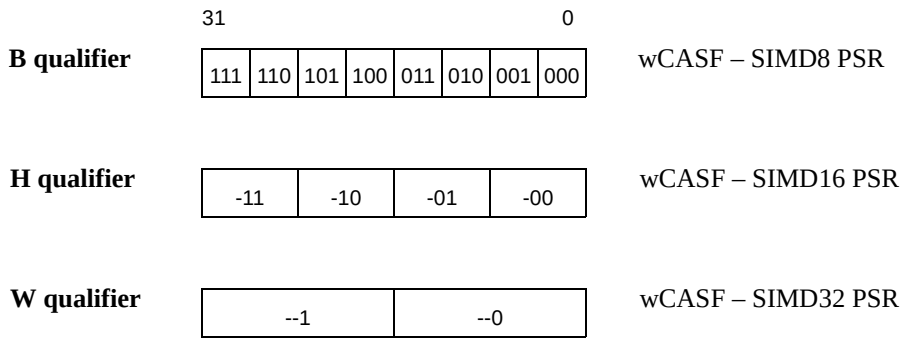
Usage TEXTRC<B,H,W>{Cond} R15, #Imm3

Qualifiers
B – Transfer Flags after a Byte Operation
H – Transfer Flags after a Half-word Operation
W – Transfer Flags after a Word Operation
Imm – Specifies the Field to transfer flags from

Operation

```
if (B Specified) then
   CPSR[31:28] = wCASF[Nibble[#Imm[2:0]]]
else if (H Specified) then
   CPSR[31:28] = wCASF[Nibble[#Imm[1:0],1]]
else if (W Specified)
   CPSR[31:28] = wCASF[Nibble[#Imm[0],1,1]]
```

IMM3 Values to get the corresponding flag fields



Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ww0			1	0011				1111				0001				011			1	0bbb			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11
Imm3	bbb	3-Bit Offset

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note Imm value specifies which nibble/byte/half to extract; Specifying any destination register other than R15 causes a undefined instruction exception.

A.6 TEXTRM

Overview Extracts 8-/16-/32-bit field specified by the 3-bit Immediate field data from Wireless MMX™ technology Source register, wRn, and transfers to the specified ARM* Core register, Rd.

Usage TEXTRM<U,S><B,H,W>{Cond} Rd, wRn, #Imm3

Qualifiers
B – Extract 8 -bits (Byte)
H – Extract 16-bits (Half Word)
W – Extract 32-bits (Word)
S – Sign Extend
U – Unsigned
Imm – Specifies which Byte/Half/Word to extract

Operation

```
if (B Specified) then
   Rd[7:0] = wRn[Byte #Imm[2:0]]
   Rd[31:8]= (S Specified) ? SignReplicate(wRn[Byte #Imm3[2:0]],24): 0
else if (H Specified)
  Rd[15:0] = wRn[Half #Imm[1:0]]
   Rd[31:16]=(S Specified) ? SignReplicate(wRn[Half #Imm3[1:0]],16): 0
else if (W Specified) then
  Rd[31:0] = wRn[Word #Imm3[0]]
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ww0			1	wRn				Rd				0000				011			1	sbbb			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11
U	s	0
S	s	1
Imm3	bbb	3-Bit Offset

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note Imm value specifies which byte/half/word to extract; specifying R15 as the Destination register produces an unpredictable result. For word extraction (W-Qualifier) the U or S qualifier is still required. As the extracted value does not get modified by these qualifiers, either U or S may be used for word extraction (the result is the same).

A.7 TINSR

Overview Transfers and inserts 8-/16-/32-bit data from ARM* Source register, Rn, to the position in Wireless MMX™ technology Destination register, wRd, specified by the 3-bit Immediate.

Usage TINSR<B,H,W>{Cond} wRd, Rn, #Imm3

Qualifiers
B – Insert 8 -bits (Byte)
H – Insert 16-bits (Half Word)
W – Insert 32-bits (Word)
Imm – Specifies which Byte/Half/Word to insert

Operation

```
if (B Specified) then
  wRd[Byte #Imm3[2:0]] = Rn[7:0]
else if (H Specified)
  wRd[Half #Imm3[1:0]] = Rn[15:0]
else if (W Specified) then
  wRd[Word #Imm3[0]] = Rn[31:0]
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				011			0	wRd				Rn				0000				ww0			1	0bbb			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11
Imm3	bbb	3-Bit Offset

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note Imm value specifies which byte/half/word to insert; remaining fields of the Destination register are unchanged.

A.8 TMCR

Overview TMCR is a pseudo-instruction that maps onto ARM* MCR instruction, and is provided for convenience; TMCR transfers the contents of the Rn ARM* Core registers to a 32-bit wCx Wireless MMX™ technology Control register.

Usage TMCR{Cond} wCx, Rn

Qualifiers None

Operation `wCx[31:0]=Rn`

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				000			0	wCx				Rn				0001				000			1	0000			

Sub-Field Encoding
None

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF), however wCASF or wCSSF is updated if it is specified as the Destination register (wCx).

Note Used with Wireless MMX™ technology Control register; for data registers, use TEXTRM, TINSR, TMRRC and TMCRR instead.

A.9 TMCRR

Overview TMCRR is a pseudo-instruction that maps onto an ARM* MCRR instruction and is provided for convenience; TMCRR transfers the contents of the two ARM* registers (RdHi and RdLo) to wRd (the 64-bit Wireless MMX™ technology Destination register.)

Usage TMCRR{Cond} wRd, RdLo, RdHi

Qualifiers None

Operation

```
wRd[63:32]=RdHi
wRd[31:0] =RdLo
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				11000100								RdHi				RdLo				0000				0000				wRd			

Sub-Field Encoding
None

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note Can only be used with Wireless MMX™ technology Data registers; similar functionality as provided by the Intel® XScale™ microarchitecture MAR instruction (note that the core MAR transfers a 40-bit accumulator whereas Wireless MMX™ technology provides a full 64-bit register transfer); specifying R15 as the Destination register produces an unpredictable result.

A.10 TMIA

Overview Provides the same functionality as the Intel® XScale™ microarchitecture MIA instruction; performs multiply-accumulate using signed 32-bit operands from the two source ARM* Core registers (Rm,Rs), and accumulates the result with the Wireless MMX™ technology Destination register, wRd.

Usage TMIA{Cond} wRd, Rm, Rs

Qualifiers None

Operation

```
wRd = (Rm[31:0] * Rs[31:0]) + wRd[63:0]
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				001			0	0000				Rs				000			wRd				1	Rm			

Sub-Field Encoding None

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note Can also use mnemonic without leading T (i.e., MIA); specifying R15 as Rs or Rm produces an unpredictable result; provides same functionality as provided by the Intel® XScale™ microarchitecture MIA instruction. (note that the core MIA calculates a 40-bit result whereas Wireless MMX™ technology provides a full 64-bit register result; core and Wireless MMX™ technology therefore produces different results if the result overflows 40-bits—the core MIA overflows whereas Wireless MMX™ technology TMIA produces the correct results until a 64-bit overflow.)

A.11 TMIAPH

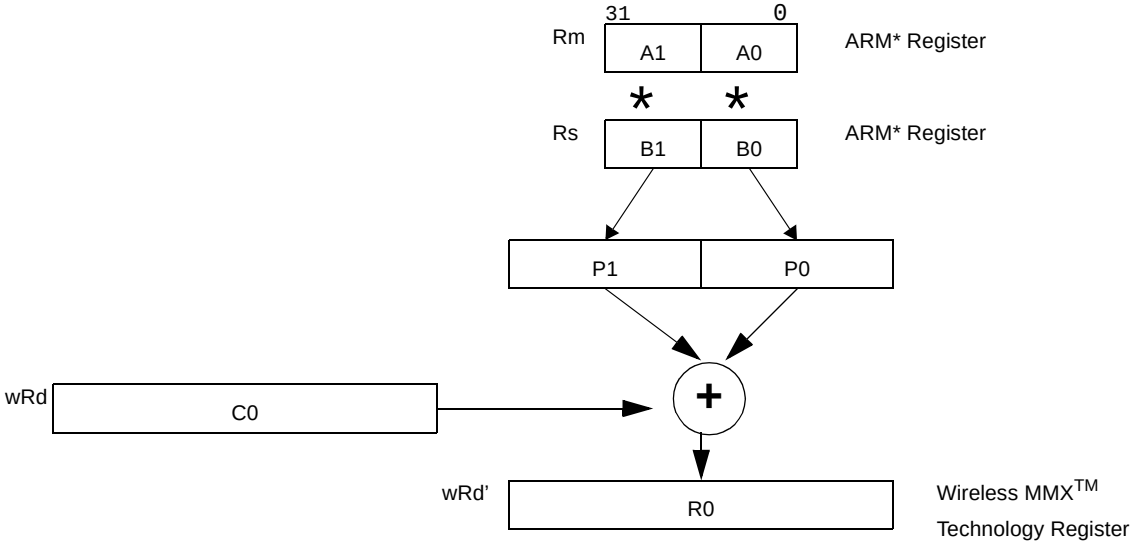
Overview Provides the same functionality as the Intel® XScale™ microarchitecture MIAPH instruction; performs multiply-accumulate using signed 16-bit operands from the two source ARM* Core registers (Rm.Rs), and accumulates the result with the Wireless MMX™ technology Destination register, wRd.

Usage TMIAPH{Cond} wRd, Rm, Rs

Qualifiers None

Operation

```
wRd[63:0] = sign_extend((Rm[31:16]*Rs[31:16])+(Rm[15:0]*Rs[15:0])) + wRd
```



Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				001			0	1000				Rs				000			wRd				1	Rm			

Sub-Field Encoding

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note Can also use mnemonic without leading T (i.e., MIAPH); specifying R15 as Rs or Rm produces an unpredictable result; provides same functionality as that from the Intel® XScale™ microarchitecture MIAPH instruction (note that the core MIAPH calculates a 40-bit result whereas Wireless MMX™ technology provides a full 64-bit register result; core and Wireless MMX™ technology therefore produces different results if the result overflows 40-bits—core MIAPH overflows whereas Wireless MMX™ technology TMIAPH produces the correct results until a 64-bit overflow.)

A.12 TMIAxy

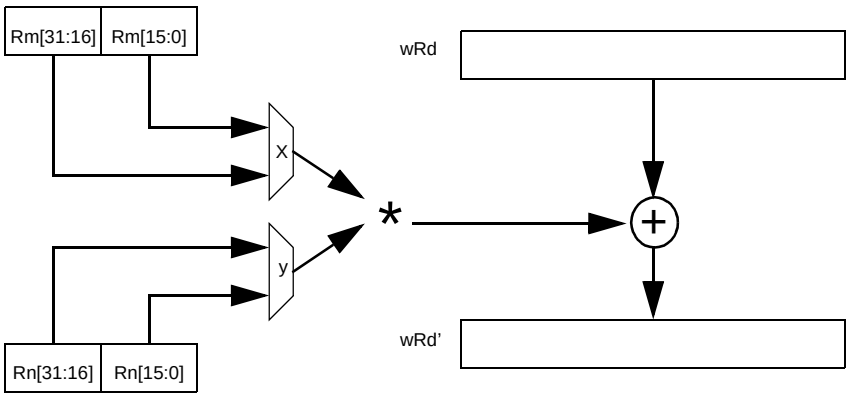
Overview Provides same functionality as Intel® XScale™ microarchitecture MIAxy instruction; performs a 16-bit multiply-accumulate using two signed operands from the ARM* Core registers (Rm,Rs) and accumulates the result with the Wireless MMX™ technology Destination register; upper or lower 16-bits of each Source register is selected by specifying either a B (bottom) or T (qualifier) in each of the xy positions of the mnemonic.

Usage TMIA<T,B><T,B>{Cond} wRd, Rm, Rs

Qualifiers B – Use bottom field
T – Use top field

Operation

```
<operand1> = (T Specified in x position)>? Rm[31:16] : Rm[15:0]
<operand2> = (T Specified in y position)? Rs[31:16] : Rs[15:0]
wRd[63:0] = sign_extend((<operand1>*<operand2>),64) + wRd[63:0]
```



Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				001			0	11xy				Rs				000			wRd				1	Rm			

Sub-Field Encoding

Qualifier	Field	Value
T	x	1
B	x	0
T	y	1
B	y	0

Sub-Field Encoding

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note Can also use mnemonic without leading T (i.e., MIAxy); specifying R15 as Rs or Rm produces an unpredictable result; provides same functionality as provided by the Intel® XScale™ microarchitecture MIAxy instruction (note that the core MIAxy calculates a 40-bit result whereas Wireless MMX™ technology provides a full 64-bit register result; core and Wireless MMX™ technology therefore produces different results if the result overflows 40-bits—core MIAxy overflows whereas Wireless MMX™ technology TMIAxy produces the correct results until a 64-bit overflow.)

A.13 TMOVMSK

Overview Transfers the most significant bit of each SIMD field of the Wireless MMX™ technology Source register (wRn) to the least significant 8, 4, or 2 bits of the specified ARM* Destination register (Rd); this instruction operates on 8-,16-, and 32-bit data values.

Usage TMOVMSK<B,H,W>{Cond} Rd, wRn

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD

Operation

```
if (B Specified) then
  Rd[7]=wRn[63]; Rd[6]=wRn[55]; Rd[5]=wRn[47]; Rd[4]=wRn[39]
  Rd[3]=wRn[31]; Rd[2]=wRn[23]; Rd[1]=wRn[15]; Rd[0]=wRn[7]
  Rd[31:8] =0;
else if (H Specified) then
  Rd[3]=wRn[63]; Rd[2]=wRn[47]; Rd[1]=wRn[31]; Rd[0]=wRn[15];
  Rd[31:4] =0;
else if (W Specified) then
  Rd[1]=wRn[63]; Rd[0]=wRn[31]; Rd[31:2]=0;
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ww0			1	wRn				Rd				0000				001			1	0000			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note Specifying R15 as the Destination register produces an unpredictable result.

A.14 TMRC

Overview TMRC is a pseudo-instruction that maps onto an ARM* core MRC instruction and is provided for convenience; TMRC transfers the contents of the 32-bit Wireless MMX™ technology Control register (wCx) to the ARM* core Destination register (Rd).

Usage TMRC{Cond}Rd, wCx

Qualifiers None

Operation

```
Rd = wCx[31:0]
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				000			1	wCx				Rd				0001				000			1	0000			

Sub-Field Encoding
None

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note Can only be used with Wireless MMX™ technology Control registers; for Data registers, use TEXTRM, TINSR, TMRRC and TMCRR instead; specifying R15 as the Destination register produces an unpredictable result.

A.15 TMRRC

Overview TMRRC is a pseudo-instruction that maps onto a standard ARM* MRRC instruction and is provided for convenience; TMRRC transfers the contents of the wRn 64-bit Wireless MMX™ technology Data register to two ARM* core Destination registers (RdHi, RdLo).

Usage TMRRC{Cond} RdLo, RdHi, wRn

Qualifiers None

Operation

```
RdHi=wRn[63:32]
RdLo=wRn[31:0]
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				11000101								RdHi				RdLo				0000				0000				wRn			

Sub-Field Encoding
None

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note Can only be used with Wireless MMX™ technology Data registers; results of this instruction are unpredictable if RdHi=RdLo. Specifying R15 as one of the destination registers (RdHi,RdLo) also produces an unpredictable result. Similar functionality as provided by the Intel® XScale™ microarchitecture MRA instruction (note that the core MRA transfers a 40-bit accumulator whereas Wireless MMX™ technology provides a full 64-bit register transfer.)

A.16 TORC

Overview Performs “OR” across the fields of the SIMD PSR (wCASF) and sends the result to the ARM* CPSR; operation can be performed after a Byte, Half-word or Word operation that sets the flags.

Usage TORC<B,H,W>{Cond} R15

Qualifiers
B – Transfer Flags after a Byte Operation
H – Transfer Flags after a Half-word Operation
W – Transfer Flags after a Word Operation

Operation

```
if (B Specified) then
  CPSR[31:28] = wCASF[31:28] | wCASF[27:24] | wCASF[23:20] | wCASF[19:16]
   | wCASF[15:12] | wCASF[11:8] | wCASF[7:4] | wCASF[3:0]

if(H Specified) then
  CPSR[31:28] = wCASF[31:28] | wCASF[23:20] | wCASF[15:12] | wCASF[7:3]

else if (W Specified)
  CPSR[31:28] = wCASF[31:28] | wCASF[15:12]
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ww0			1	0011				1111				0001				010			1	0000			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note Specifying any destination register other than R15 causes a undefined instruction exception.

A.17 WACC

Overview Performs an unsigned accumulate across the Source register (wRn) fields, and writes result to Destination register, wRd; operation can be performed on fields of byte, half-word, and word size.

Usage WACC<B,H,W>{Cond} wRd, wRn

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD

Operation

```
if (B Specified) then
  wRd = wRn[63:56] + wRn[55:48] + wRn[47:40]wRn[39:32]
   + wRn[31:24] + wRn[23:16] + wRn[15:8] +wRn[7:0]
else if (H Specified)
  wRd = wRn[63:48] + wRn[47:32] + wRn[31:16] + wRn[15:0]
else if (W Specified) then
  wRd = wRn[63:32] + wRn[31:0]
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ww00				wRn				wRd				0001				110			0	0000			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note

A.18 WADD

Overview Performs vector addition of wRn and wRm for vectors of 8-, 16-, or 32-bit signed or unsigned data, and places the result in wRd. Saturation can be specified as signed, unsigned, or no saturation.

Usage WADD<B,H,W>{US,SS}{Cond} wRd, wRn, wRm

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD
SS – Signed Saturation
US – Unsigned Saturation

Operation

```
if (B Specified) then
  wRd[byte 7] = saturate(wRn[byte 7] + wRm[byte 7], {US,SS}, 8)
  wRd[byte 6] = saturate(wRn[byte 6] + wRm[byte 6], {US,SS}, 8)
  wRd[byte 5] = saturate(wRn[byte 5] + wRm[byte 5], {US,SS}, 8)
  wRd[byte 4] = saturate(wRn[byte 4] + wRm[byte 4], {US,SS}, 8)
  wRd[byte 3] = saturate(wRn[byte 3] + wRm[byte 3], {US,SS}, 8)
  wRd[byte 2] = saturate(wRn[byte 2] + wRm[byte 2], {US,SS}, 8)
  wRd[byte 1] = saturate(wRn[byte 1] + wRm[byte 1], {US,SS}, 8)
  wRd[byte 0] = saturate(wRn[byte 0] + wRm[byte 0], {US,SS}, 8)
else if (H Specified)
then
  wRd[half 3] = saturate(wRn[half 3] + wRm[half 3], {US,SS}, 16)
  wRd[half 2] = saturate(wRn[half 2] + wRm[half 2], {US,SS}, 16)
  wRd[half 1] = saturate(wRn[half 1] + wRm[half 1], {US,SS}, 16)
  wRd[half 0] = saturate(wRn[half 0] + wRm[half 0], {US,SS}, 16)
else if (W Specified) then
  wRd[word 1] = saturate(wRn[word 1] + wRm[word 1], {US,SS}, 32)
  wRd[word 0] = saturate(wRn[word 0] + wRm[word 0], {US,SS}, 32)
```

Exceptions

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				wwss				wRn				wRd				0001				100			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11
US	ss	01

Qualifier	Field	Value
SS	ss	11
No Saturation	ss	00
Reserved	ss	10

SIMD Flags Sets SIMD (wCASF) flags on the result of the addition wRn+wRm. If US/SS is specified and saturation occurs then the SIMD (wCASF) flags are set on the post-saturation (final) value. The saturation flags (wCSSF) are set if the US/SS qualifier is specified and the corresponding fields saturates.

Flag	No Saturation Specified	US/SS specified and no Result Saturation	US/SS specified and Result Saturates
N	Set on Final Result	Set on Final Result	Set on Final Result
Z	Set on Final Result	Set on Final Result	Set on Final Result
C	Set on Final Result	Set on Final Result	Cleared
V	Set on Final Result	Set on Final Result	Cleared

Note

A.19 WALIGNI

Overview Extracts an 64-bit value from the two 64-bit Source registers (wRn,wRm), and places the result in the Destination register, wRd; instruction uses a 3-bit immediate value to specify the byte offset of the value to extract.

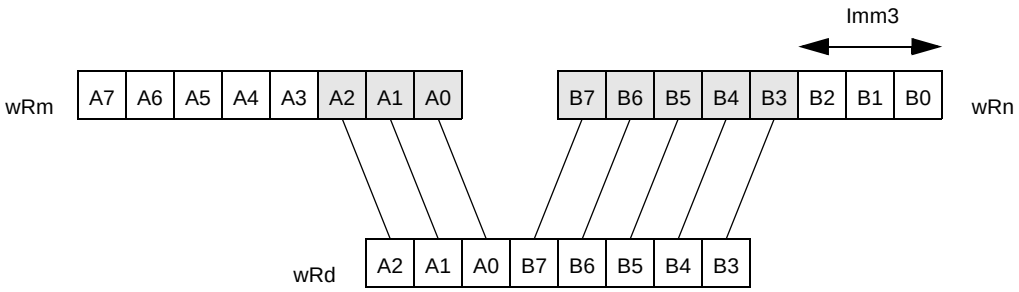
Usage WALIGNI{Cond} wRd, wRn, wRm, #Imm3

Qualifiers None

Operation

```
wRd = Low_DB_word((wRm,wRn) >>(Imm3 * 8))
```

Example Operation, with Imm3=3



Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				0vvv				wRn				wRd				0000				001			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
Imm3	vvv	3-bit Value

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note

A.20 WALIGNR

Overview Extracts a 64-bit value from the two 64-bit Source registers (wRn,wRm), and places the result in the Destination register, wRd; instruction uses a 3-bit value stored in the specified General-Purpose register to specify the byte offset of the value to extract.

Usage WALIGNR<0,1,2,3>{Cond} wRd, wRn, wRm

Qualifiers 0 – Use general purpose register 0 for alignment
1 – Use general purpose register 1 for alignment
2 – Use general purpose register 2 for alignment
3 – Use general purpose register 3 for alignment

Operation

```
if (0 Specified) then
  wRd = Low_DB_word((wRm,wRn) >> (wCGR0[2:0] * 8))
else if (1 Specified) then
  wRd = Low_DB_word((wRm,wRn) >> (wCGR1[2:0] * 8))
else if (2 Specified) then
  wRd = Low_DB_word((wRm,wRn) >> (wCGR2[2:0] * 8))
else if (3 Specified) then
  wRd = Low_DB_word((wRm,wRn) >> (wCGR3[2:0] * 8))
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				10vv				wRn				wRd				0000				001			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
0	vv	00
1	vv	01
2	vv	10
3	vv	11

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note Offset is in bytes.

A.21 WAND

Overview Performs a bit-wise logical “AND” between wRn and wRm, and places the result in the Destination register, wRd.

Usage WAND{Cond} wRd, wRn, wRm

Qualifiers

Operation

```
wRd = wRn & wRm
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				0010				wRn				wRd				0000				000			0	wRm			

SIMD Flags Sets SIMD64 flags only; C and V flags are cleared and the N and Z flags are set according to the result; does not effect the Saturation flags (wCSSF).

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

Note

A.22 WANDN

Overview Performs a bit-wise logical “AND” between wRn and not wRm, and places the result in the Destination register, wRd.

Usage WANDN{Cond} wRd, wRn, wRm

Qualifiers

Operation

```
wRd = wRn & ~wRm
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				0011				wRn				wRd				0000				000			0	wRm			

SIMD Flags Sets SIMD64 flags only; C and V flags are cleared and the N and Z flags are set according to the result; does not effect the Saturation flags (wCSSF).

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

Note

A.23 WAVG2

Overview Performs a 2-pixel average of wRn and wRm on unsigned vectors of 8- or 16-bit data with optional rounding of +1 and places the result in the Destination register, wRd.

Usage WAVG2<B,H>{R}{Cond} wRd, wRn, wRm

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
R – Round using + 1, else no rounding

Operation

```
Round = (R Specified) ? : 1 :0
if (B Specified) then
  wRd[byte 7] = (wRn[byte 7] + wRm[byte 7] + Round)/2
  wRd[byte 6] = (wRn[byte 6] + wRm[byte 6] + Round)/2
  wRd[byte 5] = (wRn[byte 5] + wRm[byte 5] + Round)/2
  wRd[byte 4] = (wRn[byte 4] + wRm[byte 4] + Round)/2
  wRd[byte 3] = (wRn[byte 3] + wRm[byte 3] + Round)/2
  wRd[byte 2] = (wRn[byte 2] + wRm[byte 2] + Round)/2
  wRd[byte 1] = (wRn[byte 1] + wRm[byte 1] + Round)/2
  wRd[byte 0] = (wRn[byte 0] + wRm[byte 0] + Round)/2
else if (H Specified) then
  wRd[half 3] = (wRn[half 3] + wRm[half 3] + Round)/2
  wRd[half 2] = (wRn[half 2] + wRm[half 2] + Round)/2
  wRd[half 1] = (wRn[half 1] + wRm[half 1] + Round)/2
  wRd[half 0] = (wRn[half 0] + wRm[half 0] + Round)/2
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				1h0r				wRn				wRd				0000				000			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
B	h	0
H	h	1
R	r	1
No R	r	0

SIMD Flags Clears N,C,V, Sets Z if zero; does not affect the Saturation flags (wCSSF).

Flag	Action
N	Cleared
Z	Set on Final Result
C	Cleared
V	Cleared

Note The rounding specifies whether a 1 is added to the intermediate result before the divide-by-2.

A.24 WCMPEQ

Overview Performs vector-equality comparison of wRn and wRm for vectors of 8-, 16-, or 32-bit data, setting the corresponding data elements of wRd to all ones when source operands are equal; else all zeros.

Usage WCMPEQ<B,H,W>{Cond} wRd, wRn, wRm

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD

Operation

```
if (B Specified)then
  wRd[byte 7] = (wRn[byte 7] == wRm[byte 7]) ? 0xFF : 0x00
  wRd[byte 6] = (wRn[byte 6] == wRm[byte 6]) ? 0xFF : 0x00
  wRd[byte 5] = (wRn[byte 5] == wRm[byte 5]) ? 0xFF : 0x00
  wRd[byte 4] = (wRn[byte 4] == wRm[byte 4]) ? 0xFF : 0x00
  wRd[byte 3] = (wRn[byte 3] == wRm[byte 3]) ? 0xFF : 0x00
  wRd[byte 2] = (wRn[byte 2] == wRm[byte 2]) ? 0xFF : 0x00
  wRd[byte 1] = (wRn[byte 1] == wRm[byte 1]) ? 0xFF : 0x00
  wRd[byte 0] = (wRn[byte 0] == wRm[byte 0]) ? 0xFF : 0x00
else if (H Specified) then
  wRd[half 3] = (wRn[half 3] == wRm[half 3]) ? 0xFFFF : 0x0000
  wRd[half 2] = (wRn[half 2] == wRm[half 2]) ? 0xFFFF : 0x0000
  wRd[half 1] = (wRn[half 1] == wRm[half 1]) ? 0xFFFF : 0x0000
  wRd[half 0] = (wRn[half 0] == wRm[half 0]) ? 0xFFFF : 0x0000
else if (W Specified) then
  wRd[word 1] = (wRn[word 1] == wRm[word 1]) ? 0xFFFFFFFF : 0x00000000
  wRd[word 0] = (wRn[word 0] == wRm[word 0]) ? 0xFFFFFFFF : 0x00000000
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ww00				wRn				wRd				0000				011			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11

SIMD Flags Set SIMD flags as if operation on the final has occurred; does not effect the Saturation flags (wCSSF). Z flag is cleared if operands are equal (final result = 0xFF) and is set if there is a mismatch (final result = 0x00).

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

Note

A.25 WCMPGT

Overview Performs vector-magnitude comparison of wRn and wRm for vectors of 8-, 16-, or 32-bit data, setting the corresponding data elements of wRd to all ones when corresponding fields of wRn and greater than wRm; else all zeros; operation can be performed on either signed or unsigned data.

Usage WCMPGT<U,S><B,H,W>{Cond} wRd, wRn, wRm

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD
S – Signed Comparison
U – Unsigned Comparison

Operation

```
if(S Specified) then
  wRn and wRm contain signed values
else
  wRn and wRm contain unsigned values.

if (B Specified) then
  wRd[byte 7] = (wRn[byte 7] > wRm[byte 7]) ? 0xFF : 0x00
  wRd[byte 6] = (wRn[byte 6] > wRm[byte 6]) ? 0xFF : 0x00
  wRd[byte 5] = (wRn[byte 5] > wRm[byte 5]) ? 0xFF : 0x00
  wRd[byte 4] = (wRn[byte 4] > wRm[byte 4]) ? 0xFF : 0x00
  wRd[byte 3] = (wRn[byte 3] > wRm[byte 3]) ? 0xFF : 0x00
  wRd[byte 2] = (wRn[byte 2] > wRm[byte 2]) ? 0xFF : 0x00
  wRd[byte 1] = (wRn[byte 1] > wRm[byte 1]) ? 0xFF : 0x00
  wRd[byte 0] = (wRn[byte 0] > wRm[byte 0]) ? 0xFF : 0x00
else if (H Specified) then
  wRd[half 3] = (wRn[half 3] > wRm[half 3]) ? 0xFFFF : 0x0000
  wRd[half 2] = (wRn[half 2] > wRm[half 2]) ? 0xFFFF : 0x0000
  wRd[half 1] = (wRn[half 1] > wRm[half 1]) ? 0xFFFF : 0x0000
  wRd[half 0] = (wRn[half 0] > wRm[half 0]) ? 0xFFFF : 0x0000
else if (W Specified) then
  wRd[word 1] = (wRn[word 1] > wRm[word 1]) ? 0xFFFFFFFF : 0x00000000
  wRd[word 0] = (wRn[word 0] > wRm[word 0]) ? 0xFFFFFFFF : 0x00000000
```

Exceptions

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				wws1				wRn				wRd				0000				011			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11
S	s	1
U	s	0

SIMD Flags Sets SIMD PSR flags; does not effect the Saturation flags (wCSSF).

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

Note

A.26 WLDR

Overview Performs a load from memory into either a Data register (wRd) or a Control register (wCx); 8-,16-, 32-, and 64-bit data values can be loaded in the Data registers and 32-bit values only can be loaded into the Control registers; loaded data is zero-extended.

Usage WLDR<B,H,W,D>{Cond} wRd, <address_mode>
WLDRW wCx, <address_mode>

Qualifiers
B – Load Byte
H – Load Half
W – Load Word
D – Load Double

Operation

```
if (B specified) then
  wRd[7:0] = Mem[<address_mode>] ; wRd[63:8]=0;
else if (H specified) then
  wRd[15:0] = Mem[<address_mode>] ; wRd[63:16]=0;
else if (W specified) then
  wRd[31:0] = Mem[<address_mode>] ; wRd[63:32]=0;
else if (D Specifed) then
  wRd[63:0] = Mem[<address_mode>] ;
```

Exceptions None

Encoding WLDR<B,H,W,D>{Cond} wRd

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				110			P	U	N	W	1	Rn				wRd				000			M	offset_8							

Sub-Field EncodingWLDR<B,H,W,D>{Cond} wRd

Qualifier	Field	Value
B	N,M	0,0
H	N,M	1,0
W	N,M	0,1
D	N,M	1,1

P,U,W and offset_8 encoding is as specified in the ARM* V5 architecture.

Encoding WLDRW wCx

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111				110			P	U	0	W	1	Rn				wCx				0001				offset_8							

Sub-Field Encoding
P,U,W and offset_8 encoding is as specified in the ARM* V5 architecture.

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note: The data address is generated by the Intel® XScale™ microarchitecture where <address_mode> is allowable modes specified in ARM* load coprocessor. The offset used in the address calculation is always specified to the assembler in bytes. However for the W and D qualifiers the offset must be a word multiple (i.e divisible by 4). This allows the word- and double-load variants to have a greater offset range (0 – 0x3FC) than the byte-and half-word load variants (0 -0xFF). See Table A-4.

Table A-4. Load Address Mode Assembler Offset Specification and Instruction Encoding

Flag	Assembler Offset Units (AS_offset)	Assembler Offset Restriction	Instruction encoding (offset_8)
B	Byte	AS_offset < 0xFF	AS_offset[7:0]
H	Byte	AS_offset < 0xFF	AS_offset[7:0]
W	Byte	AS_offset < 0x3FC & AS_offset mod 4 = 0	AS_offset[[9:2]
D	Byte	AS_offset < 0x3FC & AS_offset mod 4 =	AS_offset[9:2]

If an offset is required that does not meet the above restrictions, the address must be modified using a separate instruction (e.g., add/subtract).

Addresses should be aligned at the byte/half/word/double boundary of the data item being load. Lower address bits should be cleared if the unaligned address traps is enabled on the main core, otherwise unaligned address trap is taken.

If unaligned traps are disabled, this table illustrates what happens.

Qualifier	Unaligned Behavior
B	never unaligned
H,W,D	unpredictable

Note: Load to Wireless MMX™ technology Control registers are word only and cannot be conditionally executed.

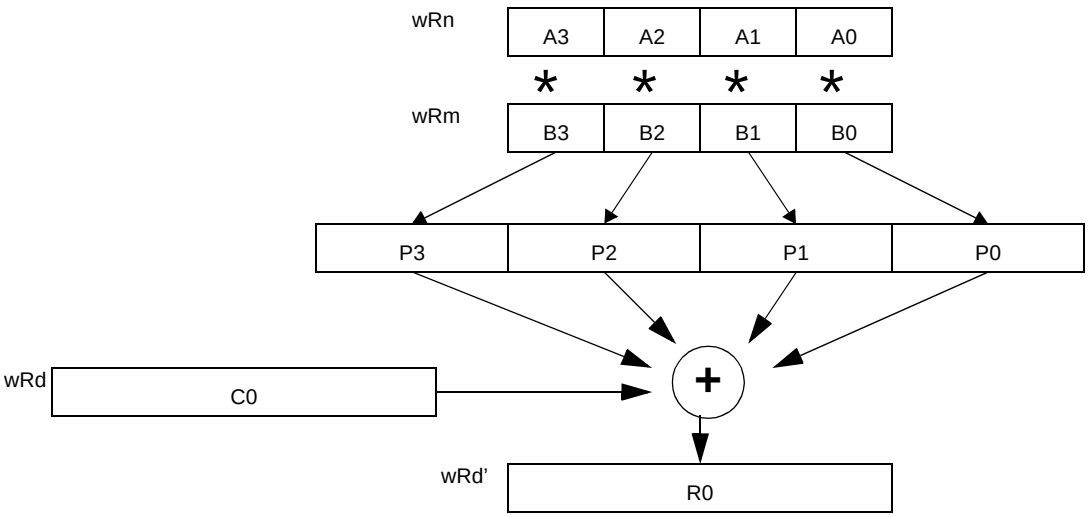
A.27 WMAC

Overview Performs a vector multiplication of wRn and wRm and can accumulate the result with wRd on vectors of 16-bit data only.

Usage WMAC<U,S>{Z}{Cond} wRd, wRn, wRm

Qualifiers
Z – Zero Destination before accumulation
S – Signed
U – Unsigned

`Operation`



Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				01sz				wRn				wRd				0001				000			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
S	s	1
U	s	0
Z	z	1
No Z	z	0

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note The input arguments are (Ax,Bx) 16-bits, the intermediate values (Px) are 32-bits and the result is 64-bits. Signed and unsigned qualifier applies to both multiplication and the final accumulation. Sign extension is performed, where appropriate, to the intermediate results.

A.28 WMADD

Overview Performs a 16-bit vector multiplication and then sums the lower two products into the bottom word of wRd, and the upper two products into the upper word of wRd; intermediate products are 32 bit; can operate on either signed or unsigned data.

Usage WMADD<U,S>{Cond} wRd, wRn, wRm

Qualifiers S – Signed
U – Unsigned

Operation

```
wRd[31:0] = (wRn[15:0]*wRm[15:0]) + (wRn[31:16]*wRm[31:16]);
wRd[63:32] = (wRn[47:32]*wRm[47:32]) + (wRn[63:48]*wRm[63:48]);
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				10s0				wRn				wRd				0001				000			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
S	s	1
U	s	0

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note If the result is larger than the result field, the result just truncates; it does not saturate (this is the same behavior as the MMX™ technology PMADDWM instruction.); overflow can be prevented by prescaling the input arguments.

A.29 WMAX

Overview Performs vector maximum selection of elements from wRn and wRm for vectors of 8-, 16-, or 32-bit data and places the maximum fields in the Destination register, wRd; can be performed on signed or unsigned data.

Usage WMAX<U,S><B,H,W>{Cond} wRd, wRn, wRm

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD
S – Signed Comparison
U – Unsigned Comparison

Operation

```
if (B Specified)then
  wRd[byte 7] = (wRn[byte 7] > wRm[byte 7]) ? wRn[byte 7] : wRm[byte 7]
  wRd[byte 6] = (wRn[byte 6] > wRm[byte 6]) ? wRn[byte 6] : wRm[byte 6]
  wRd[byte 5] = (wRn[byte 5] > wRm[byte 5]) ? wRn[byte 5] : wRm[byte 5]
  wRd[byte 4] = (wRn[byte 4] > wRm[byte 4]) ? wRn[byte 4] : wRm[byte 4]
  wRd[byte 3] = (wRn[byte 3] > wRm[byte 3]) ? wRn[byte 3] : wRm[byte 3]
  wRd[byte 2] = (wRn[byte 2] > wRm[byte 2]) ? wRn[byte 2] : wRm[byte 2]
  wRd[byte 1] = (wRn[byte 1] > wRm[byte 1]) ? wRn[byte 1] : wRm[byte 1]
  wRd[byte 0] = (wRn[byte 0] > wRm[byte 0]) ? wRn[byte 0] : wRm[byte 0]
else if (H Specified)then
  wRd[half 3] = (wRn[half 3] > wRm[half 3]) ? wRn[half 3] : wRm[half 3]
  wRd[half 2] = (wRn[half 2] > wRm[half 2]) ? wRn[half 2] : wRm[half 2]
  wRd[half 1] = (wRn[half 1] > wRm[half 1]) ? wRn[half 1] : wRm[half 1]
  wRd[half 0] = (wRn[half 0] > wRm[half 0]) ? wRn[half 0] : wRm[half 0]
else if (W Specified) then
  wRd[word 1] = (wRn[word 1] > wRm[word 1]) ? wRn[word 1] : wRm[word 1]
  wRd[word 0] = (wRn[word 0] > wRm[word 0]) ? wRn[word 0] : wRm[word 0]
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				wws0				wRn				wRd				0001				011			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11
S	s	1
U	s	0

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note

A.30 WMIN

Overview Performs vector minimum selection of elements from wRn and wRm for vectors of 8-, 16-, or 32-bit data, and places the minimum fields in the Destination register, wRd; can be performed on signed or unsigned data.

Usage WMIN<U,S><B,H,W>{Cond} wRd, wRn, wRm

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD
S – Signed Comparison
U – Unsigned Comparison

Operation

```
if (B Specified) then
  wRd[byte 7] = (wRn[byte 7] < wRm[byte 7]) ? wRn[byte 7] : wRm[byte 7]
  wRd[byte 6] = (wRn[byte 6] < wRm[byte 6]) ? wRn[byte 6] : wRm[byte 6]
  wRd[byte 5] = (wRn[byte 5] < wRm[byte 5]) ? wRn[byte 5] : wRm[byte 5]
  wRd[byte 4] = (wRn[byte 4] < wRm[byte 4]) ? wRn[byte 4] : wRm[byte 4]
  wRd[byte 3] = (wRn[byte 3] < wRm[byte 3]) ? wRn[byte 3] : wRm[byte 3]
  wRd[byte 2] = (wRn[byte 2] < wRm[byte 2]) ? wRn[byte 2] : wRm[byte 2]
  wRd[byte 1] = (wRn[byte 1] < wRm[byte 1]) ? wRn[byte 1] : wRm[byte 1]
  wRd[byte 0] = (wRn[byte 0] < wRm[byte 0]) ? wRn[byte 0] : wRm[byte 0]
else if (H Specified) then
  wRd[half 3] = (wRn[half 3] < wRm[half 3]) ? wRn[half 3] : wRm[half 3]
  wRd[half 2] = (wRn[half 2] < wRm[half 2]) ? wRn[half 2] : wRm[half 2]
  wRd[half 1] = (wRn[half 1] < wRm[half 1]) ? wRn[half 1] : wRm[half 1]
  wRd[half 0] = (wRn[half 0] < wRm[half 0]) ? wRn[half 0] : wRm[half 0]
else if (W Specified) then
  wRd[word 1] = (wRn[word 1] < wRm[word 1]) ? wRn[word 1] : wRm[word 1]
  wRd[word 0] = (wRn[word 0] < wRm[word 0]) ? wRn[word 0] : wRm[word 0]
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				wws1				wRn				wRd				0001				011			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11
S	s	1
U	s	0

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note

A.31 WMOV

Overview This pseudo-instruction moves register wRn to register wRd; this instruction is a form of WOR.

Usage WMOV{Cond} wRd, wRn

Qualifiers None

Operation `WOR wRd,wRn,wRn`

Exceptions None

Encoding see WOR

Sub-Field Encoding see WOR

SIMD Flags see WOR

Note This is a pseudo-instruction that maps onto another Wireless MMX™ technology instruction and is provided for convenience.

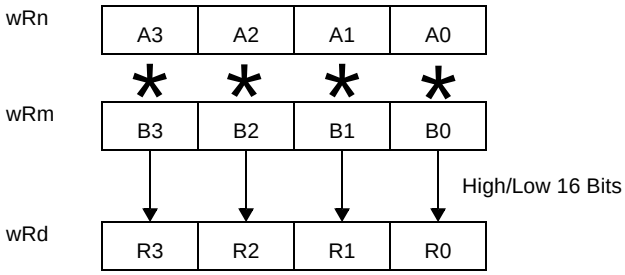
A.32 WMUL

Overview Performs a vector multiplication of wRn and wRm on vectors of 16-bit data only; M-qualifier indicates that the higher order 16 bits of the result are to be stored in wRd, the L-qualifier indicates that the lower 16 bits of the result are to be stored in wRd; can be performed on signed or unsigned data.

Usage WMUL<U,S><M,L>{Cond} wRd, wRn, wRm

Qualifiers
M – Most significant bits of the result to be saved
L – Least significant bits of the result to be saved
S – Signed
U – Unsigned

`Operation`



Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				00sf				wRn				wRd				0001				000			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
S	s	1
U	s	0
M	f	1
L	f	0

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note When L is specified the U and S qualifiers produce the same result.

A.33 WOR

Overview Performs a bit-wise logical “OR” between wRn and wRm and places the result in the Destination register, wRd.

Usage WOR{Cond} wRd, wRn, wRm

Qualifiers None

Operation

```
wRd = wRn | wRm
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				0000				wRn				wRd				0000				000			0	wRm			

Sub-Field Encoding None

SIMD Flags Sets SIMD64 flags only; C and V flags are cleared and the N and Z flags are set according to the result. Does not effect the Saturation flags (wCSSF).

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

Note

A.34 WPACK

Overview Packs data from wRn and wRm into wRd, with wRm being packed into the upper half, and wRn being packed into the lower half for vectors of 16-, 32-, or 64-bit data, and saturate the results and place in the destination register wRd; packing can be performed with signed saturation or unsigned saturation.

Usage WPACK<H,W,D><US,SS>{Cond} wRd, wRn, wRm

Qualifiers
H – Pack 16-bit (Half Word) into 8-bits
W – Pack 32-bit (Word) into 16-bits
D – Pack 64-bit (Double) into 32-bits
SS – Signed Saturation
US – Unsigned Saturation

Operation

```
if (H Specified)then
  wRd[byte 7] = saturate(wRm[half 3], {US,SS}, 8)
  wRd[byte 6] = saturate(wRm[half 2], {US,SS}, 8)
  wRd[byte 5] = saturate(wRm[half 1], {US,SS}, 8)
  wRd[byte 4] = saturate(wRm[half 0], {US,SS}, 8)
  wRd[byte 3] = saturate(wRn[half 3], {US,SS}, 8)
  wRd[byte 2] = saturate(wRn[half 2], {US,SS}, 8)
  wRd[byte 1] = saturate(wRn[half 1], {US,SS}, 8)
  wRd[byte 0] = saturate(wRn[half 0], {US,SS}, 8)
else if (W Specified)then
  wRd[half 3] = saturate(wRm[word 1], {US,SS}, 16)
  wRd[half 2] = saturate(wRm[word 0], {US,SS}, 16)
  wRd[half 1] = saturate(wRn[word 1], {US,SS}, 16)
  wRd[half 0] = saturate(wRn[word 0], {US,SS}, 16)
else if (D Specified) then
  wRd[word 1] = saturate(wRm, {US,SS}, 32)
  wRd[word 0] = saturate(wRn, {US,SS}, 32)
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				wwss				wRn				wRd				0000				100			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
H	ww	01
W	ww	10
D	ww	11
Reserved	ww	00
US	ss	01
SS	ss	11
Reserved	ss	00
Reserved	ss	10

SIMD Flags Sets Saturation flags if corresponding field saturates (US, SS)
The C and V are cleared and the Z, N set according to the final result of the pack operation.

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

SIMD flags are set on the SIMD width of the result (see below).

Qualifier	SIMD flags
H	SIMD8
W	SIMD16
D	SIMD32

Note The US and SS refer to the saturation limits applied not to the type of the incoming arguments. The incoming arguments are always treated as signed values (same as MMX™ technology.)

A.35 WROR

Overview Performs vector logical rotate-right of wRn by wRm for vectors of 16-, 32-, or 64-bit data and places the result in the Destination register, wRd.

Usage WROR<H,W,D>{Cond} wRd, wRn, wRm
WROR<H,W,D>G{Cond} wRd, wRn, wCGRm

Qualifiers
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD
D – 64-bit (Double)
G – wRm field specifies general purpose register to use for shift value

Operation

```
if (H Specified) then
  wRd[half 3] = wRn[half 3] rotate_by ((G Specified) ?
wCGRm[7:0]:wRm[7:0])
  wRd[half 2] = wRn[half 2] rotate_by ((G Specified) ?
wCGRm[7:0]:wRm[7:0])
  wRd[half 1] = wRn[half 1] rotate_by ((G Specified) ?
wCGRm[7:0]:wRm[7:0])
  wRd[half 0] = wRn[half 0] rotate_by ((G Specified) ?
wCGRm[7:0]:wRm[7:0])
else if (W Specified) then
  wRd[word 1] = wRn[word 1] rotate_by ((G Specified) ?
wCGRm[7:0]:wRm[7:0])
  wRd[word 0] = wRn[word 0] rotate_by ((G Specified) ?
wCGRm[7:0]:wRm[7:0])
else if (D Specified) then
  wRd = wRn rotate_by ((G Specified) ? wCGRm[7:0]:wRm[7:0])
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ww11				wRn				wRd				000g				010			0	wRm/wCGRn			

Sub-Field Encoding

Qualifier	Field	Value
H	ww	01
W	ww	10
D	ww	11
Reserved	ww	00
G	g	1
No G	g	0

SIMD Flags Clears C and V flags, and sets Z and N according to the result. Does not effect the Saturation flags (wCSSF).

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

Note Specifying an illegal value for wCGRm when the G qualifier is used causes an undefined instruction exception.

A.36 WSAD

Overview Performs the sum of absolute differences of wRn and wRm, and accumulates the result with wRd; can be applied to 8-bit or 16-bit unsigned data vectors.

Usage WSAD<B,H>{Z}{Cond} wRd, wRn, wRm

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
Z – Zero Accumulator First

Operation

```
wRd[word 1] = 0;
if (B specified)then
wRd[word 0] = (Z Specified) ? 0: wRd[word 0]
  + abs(wRn[byte 7] - wRm[byte 7])
  + abs(wRn[byte 6] - wRm[byte 6])
  + abs(wRn[byte 5] - wRm[byte 5])
  + abs(wRn[byte 4] - wRm[byte 4])
  + abs(wRn[byte 3] - wRm[byte 3])
  + abs(wRn[byte 2] - wRm[byte 2])
  + abs(wRn[byte 1] - wRm[byte 1])
  + abs(wRn[byte 0] - wRm[byte 0])
else if (H specified)then
  wRd[word 0] = Z Specified) ? 0: wRd[word 0]
  + abs(wRn[half 3] - wRm[half 3])
  + abs(wRn[half 2] - wRm[half 2])
  + abs(wRn[half 1] - wRm[half 1])
  + abs(wRn[half 0] - wRm[half 0])
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				0h0z				wRn				wRd				0001				001			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
B	h	0
H	h	1
Z	z	1
No Z	z	0

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note Accumulates the result to 32-bits only

A.37 WSHUFH

Overview Select (shuffle) 16-bit data values in Destination register, wRd, from 16-bit fields in Source register specified by the 8-bit immediate value.

Usage WSHUFH{Cond} wRd, wRn, #Imm8

Qualifiers None

Operation

```
wRd[Half 0]=wRn[Half Imm8[1:0]]
wRd[Half 1]=wRn[Half Imm8[3:2]]
wRd[Half 2]=wRn[Half Imm8[5:4]]
wRd[Half 3]=wRn[Half Imm8[7:6]]
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ddcc				wRn				wRd				0001				111			0	bbaa			

Sub-Field Encoding

Field	Value
bbaa	Imm8[3:0]
ddcc	Imm8[7:4]

SIMD Flags Sets Z and N flag, clears C, and V; sets SIMD16 flags only; does not effect the Saturation flags (wCSSF).

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

Note

A.38 WSLL

Overview Performs vector logical shift-left of wRn by wRm for vectors of 16-, 32-, or 64-bit data and places the result in wRd.

Usage WSLL<H,W,D>{Cond} wRd, wRn, wRm
WSLL<H,W,D>G {Cond} wRd, wRn, wCGRn

Qualifiers
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD
D – 64-bit (Double)
G – wRm field specifies general purpose register to use for shift value

Operation

```
if (H Specified)then
  wRd[half 3] = wRn[half 3] << ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[half 2] = wRn[half 2] << ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[half 1] = wRn[half 1] << ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[half 0] = wRn[half 0] << ((G Specified) ? wCGRm[7:0]:wRm[7:0])
else if (W Specified) then
  wRd[word 1] = wRn[word 1] << ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[word 0] = wRn[word 0] << ((G Specified) ? wCGRm[7:0]:wRm[7:0])
else if (D Specified) then
  wRd = wRn << ((G Specified) ? wCGRm[7:0]:wRm[7:0])
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ww01				wRn				wRd				000g				010			0	wRm/wCGRn			

Sub-Field Encoding

Qualifier	Field	Value
H	ww	01
W	ww	10
D	ww	11
Reserved	ww	00
G	g	1
No G	g	0

SIMD Flags The V, C flags are cleared; the Z and N flag are set as a result of the operation,; does not effect the Saturation flags (wCSSF).

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

Note Zero is shifted in from the right; if shift value > 63 (for D qualifier), 31 (for W qualifier) or 15 (for H qualifier), the Destination register contains all zeros. Specifying an illegal value for wCGRm when the G qualifier is used causes an undefined instruction exception.

A.39 WSRA

Overview Performs vector arithmetic shift-right of wRn by wRm for vectors of 16-, 32-, or 64-bit data and places the result in wRd.

Usage WSRA<H,W,D>{Cond} wRd, wRn, wRm
WSRA<H,W,D>G{Cond} wRd, wRn, wCGRn

Qualifiers
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD
D – 64-bit (Double)
G – wRm field specifies general purpose register to use for shift value

Operation

```
if (H Specified)then
  wRd[half 3] = wRn[half 3] >> ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[half 2] = wRn[half 2] >> ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[half 1] = wRn[half 1] >> ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[half 0] = wRn[half 0] >> ((G Specified) ? wCGRm[7:0]:wRm[7:0])
else if (W Specified) then
  wRd[word 1] = wRn[word 1] >> ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[word 0] = wRn[word 0] >> ((G Specified) ? wCGRm[7:0]:wRm[7:0])
else if (D Specified) then
  wRd = wRn >> ((G Specified) ? wCGRm[7:0]:wRm[7:0])
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ww00				wRn				wRd				000g				010			0	wRm/wCGRn			

Sub-Field Encoding

Qualifier	Field	Value
H	ww	01
W	ww	10
D	ww	11
Reserved	ww	00
G	g	1
No G	g	0

SIMD Flags The C and V flags are cleared; the and Z and N flags are set as a result of the operation; does not effect the Saturation flags (wCSSF).

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

Note The original *sign* bit is shifted in from the left; if the shift value > 63 (for D qualifier), 31 (for W qualifier) or 15 (for H qualifier), the Destination register contains all *sign* bits. Specifying an illegal value for wCGRm when the G qualifier is used causes an undefined instruction exception.

A.40 WSRL

Overview Performs vector logical shift-right of wRn by wRm for vectors of 16-, 32-, or 64-bit data and places the result in wRd.

Usage WSRL<H,W,D>{Cond} wRd, wRn, wRm
WSRL<H,W,D>G{Cond} wRd, wRn, wCGRn

Qualifiers
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD
D – 64-bit (Double)
G – wRm field specifies general purpose register to use for shift value

Operation

```
if (H Specified)then
  wRd[half 3] = wRn[half 3] >> ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[half 2] = wRn[half 2] >> ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[half 1] = wRn[half 1] >> ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[half 0] = wRn[half 0] >> ((G Specified) ? wCGRm[7:0]:wRm[7:0])
else if (W Specified) then
  wRd[word 1] = wRn[word 1] >> ((G Specified) ? wCGRm[7:0]:wRm[7:0])
  wRd[word 0] = wRn[word 0] >> ((G Specified) ? wCGRm[7:0]:wRm[7:0])
else if (D Specified) then
  wRd = wRn >> ((G Specified) ? wCGRm[7:0]:wRm[7:0])
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ww10				wRn				wRd				000g				010			0	wRm/wCGRn			

Sub-Field Encoding

Qualifier	Field	Value
H	ww	01
W	ww	10
D	ww	11
Reserved	ww	00
G	g	1
No G	g	0

SIMD Flags The C and V flags are cleared; the Z and N flags are set as a result of the operation; does not effect the Saturation flags (wCSSF).

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

Note Zero is shifted in from the left; if the shift value > 63 (for D qualifier), 31 (for W qualifier) or 15 (for H qualifier), the Destination register contains all zeros. Specifying an illegal value for wCGRm when the G qualifier is used causes an undefined instruction exception.

A.41 WSTR

Overview Performs a store from the Source register (either wRm or wCx) into the memory location whose address is specified by the Rn register and corresponding address modifiers; 8-, 16-, 32-, and 64-bit data values can be stored from the Data registers (wRm), and 32-bits only can be stored from the Control registers (wCx).

Usage WSTR<B,H,W,D>{Cond} wRm, <address_mode>
WSTRW wCx, <address_mode>

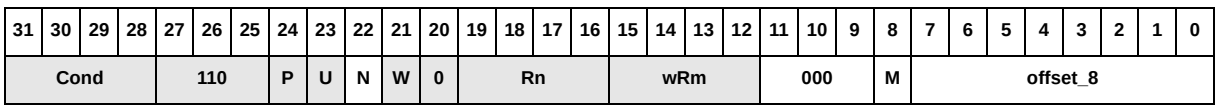
Qualifiers
B – Store Byte
H – Store Half
W – Store Word
D – Store Double

Operation

```
if (B specified) then
  Mem[<address_mode>] = wRm[7:0];
else if (h specified) then
  Mem[<address_mode>] = wRm[15:0];
else if (W specified) then
  Mem[<address_mode>] = wRm[31:0];
else if (D specified) then
  Mem[<address_mode>] = wRm[63:0];
```

Exceptions

Encoding WSTR<B,H,W,D>



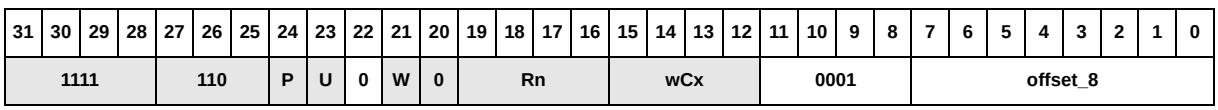
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				110			P	U	N	W	0	Rn				wRm				000			M	offset_8							

Sub-Field EncodingWSTR<B,H,W,D>

Qualifier	Field	Value
B	N,M	0,0
H	N,M	1,0
W	N,M	0,1
D	N,M	1,1

P,U,W and offset_8 encoding is as specified in the ARM* V5 architecture.

Encoding WSTRW wCx



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111				110			P	U	0	W	0	Rn				wCx				0001				offset_8							

Sub-Field EncodingP,U,W, and offset_8 encoding is as specified in the ARM* V5 architecture.

SIMD Flags Does not effect the SIMD PSR flags (wCASF) or Saturation flags (wCSSF).

Note: The Data address is generated by the Intel® XScale™ microarchitecture where <address_mode> is allowable modes specified in ARM* store coprocessor. The offset used in the address calculation is always specified to the assembler in bytes. However for the W and D qualifiers the offset must be a word multiple (i.e divisible by 4). This allows the word- and double-store variants to have a greater offset range (0 – 0x3FC) than the byte- and half-word store variants (0 -0xFF). See Table A-5.

Table A-5. Store Address Mode Assembler Offset Specification and Instruction Encoding

Qualifier	Assembler Offset Units (AS_offset)	Assembler Offset Restriction	Instruction encoding (offset_8)
B	Byte	AS_offset =< 0xFF	AS_offset[7:0]
H	Byte	AS_offset =< 0xFF	AS_offset[7:0]
W	Byte	AS_offset =< 0x3FC & AS_offset mod 4 = 0	AS_offset[[9:2]
D	Byte	AS_offset =< 0x3FC & AS_offset mod 4 = 0	AS_offset[9:2]

If an offset is required that does not meet the above restrictions, the address must be modified using a separate instruction (e.g., add/subtract).

Addresses should be aligned at the byte/half/word boundary of the data item being stored. Lower *address* bits should be cleared if the unaligned address traps is enabled on the main core, otherwise unaligned address trap is taken.

If unaligned traps are disabled the following occurs.

Qualifier	Unaligned Behavior
B	never unaligned
H,W,D	unpredictable

Note: Stores from Wireless MMX™ technology Control registers are word only and cannot be conditionally executed.

A.42 WSUB

Overview Performs vector subtraction of wRm from wRn for vectors of 8-, 16-, or 32-bit signed or unsigned data, and places the result in wRd. Saturation can be specified as signed, unsigned, or no saturation.

Usage WSUB<B,H,W>{US,SS}{Cond} wRd, wRn, wRm

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD
SS – Signed Saturation
US – Unsigned Saturation

Operation

```
if (B Specified)then
  wRd[byte 7] = saturate(wRn[byte 7] - wRm[byte 7], {US,SS}, 8)
  wRd[byte 6] = saturate(wRn[byte 6] - wRm[byte 6], {US,SS}, 8)
  wRd[byte 5] = saturate(wRn[byte 5] - wRm[byte 5], {US,SS}, 8)
  wRd[byte 4] = saturate(wRn[byte 4] - wRm[byte 4], {US,SS}, 8)
  wRd[byte 3] = saturate(wRn[byte 3] - wRm[byte 3], {US,SS}, 8)
  wRd[byte 2] = saturate(wRn[byte 2] - wRm[byte 2], {US,SS}, 8)
  wRd[byte 1] = saturate(wRn[byte 1] - wRm[byte 1], {US,SS}, 8)
  wRd[byte 0] = saturate(wRn[byte 0] - wRm[byte 0], {US,SS}, 8)
else if (H Specified)then
  wRd[half 3] = saturate(wRn[half 3] - wRm[half 3], {US,SS}, 16)
  wRd[half 2] = saturate(wRn[half 2] - wRm[half 2], {US,SS}, 16)
  wRd[half 1] = saturate(wRn[half 1] - wRm[half 1], {US,SS}, 16)
  wRd[half 0] = saturate(wRn[half 0] - wRm[half 0], {US,SS}, 16)
else if (W Specified) then
  wRd[word 1] = saturate(wRn[word 1] - wRm[word 1], {US,SS}, 32)
  wRd[word 0] = saturate(wRn[word 0] - wRm[word 0], {US,SS}, 32)
```

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				wwss				wRn				wRd				0001				101			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11
US	ss	01
SS	ss	11
No Saturation	ss	00
Reserved	ss	10

SIMD Flags Sets SIMD flags (wCASF) on the result of the subtraction wRn-wRm. If US/SS is specified and saturation occurs, the SIMD flags (wCASF) are set on the post-saturation (final result) value. The Saturation flags (wCSSF) are set if the US/SS qualifier is specified and the corresponding fields saturates.

Flag	No Saturation Specified	US/SS specified and no Result Saturation	US/SS specified and Result Saturates
N	Set on Final Result	Set on Final Result	Set on Final Result
Z	Set on Final Result	Set on Final Result	Set on Final Result
C	Set on Final Result	Set on Final Result	Cleared
V	Set on Final Result	Set on Final Result	Cleared

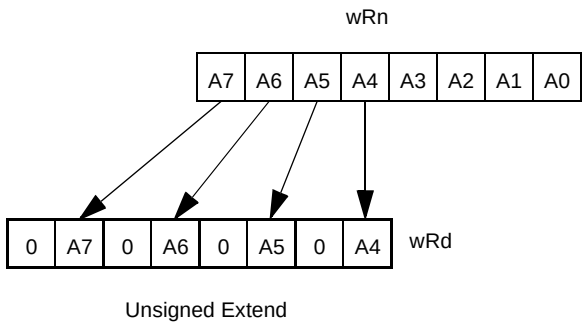
A.43 WUNPCKEH

Overview Unpacks 8-bit, 16-bit, or 32-bit data from the top half of wRn (the Source register), and either zero- or signed-extends each field, and places the result into the Destination register, wRd.

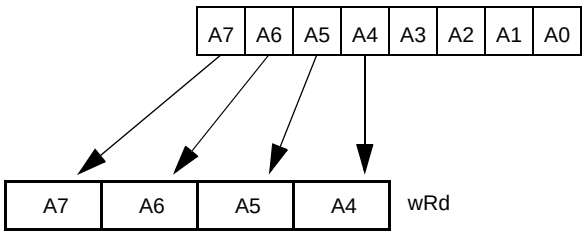
Usage WUNPCKEH<U,S><B,H,W>{Cond} wRd, wRn

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD
U – Unsigned
S – Sign extend

Operation `WUNPCKEHUB wRd,wRn`



Unsigned Extend



Signed Extend

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				wws0				wRn				wRd				0000				110			0	0000			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11
U	s	0
S	s	1

SIMD Flags Sets N and Z flags, depending on the result fields, and clears C and V flags.

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

SIMD flags are set on the SIMD width of the result (see below.)

Qualifier	SIMD flags
B	SIMD16
H	SIMD32
W	SIMD64

Note

A.44 WUNPCKIH

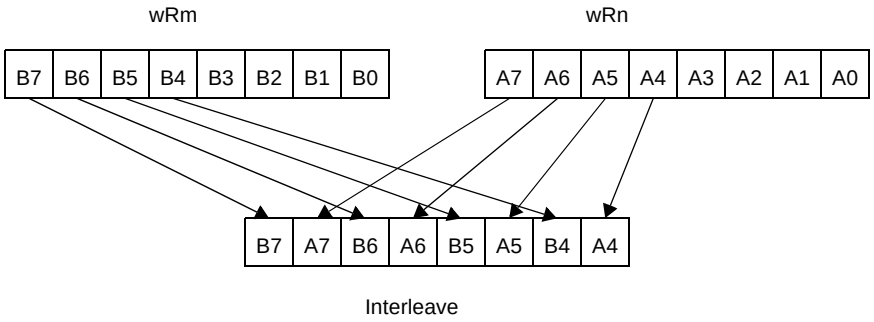
Overview Unpacks either 8-bit, 16-bit, or 32 bit data from the top half of wRn, interleaves with the top half of wRm, and places the result into the Destination register, wRd.

Usage WUNPCKIH<B,H,W>{Cond} wRd, wRn, wRm

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD

Operation

```
WUNPCKIHB wRd,wRn,wRm
```



Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ww01				wRn				wRd				0000				110			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11

SIMD Flags Sets N and Z flags, depending on the result fields, and clears C and V flags.

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

The SIMD flags are set on the SIMD width of the result (see table below.)

Qualifier	SIMD flags
B	SIMD8
H	SIMD16
W	SIMD32

Note

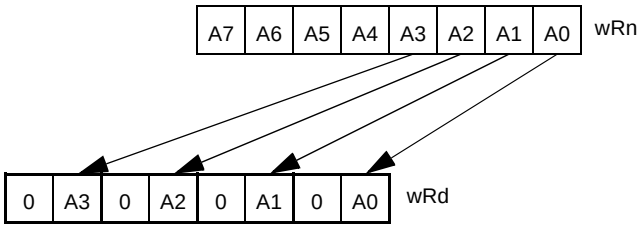
A.45 WUNPCKEL

Overview Unpacks either 8-bit, 16-bit, or 32 bit data from the lower half of wRn (the Source register), and either zero- or sign-extends each field, and places the result into the Destination register, wRd.

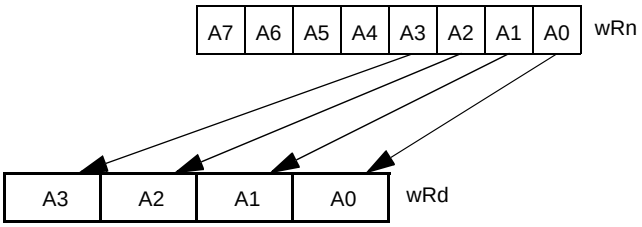
Usage WUNPCKEL<U,S><B,H,W>{Cond} wRd, wRn

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD
U – Unsigned
S – Sign extend

Operation `WUNPCKELB wRn`



Unsigned Extend



Signed Extend

Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				wws0				wRn				wRd				0000				111			0	0000			

Sub-Field Encoding

Qualifier	Field	Value	SIMD
B	ww	00	
H	ww	01	
W	ww	10	
Reserved	ww	11	
U	s	0	
S	s	1	

SIMD Flags Sets N and Z flags, depending on the result fields, and clears C and V flags.

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

SIMD flags are set on the SIMD width of the result (see below.)

Qualifier	SIMD flags
B	SIMD16
H	SIMD32
W	SIMD64

Note

A.46 WUNPCKIL

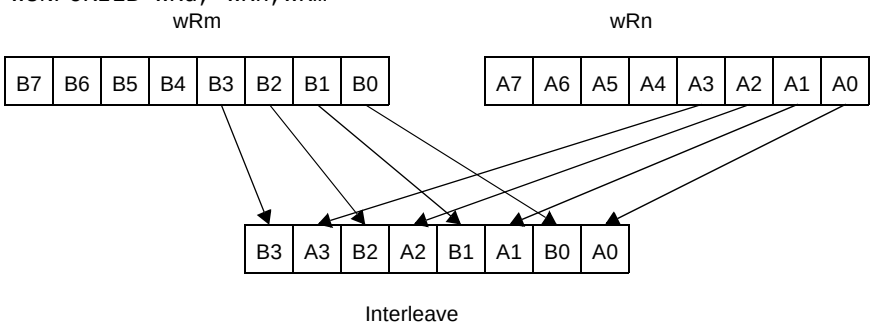
Overview Unpacks either 8-bit, 16-bit, or 32 bit data from the lower half of wRn and the lower half of wRm, and places the result into the Destination register, wRd.

Usage WUNPCKIL<B,H,W>{Cond} wRd, wRn,wRm

Qualifiers
B – 8-bit (Byte) SIMD
H – 16-bit (Half Word) SIMD
W – 32-bit (Word) SIMD

Operation

```
WUNPCKILB wRd, wRn,wRm
```



Exceptions None

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				ww01				wRn				wRd				0000				111			0	wRm			

Sub-Field Encoding

Qualifier	Field	Value
B	ww	00
H	ww	01
W	ww	10
Reserved	ww	11

SIMD Flags Sets N and Z flags, depending on the result fields, and clears C and V flags.

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

SIMD flags are set on the SIMD width of the result (see table below.)

Qualifier	SIMD flags
B	SIMD8
H	SIMD16
W	SIMD32

Note

A.47 WXOR

Overview Performs a bit-wise logical “XOR” between wRn and wRm, and places the result in wRd.

Usage WXOR{Cond} wRd, wRn, wRm

Qualifiers None

Operation `wRd = wRn ^ wRm`

Exceptions

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				0001				wRn				wRd				0000				000			0	wRm			

Sub-Field Encoding None

SIMD Flags Sets SIMD64 flags only; C and V flags are cleared and the N and Z flags are set according to the result; does not affect the Saturation flags (wCSSF).

Flag	Action
N	Set on Final Result
Z	Set on Final Result
C	Cleared
V	Cleared

Note

A.48 WZERO

Overview This pseudo-instruction zeros the Wireless MMX™ technology Destination register, wRd; this instruction is a form of the WANDN instruction.

Usage WZERO{Cond} wRd

Qualifiers None

Operation

```
WANDN wRd, wRd, wRd
```

Exceptions None

Encoding see WANDN

Sub-Field Encoding see WANDN

SIMD Flags see WANDN

Note This is a pseudo-instruction that maps onto another Wireless MMX™ technology instruction and is provided for convenience.

Instruction Encoding Summary B

This appendix gives an overview of the instruction encoding for the entire Wireless MMX™ instruction set.

B.1 Coprocessor Data (CDP) Instructions

Encoded instructions that operate on the Coprocessor Register file use the ARM* CDP format as show below.

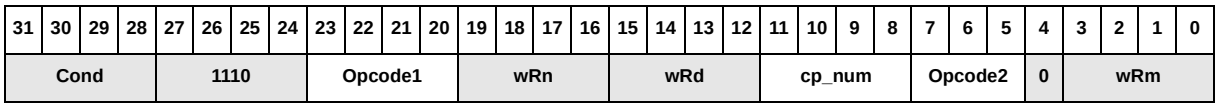


Table B-1 summarizes the encoding for CDP instructions.

Table B-1. CDP Encoding (Sheet 1 of 2)

Type	Instruction	Opcode2	Opcode1	cp_num	wRm
Misc	WOR	000	0000	0000	-
	WXOR		0001	0000	-
	WAND		0010	0000	-
	WANDN		0011	0000	
	WAVG2		1h0r	0000	-
	reserved		01--, 101-,111-	0000	-
Align	WALIGNI	001	0vvv	0000	-
	WALIGNR		10vv	0000	-
	reserved		11--	0000	
Shift	WSRA	010	ww00	000g	-
	WSLL		ww01	000g	-
	WSRL		ww10	000g	-
	WROR		ww11	000g	-
Compare	WCMPEQ	011	ww00	0000	-
	WCMPGT		wws1	0000	-
	reserved		--10	0000	
PACK	WPACK	100	wwss	0000	-

Table B-1. CDP Encoding (Sheet 2 of 2)

Type	Instruction	Opcode2	Opcode1	cp_num	wRm
	reserved	101	-	0000	-
UnPack	reserved	110	--11	0000	-
	WUNPCKEH		wws0	0000	-
	WUNPCKIH		ww01	0000	-
	reserved	111	--11	0000	-
	WUNPCKEL		wws0	0000	-
	WUNPCKIL		ww01	0000	-
Multiply	WMUL	000	00sf	0001	-
	WMAC		01sz	0001	-
	WMADD		10s0	0001	-
	reserved		10-1,11--	0001	-
Difference	reserved	001	0---,1-1-	0001	-
	WSAD		0h0z	0001	-
Shift	used for shifts	010	-	0001	-
Max/Min	WMAX	011	wws0	0001	-
	WMIN		wws1	0001	-
Arithmetic	WADD	100	wwss	0001	-
	WSUB	101	wwss	0001	-
	WACC	110	ww00	0001	-
	reserved		--10, ---1	0001	-
Shuffle	WSHUF	111	ddcc	0001	bbaa

B.2 Coprocessor Data Transfer Instructions

Instructions that transfer data between the ARM* and Coprocessor Register file use the MRC, MCR, MRRC or MCRR instruction format (depending on direction and size).

B.2.1 Transfers to Coprocessor Register (MCR)

Transfers to the coprocessor register use the MCR format instruction as show below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				Opcode1			0	wRd				Rn				cp_num				Opcode2			1	wRm			

Table B-2 summarizes the encoding for MCR transfers to the coprocessor register.

Table B-2. MCR Encoding

Instruction	Opcode1	Opcode2	cp_num	wRm
TMCR	000	000	0001	0000
TMIA	001	See Special case below		
TBCST	010	ww0	0000	0000
TINSR	011	ww0		0bbb
reserved	1--,-1-,--1	---	0001	-
	-	1--,-1-,--1		-
	-	-		1---,-1--,--1-,---1
	010	--1	0000	-
	011	--1		-
	1--	--		-

For MIA, these instruction formats apply:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				Opcode1			0	OpCode2				Rs				cp_num[3:1]			wRd				1	Rm			

Table B-3 summarizes the encoding for MIA transfers to the coprocessor register.

Table B-3. MIA Encoding

Instruction	Opcode1	Opcode2	cp_num[3:1]
TMIA	001	0000	000
TMIAPH	001	1000	000
TMIAXY	001	11xy	000
reserved	001	01--	000

B.2.2 Transfers to Coprocessor Register (MCRR)

The TMCRR instruction uses the following format:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				11000100								RdHi				RdLo				cp_num				Opcode2				wRn			

Table B-4 summarizes the encoding for MCRR transfers to the coprocessor register.

Table B-4. TMCRR Encoding

Instruction	Opcode2	cp_num
TMCRR	0000	0000
reserved	0001 – 1111	0000
	-	0001

B.2.3 Transfers from Coprocessor Register (MRC)

Transfers from the coprocessor register using the MRC format are show below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1110				Opcode1			1	wRn				Rd				cp_num				Opcode2			1	wRm			

Table B-5 summarizes the encoding for MCR transfers from the coprocessor register.

Table B-5. MRC Encoding

Instruction	Opcode2	Opcode1	cp_num	wRm
TMRC	000	000	0001	0000
TMOVMSK	001	ww0	0000	0000
TANDC	001	ww0	0001	0000
TORC	010	ww0	0001	0000
TEXTRC	011	ww0	0001	0bbb
TEXTRM		ww0	0000	sbbb
reserved	000	1--,-1-,--1	0001	-
	000	000		1---,-1--,--1-,---1
	001	--1		-
	010	--1		-
	011	--1		-
	1--	-		-
	001	--1	0000	-
	1--,-1-	-		

B.2.4 Transfers from Coprocessor Register (MRRC)

The TMRRC instruction uses the following format:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				11000101								RdHi				RdLo				cp_num				Opcode2				wRn			

Table B-6 summarizes the sub-field encoding for MRRC transfers from the coprocessor register.

Table B-6. TMRRC Encoding

Instruction	Opcode2	cp_num
TMRRC	0000	0000
reserved	0001 – 1111	0000
	-	0001

B.3 Load Store Instructions

B.3.1 Load/Store to Wireless MMX™ technology Data Registers

This instruction uses the ARM* Load/Store Coprocessor instruction so flags are compatible with the N-flag and 1 bit of the coprocessor number encode the 4 size options (Byte, Half, Word, Double Word).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				110			P	U	N	W	L	Rn				wRd				000m				offset_8								

B.3.2 Load/Store to Wireless MMX™ technology Control Registers

This instruction uses the ARM* LDC2/STC2 instruction class, which cannot be conditionally executed.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111				110			P	U	N	W	L	Rn				wRd				0001				offset_8								

Intrinsic Support C

Intrinsics are special coding extensions that allow using the syntax of C function calls and C variables instead of hardware registers. Using these intrinsics frees programmers from having to program in assembly language and manage registers. In addition, the compiler optimizes the instruction scheduling so that executables run faster.

The major benefit of using intrinsics is that you now have access to key features that are not available using conventional coding practices. Intrinsics 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.

Wireless MMX™ instructions use the following new features:

- New registers enable packed data of up to 64 bits in length for optimal SIMD processing.
- New data types enable packing of up to 8 elements of data in one register.

For each computational and data manipulation instruction in the new extension sets, there is a corresponding C intrinsic that implements that instruction directly. This frees you from managing registers and assembly programming. Further, the compiler optimizes the instruction scheduling so that your executable runs faster.

C.1 New Data Types

Wireless MMX™ technology intrinsic functions use a new C data type, the `__m64` data type for some operands, representing the new registers that are used as the operands to these intrinsic functions.

The `__m64` data type is used to represent the contents of Wireless MMX™ technology register, which is the register that is used by the Wireless MMX™ technology intrinsics. The `__m64` data type can hold eight 8-bit values, four 16-bit values, two 32-bit values, or one 64-bit value.

C.1.1 New Data Types Usage Guidelines

Since the new data type is not basic ANSI C data types, you must observe the following usage restrictions:

- Use the new data type only on either side of an assignment, as a return value, or as a parameter. You cannot use it with other arithmetic expressions (“+”, “-”, etc.).
- Use the new data type as objects in aggregates, such as unions to access the byte elements and structures.
- Use the new data type only with the respective intrinsics described in this book. The new data type is supported on both sides of an assignment statement: as parameters to a function call, and as a return value from a function call.

C.2 Naming and Usage Syntax

Most of the Wireless MMX™ technology intrinsic names use a notational convention as follows:

`_mm_intrin_op_suffix`

intrin_op	Indicates the intrinsics basic operation; for example, add for addition and sub for subtraction.
suffix	Denotes the type of data operated on by the instruction. The first one or two letters of each suffix denotes whether the data is packed (p), or scalar (s). The remaining letters denote the type: i64 signed 64-bit integeru64 unsigned 64-bit integeri32 signed 32-bit integeru32 unsigned 32-bit integeri16 signed 16-bit integeru16 unsigned 16-bit integeri8 signed 8-bit integeru8 unsigned 8-bit integer

A number appended to a variable name indicates the element of a packed object. For example, r0 is the lowest word of r. The packed values are represented in right-to-left order, with the lowest value being used for scalar operations.

Due to the nature of the instruction, some intrinsics require their arguments to be immediates (constant integer literals). Some intrinsics are “composite” because they require more than one instruction to map.

C.2.1 Conventions

This appendix uses several notational and typographical conventions to visually differentiate text. The following table explains these conventions.

Convention	Description
Italics	Italics identify variables and introduce new terminology. Titles of manuals are in italic font. Italics are used for emphasis.
Bold	Used to specify something important.
`Plain courier`	Used to specify written code.
`Italic courier`	Used to specify parameters.
UPPERCASE	Used to specify options and commands.
GUI-elements	Elements of the graphical user interface are written in Arial.
<>	Filenames and function names to be defined are delimited by <>.
,...	Used to specify an item which may be repeated.
[ ]	Used to specify optional items.
{ }	Used to specify an optional item which may be repeated.

C.2.2 Intrinsic Syntax

To use Wireless MMX™ technology intrinsics, you must include the file mmintrin.h found in the Intel compiler include directory. A 64-bit data type is defined in this include file. The include file also contains the ANSI C prototypes for these intrinsics.

The syntax of Wireless MMX™ technology intrinsic prototype is:

Syntax: `data_type intrinsic_name (parameters);`

data_type	Is the return data type, which can be either void, int, `__m64`, `__int64`. Intrinsics may return other data types as well, as indicated 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++ code instead of in-lining the actual instruction.
parameters	Represents the parameters required by each intrinsic.

Note: SIMD intrinsics on Floating point data types are not listed in this version of bookbookbook. These intrinsics are available in the next version.

C.3 Wireless MMX™ Technology Arithmetic Intrinsics

Table C-1. Overview of Wireless MMX™ technology Arithmetic Intrinsics (Sheet 1 of 2)

Intrinsic Name	Wireless MMX™ Instruction	MMX™ technology/ SSE Instruction	Operation	Signed	Argument – Values/ Bits	Result- Values/ Bits
_mm_add_pi8	WADDB	PADDB	Addition	--	8/8	8/8
_mm_add_pi16	WADDH	PADDW	Addition	--	4/16	4/16
_mm_add_pi32	WADDW	PADDD	Addition	--	2/32	2/32
_mm_adds_pi8	WADDBSS	PADDSB	Addition	Yes	8/8	8/8
_mm_adds_pi16	WADDHSS	PADDSW	Addition	Yes	4/16	4/16
_mm_adds_pi32	WADDWSS	--	Addition	Yes	2/32	2/32
_mm_adds_pu8	WADDBUS	PADDUSB	Addition	No	8/8	8/8
_mm_adds_pu16	WADDHUS	PADDUSW	Addition	No	4/16	4/16
_mm_adds_pu32	WADDWUS	--	Addition	No	2/32	2/32
_mm_sub_pi8	WSUBB	PSUBB	Subtraction	--	8/8	8/8
_mm_sub_pi16	WSUBH	PSUBW	Subtraction	--	4/16	4/16
_mm_sub_pi32	WSUBW	PSUBD	Subtraction	--	2/32	2/32
_mm_subs_pi8	WSUBBSS	PSUBSB	Subtraction	Yes	8/8	8/8
_mm_subs_pi16	WSUBHSS	PSUBSW	Subtraction	Yes	4/16	4/16
_mm_subs_pi32	WSUBWSS	--	Subtraction	Yes	2/32	2/32
_mm_subs_pu8	WSUBBUS	PSUBUSB	Subtraction	No	8/8	8/8
_mm_subs_pu16	WSUBHS	PSUBUSW	Subtraction	No	4/16	4/16
_mm_subs_pu32	WSUBWUS	--	Subtraction	No	2/32	2/32
_mm_madd_pi16	WMADDS	PMADDWD	Multiplication	--	4/16	2/32
_mm_madd_pu16	WMADDU	--	Multiplication	No	4/16	2/32
_mm_mulhi_pi16	WMULSH	PMULHW	Multiplication	Yes	4/16	4/16 (high)

Table C-1. Overview of Wireless MMX™ technology Arithmetic Intrinsics (Sheet 2 of 2)

Intrinsic Name	Wireless MMX™ Instruction	MMX™ technology/ SSE Instruction	Operation	Signed	Argument – Values/ Bits	Result-Values/ Bits
_mm_mulhi_pu16	WMULUH	PMULHUW	Multiplication	Yes	4/16	4/16 (high)
_mm_mullo_pi16	WMULSL/ WMULUL	PMULLW	Multiplication	--	4/16	4/16 (low)
_mm_mac_pi16	WMACS	--	Multiply-Accumulate	Yes	4/16	4/16
_mm_mac_pu16	WMACU	--	Multiply-Accumulate	No	4/16	4/16
_mm_macz_pi16	WMACSZ	--	Multiply-Accumulate	Yes	4/16	4/16
_mm_macz_pu16	WMACUZ	--	Multiply-Accumulate	No	4/16	4/16
_mm_acc_pu8	WACCB	--	Accumulate	No	8/8	1/8
_mm_acc_pu16	WACCH	--	Accumulate	No	4/16	1/16
_mm_acc_pu32	WACCW	--	Accumulate	No	2/32	1/32
_mm_mia_si64	TMIA	--	Multiply-Accumulate	Yes	--	1/64
_mm_miaph_si64	TMIAPH	--	Multiply-Accumulate	Yes	--	1/64
_mm_miabb_si64	TMIABB	--	Multiply-Accumulate	Yes	--	1/64
_mm_miabt_si64	TMIABT	--	Multiply-Accumulate	Yes	--	1/64
_mm_miatb_si64	TMIATB	--	Multiply-Accumulate	Yes	--	1/64
_mm_miatt_si64	TMIATT	--	Multiply-Accumulate	Yes	--	1/64

C.3.1 _mm_add_pi8

Syntax: `__m64 _mm_add_pi8 (__m64 m1, __m64 m2)`

Description:
Adds the eight 8-bit values in `m1` to the eight 8-bit values in `m2`.

C.3.2 _mm_add_pi16

Syntax: `__m64 _mm_add_pi16 (__m64 m1, __m64 m2)`

Description:
Adds the four 16-bit values in `m1` to the four 16-bit values in `m2`.

C.3.3 _mm_add_pi32

Syntax: `__m64 _mm_add_pi32 (__m64 m1, __m64 m2)`

Description:
Adds the two 32-bit values in `m1` to the two 32-bit values in `m2`.

C.3.4 _mm_adds_pi8

Syntax: `__m64 _mm_adds_pi8 (__m64 m1, __m64 m2)`

Description:
Adds the eight signed 8-bit values in `m1` to the eight signed 8-bit values in `m2` using saturating arithmetic.

C.3.5 _mm_adds_pi16

Syntax: `__m64 _mm_adds_pi16 (__m64 m1, __m64 m2)`

Description:
Adds the four signed 16-bit values in m1 to the four signed 16-bit values in `m2` using saturating arithmetic.

C.3.6 _mm_adds_pi32

Syntax: `__m64 _mm_adds_pi32 (__m64 m1, __m64 m2)`

Description:
Adds the two signed 32-bit values in `m1` to the two signed 32-bit values in `m2` using saturating arithmetic.

C.3.7 _mm_adds_pu8

Syntax: `__m64 _mm_adds_pu8 (__m64 m1, __m64 m2)`

Description:
Adds the eight unsigned 8-bit values in `m1` to the eight unsigned 8-bit values in `m2` using saturating arithmetic.

C.3.8 _mm_adds_pu16

Syntax: `__m64 _mm_adds_pu16 (__m64 m1, __m64 m2)`

Description:
Adds the four unsigned 16-bit values in `m1` to the four unsigned 16-bit values in `m2` using saturating arithmetic.

C.3.9 _mm_adds_pu32

Syntax: `__m64 _mm_adds_pu32 (__m64 m1, __m64 m2)`

Description:
Adds the two unsigned 32-bit values in `m1` to the two unsigned 16-bit values in `m2` using saturating arithmetic.

C.3.10 _mm_sub_pi8

Syntax: `__m64 _mm_sub_pi8 (__m64 m1, __m64 m2)`

Description:
Subtracts the eight 8-bit values in `m2` from the eight 8-bit values in `m1`.

C.3.11 _mm_sub_pi16

Syntax: `__m64 _mm_sub_pi16 (__m64 m1, __m64 m2)`

Description:
Subtracts the four 16-bit values in `m2` from the four 16-bit values in `m1`.

C.3.12 _mm_sub_pi32

Syntax: `__m64 _mm_sub_pi32 (__m64 m1, __m64 m2)`

Description:
Subtracts the two 32-bit values in `m2` from the two 32-bit values in `m1`.

C.3.13 _mm_subs_pi8

Syntax: `__m64 _mm_subs_pi8 (__m64 m1, __m64 m2)`

Description:
Subtracts the eight signed 8-bit values in `m2` from the eight signed 8-bit values in `m1` using saturating arithmetic.

C.3.14 _mm_subs_pi16

Syntax: `__m64 _mm_subs_pi16 (__m64 m1, __m64 m2)`

Description:
Subtracts the four signed 16-bit values in `m2` from the four signed 16-bit values in `m1` using saturating arithmetic.

C.3.15 _mm_subs_pi32

Syntax: `__m64 _mm_subs_pi32 (__m64 m1, __m64 m2)`

Description:
Subtracts the two signed 32-bit values in `m2` from the two signed 32-bit values in `m1` using saturating arithmetic.

C.3.16 _mm_subs_pu8

Syntax: `__m64 _mm_subs_pu8 (__m64 m1, __m64 m2)`

Description:
Subtracts the eight unsigned 8-bit values in `m2` from the eight unsigned 8-bit values in `m1` using saturating arithmetic.

C.3.17 _mm_subs_pu16

Syntax: `__m64 _mm_subs_pu16 (__m64 m1, __m64 m2)`

Description:
Subtracts the four unsigned 16-bit values in `m2` from the four unsigned 16-bit values in `m1` using saturating arithmetic.

C.3.18 _mm_subs_pu32

Syntax: `__m64 _mm_subs_pu32 (__m64 m1, __m64 m2)`

Description:
Subtracts the two unsigned 32-bit values in `m2` from the two unsigned 32-bit values in `m1` using saturating arithmetic.

C.3.19 _mm_madd_pi16

Syntax: `__m64 _mm_madd_pi16 (__m64 m1, __m64 m2)`

Description:
Multiplies four 16-bit values in `m1` by four 16-bit values in `m2` producing four 32-bit intermediate results, which are then summed the lower two products into the bottom word and the upper two products into the upper word of result.

C.3.20 _mm_madd_pu16

Syntax: `__m64 _mm_madd_pu16 (__m64 m1, __m64 m2)`

Description:
Multiplies four unsigned 16-bit values in `m1` by four unsigned 16-bit values in `m2` producing four 32-bit intermediate results, which are then summed the lower products into the bottom word and the upper two products into the upper word of result.

C.3.21 _mm_mulhi_pi16

Syntax: `__m64 _mm_mulhi_pi16(__m64 a, __m64 b)`

Description:
Multiplies four signed 16-bit values in `m1` by four unsigned 16-bit values in `m2` and produces the high 16 bits of the four results.

If `r = _mm_mulhi_pi16(a, b)`, the action will be

```
r0 := hiword(a0 * b0);
r1 := hiword(a1 * b1);
r2 := hiword(a2 * b2);
r3 := hiword(a3 * b3);
```

C.3.22 _mm_mulhi_pu16

Syntax: `__m64 _mm_mulhi_pu16(__m64 a, __m64 b)`

Description:
Multiplies four unsigned 16-bit values in `m1` by four unsigned 16-bit values in `m2` and produces the high 16 bits of the four results.

If `r = _mm_mulhi_pu16(a, b)`, the action will be

```
r0 := hiword(a0 * b0);
r1 := hiword(a1 * b1);
r2 := hiword(a2 * b2);
r3 := hiword(a3 * b3);
```

C.3.23 _mm_mullo_pi16

Syntax: `__m64 _mm_mullo_pi16 (__m64 m1, __m64 m2)`

Description:
Multiplies four 16-bit values in `m1` by four 16-bit values in `m2` and produces the low 16 bits of the four results.

If `r = _mm_mullo_pu16(a, b)`, the action will be

```
r0 := lowword(a0 * b0);
r1 := lowword(a1 * b1);
r2 := lowword(a2 * b2);
r3 := lowword(a3 * b3);
```

C.3.24 _mm_mac_pi16

Syntax: `__m64 _mm_mac_pi16 (__m64 m1, __m64 m2, __m64 m3)`

Description:
Multiplies four signed 16-bit values in signed `m2` by four 16-bit values in `m3` and accumulates result with value in `m1`.

C.3.25 _mm_mac_pu16

Syntax: `__m64 _mm_mac_pu16 (__m64 m1, __m64 m2, __m64 m3)`

Description:
Multiplies four unsigned 16-bit values in unsigned `m2` by four 16-bit values in `m3` and accumulates result with value in `m1`.

C.3.26 _mm_macz_pi16

Syntax: `__m64 _mm_macz_pi16 (__m64 m1, __m64 m2)`

Description:
Multiplies four signed 16-bit values in signed `m1` by four 16-bit values in `m2` and accumulates zero.

C.3.27 _mm_macz_pu16

Syntax: `__m64 _mm_macz_pu16 (__m64 m1, __m64 m2)`

Description:
Multiplies four unsigned 16-bit values in unsigned `m1` by four 16-bit values in `m2` and accumulates result with zero.

C.3.28 _mm_acc_pu8

Syntax: `__m64 _mm_acc_pu8 (__m64 m1)`

Description:
Unsigned accumulate across eight 8-bit values in `m1`.

C.3.29 _mm_acc_pu16

Syntax: `__m64 _mm_acc_pu16 (__m64 m1)`

Description:
Unsigned accumulate across four 16-bit values in `m1`.

C.3.30 _mm_acc_pu32

Syntax: `__m64 _mm_acc_pu32 (__m64 m1)`

Description: Unsigned accumulate across two 32-bit values in `m1`.

C.3.31 _mm_mia_si64

Syntax: `__m64 _mm_mia_si64 (__m64 m1, int a, int b)`

Description:
Multiplies two signed 32-bit values in a & b and accumulates the result with 64-bit values in `m1`.

C.3.32 _mm_miaph_si64

Syntax: `__m64 _mm_miaph_si64 (__m64 m1, int a, int b)`

Description:
Multiplies accumulate signed 16-bit values in `a` & `b` and accumulates the result with 64-bit values in `m1`.

Result = `sign_extend((a[31:16] * b[31:16]) + (a[15:0] * b[15:0])) + m1`

C.3.33 _mm_miabb_si64

Syntax: `__m64 _mm_miabb_si64(__m64 m1, int a, int b)`

Multiplies bottom half of signed 16-bit values in `a` and bottom half of signed 16-bit value in `b` and accumulates the result with 64-bit values in `m1`.

Result = `sign_extend(a[15:0] * b[15:0]) + m1`

C.3.34 _mm_miabt_si64

Syntax: `__m64 _mm_miabt_si64(__m64 m1, int a, int b)`

Description:
Multiplies bottom half of signed 16-bit values in a and top half of signed 16-bit value in b and accumulates the result with 64-bit values in m1.

Result = `sign_extend(a[15:0] * b[31:16]) + m1`

C.3.35 _mm_miatb_si64

Syntax: `__m64 _mm_miatb_si64(__m64 m1, int a, int b)`

Description:
Multiplies top half of signed 16-bit values in a and bottom half of signed 16-bit value in b and accumulates the result with 64-bit values in m1.

Result = `sign_extend(a[31:16] * b[15:0]) + m1`

C.3.36 _mm_miatt_si64

Syntax: `__m64 _mm_miatt_si64(__m64 m1, int a, int b)`

Description:
Multiplies top half of signed 16-bit values in a and top half of signed 16-bit value in b and accumulates the result with 64-bit values in m1.

Result = `sign_extend(a[31:16] * b[31:16]) + m1`

C.4 Wireless MMX™ technology Shift Intrinsics

Table C-2. Overview of Wireless MMX™ technology Shift Intrinsics (Sheet 1 of 2)

Intrinsic Name	Shift Direction	Shift Type	Wireless MMX™ Instruction	MMX™ technology/SSE Instruction
_mm_sll_pi16	Left	Logical	WSLLH	PSLLW
_mm_slli_pi16	Left	Logical	Composite	PSLLWI
_mm_sll_pi32	Left	Logical	WSLLW	PSLLD
_mm_slli_pi32	Left	Logical	Composite	PSLLDI
_mm_sll_si64	Left	Logical	WSLLD	PSLLQ
_mm_slli_si64	Left	Logical	Composite	PSLLQI
_mm_sra_pi16	Right	Arithmetic	WSRAH	PSRAW
_mm_srai_pi16	Right	Arithmetic	Composite	PSRAWI

Table C-2. Overview of Wireless MMX™ technology Shift Intrinsics (Sheet 2 of 2)

Intrinsic Name	Shift Direction	Shift Type	Wireless MMX™ Instruction	MMX™ technology/SSE Instruction
_mm_sra_pi32	Right	Arithmetic	WSRAW	PSRAD
_mm_srai_pi32	Right	Arithmetic	Composite	PSRADI
_mm_sra_si64	Right	Arithmetic	WSRAD	--
_mm_srai_si64	Right	Arithmetic	Composite	--
_mm_srl_pi16	Right	Logical	WSRLH	PSRLW
_mm_srli_pi16	Right	Logical	Composite	PSRLWI
_mm_srl_pi32	Right	Logical	WSRLW	PSRLD
_mm_srli_pi32	Right	Logical	Composite	PSRLDI
_mm_srl_si64	Right	Logical	WSRLD	PSRLQ
_mm_srli_si64	Right	Logical	Composite	PSRLQI
_mm_ror_pi16	Rotate right	Logical	WRORH	--
_mm_ror_pi32	Rotate right	Logical	WRORW	--
_mm_ror_si64	Rotate right	Logical	WRORD	--
_mm_rori_pi16	Rotate right	Logical	Composite	--
_mm_rori_pi32	Rotate right	Logical	Composite	--
_mm_rori_si64	Rotate right	Logical	Composite	--

C.4.1 _mm_sll_pi16

Syntax: `__m64 _mm_sll_pi16 (__m64 m, __m64 count)`

Description:
Shifts four 16-bit values in `m` left the amount specified by `count` while shifting in zeros.

C.4.2 _mm_slli_pi16

Syntax: `__m64 _mm_slli_pi16 (__m64 m, int count)`

Description:
Shifts four 16-bit values in `m` left the amount specified by `count` while shifting in zeros.

C.4.3 _mm_sll_pi32

Syntax: `__m64 _mm_sll_pi32 (__m64 m, __m64 count)`

Description:
Shifts two 32-bit values in `m` left the amount specified by `count` while shifting in zeros.

C.4.4 _mm_slli_pi32

Syntax: `__m64 _mm_slli_pi32 (__m64 m, int count)`

Description:
Shifts two 32-bit values in `m` left the amount specified by `count` while shifting in zeros.

C.4.5 _mm_sll_si64

Syntax: `__m64 _mm_sll_si64 (__m64 m, __m64 count)`

Description:
Shifts the 64-bit value in `m` left the amount specified by `count` while shifting in zeros.

C.4.6 _mm_slli_si64

Syntax: `__m64 _mm_slli_si64 (__m64 m, int count)`

Description:
Shifts the 64-bit value in `m` left the amount specified by `count` while shifting in zeros.

C.4.7 _mm_sra_pi16

Syntax: `__m64 _mm_sra_pi16 (__m64 m, __m64 count)`

Description:
Shifts four 16-bit values in `m` right the amount specified by `count` while shifting in the sign bit.

C.4.8 _mm_srai_pi16

Syntax: `__m64 _mm_srai_pi16 (__m64 m, int count)`

Description:
Shifts four 16-bit values in `m` right the amount specified by `count` while shifting in the sign bit.

C.4.9 _mm_sra_pi32

Syntax: `__m64 _mm_sra_pi32 (__m64 m, __m64 count)`

Description:
Shifts two 32-bit values in `m` right the amount specified by `count` while shifting in the sign bit.

C.4.10 _mm_srai_pi32

Syntax: `__m64 _mm_srai_pi32 (__m64 m, int count)`

Description:
Shifts two 32-bit values in `m` right the amount specified by `count` while shifting in the sign bit.

C.4.11 _mm_sra_si64

Syntax: `__m64 _mm_sra_si64 (__m64 m, __m64 count)`

Description:
Shifts 64-bit value in `m` right the amount specified by `count` while shifting in the sign bit.

C.4.12 _mm_srai_si64

Syntax: `__m64 _mm_srai_si64 (__m64 m, int count)`

Description:
Shifts 64-bit value in `m` right the amount specified by `count` while shifting in the sign bit.

C.4.13 _mm_srl_pi16

Syntax: `__m64 _mm_srl_pi16 (__m64 m, __m64 count)`

Description:
Shifts four 16-bit values in `m` right the amount specified by `count` while shifting in zeros.

C.4.14 _mm_srli_pi16

Syntax: `__m64 _mm_srli_pi16 (__m64 m, int count)`

Description:
Shifts four 16-bit values in `m` right the amount specified by `count` while shifting in zeros.

C.4.15 _mm_srl_pi32

Syntax: `__m64 _mm_srl_pi32 (__m64 m, __m64 count)`

Description:
Shifts two 32-bit values in `m` right the amount specified by `count` while shifting in zeros.

C.4.16 _mm_srli_pi32

Syntax: `__m64 _mm_srli_pi32 (__m64 m, int count)`

Description:
Shifts two 32-bit values in m right the amount specified by count while shifting in zeros.

C.4.17 _mm_srl_si64

Syntax: `__m64 _mm_srl_si64 (__m64 m, __m64 count)`

Description:
Shifts the 64-bit value in `m` right the amount specified by `count` while shifting in zeros.

C.4.18 _mm_srli_si64

Syntax: `__m64 _mm_srli_si64 (__m64 m, int count)`

Description:
Shifts the 64-bit value in `m` right the amount specified by `count` while shifting in zeros.

C.4.19 _mm_ror_pi16

Syntax: `__m64 _mm_ror_pi16 (__m64 m, __m64 count)`

Description:
Rotates four 16-bit values in `m` right the amount specified by `count`.

C.4.20 _mm_ror_pi32

Syntax: `__m64 _mm_ror_pi32 (__m64 m, __m64 count)`

Description:
Rotates two 32-bit values in `m` right the amount specified by `count`.

C.4.21 _mm_ror_si64

Syntax: `__m64 _mm_ror_si64 (__m64 m, __m64 count)`

Description:
Rotates 64-bit value in `m` right the amount specified by `count`.

C.4.22 _mm_rori_pi16

Syntax: `__m64 _mm_rori_pi16 (__m64 m, int count)`

Description:
Rotates four 16-bit values in `m` right the amount specified by `count`.

C.4.23 _mm_rori_pi32

Syntax: `__m64 _mm_rori_pi32 (__m64 m, int count)`

Description:
Rotates two 32-bit values in `m` right the amount specified by `count`.

C.4.24 _mm_rori_si64

Syntax: `__m64 _mm_rori_si64 (__m64 m, int count)`

Description:
Rotates 64-bit value in `m` right the amount specified by `count`.

C.5 Wireless MMX™ technology Logical Intrinsics

Table C-3. Overview of Wireless MMX™ technology Logical Intrinsics

Intrinsic Name	Operation	Wireless MMX™ Instruction	MMX™ technology/SSE Instruction
_mm_and_si64	Bitwise AND	WAND	PAND
_mm_andnot_si64	Logical NOT	WANDN	PANDN
_mm_or_si64	Bitwise OR	WOR	POR
_mm_xor_si64	Bitwise Exclusive OR	WXOR	PXOR

C.5.1 _mm_and_si64

Syntax: `__m64 _mm_and_si64 (__m64 m1, __m64 m2)`

Description:
Performs a bitwise AND of the 64-bit value in `m1` with the 64-bit value in `m2`.

C.5.2 _mm_andnot_si64

Syntax: `__m64 _mm_andnot_si64 (__m64 m1, __m64 m2)`

Description:
Performs a logical NOT on the 64-bit value in `m1` and use the result in a bitwise AND with the 64-bit value in `m2`.

C.5.3 _mm_or_si64

Syntax: `__m64 _mm_or_si64 (__m64 m1, __m64 m2)`

Description:
Performs a bitwise OR of the 64-bit value in `m1` with the 64-bit value in `m2`.

C.5.4 _mm_xor_si64

Syntax: `__m64 _mm_xor_si64 (__m64 m1, __m64 m2)`

Description:
Performs a bitwise XOR of the 64-bit value in `m1` with the 64-bit value in `m2`.

C.6 Wireless MMX™ Technology Compare Intrinsics

Table C-4. Overview of Wireless MMX™ Technology Compare Intrinsics

Intrinsic Name	Comparison	Number of Elements	Element Bit Size	Wireless MMX™ Instruction	MMX™ technology/SSE Instruction
_mm_cmpeq_pi8	Equal	8	8	WCMPEQB	PCMPEQB
_mm_cmpeq_pi16	Equal	4	16	WCMPEQH	PCMPEQW
_mm_cmpeq_pi32	Equal	2	32	WCMPEQW	PCMPEQD
_mm_cmpgt_pi8	Signed Greater Than	8	8	WCMPGTSB	PCMPGTB
_mm_cmpgt_pu8	Unsigned Greater Than	8	8	WCMPGTUB	--
_mm_cmpgt_pi16	Signed Greater Than	4	16	WCMPGTSH	PCMPGTW
_mm_cmpgt_pu16	Unsigned Greater Than	4	16	WCMPGTUH	--
_mm_cmpgt_pi32	Signed Greater Than	2	32	WCMPGTSW	PCMPGTD
_mm_cmpgt_pu32	Unsigned Greater Than	2	32	WCMPGTUW	--

C.6.1 _mm_cmpeq_pi8

Syntax: `__m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)`

Description:
If the respective 8-bit values in `m1` are equal to the respective 8-bit values in `m2`, the function sets the respective 8-bit resulting values to all ones, otherwise it sets them to all zeros.

C.6.2 _mm_cmpeq_pi16

Syntax: `__m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)`

Description:
If the respective 16-bit values in `m1` are equal to the respective 16-bit values in `m2`, the function sets the respective 16-bit resulting values to all ones, otherwise it sets them to all zeros.

C.6.3 _mm_cmpeq_pi32

Syntax: `__m64 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)`

Description:
If the respective 32-bit values in `m1` are equal to the respective 32-bit values in `m2`, the function sets the respective 32-bit resulting values to all ones, otherwise it sets them to all zeros.

C.6.4 _mm_cmpgt_pi8

Syntax: `__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)`

Description:
If the respective 8-bit values in `m1` are greater than the respective 8-bit values in `m2`, the function sets the respective 8-bit resulting values to all ones, otherwise it sets them to all zeros.

C.6.5 _mm_cmpgt_pi16

Syntax: `__m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)`

Description:
If the respective 16-bit values in `m1` are greater than the respective 16-bit values in `m2`, the function sets the respective 16-bit resulting values to all ones, otherwise it sets them to all zeros.

C.6.6 _mm_cmpgt_pi32

Syntax: `__m64 _mm_cmpgt_pi32 (__m64 m1, __m64 m2)`

Description:
If the respective 32-bit values in `m1` are greater than the respective 32-bit values in `m2`, the function sets the respective 32-bit resulting values to all ones, otherwise it sets them all to zeros.

C.6.7 _mm_cmpgt_pu8

Syntax: `__m64 _mm_cmpgt_pu8 (__m64 m1, __m64 m2)`

Description:
If the respective 8-bit values in `m1` are unsigned greater than the respective 8-bit values in `m2`, the function sets the respective 8-bit resulting values to all ones, otherwise it sets them to all zeros.

C.6.8 _mm_cmpgt_pu16

Syntax: `__m64 _mm_cmpgt_pu16 (__m64 m1, __m64 m2)`

Description:
If the respective 16-bit values in `m1` are unsigned greater than the respective 16-bit values in `m2`, the function sets the respective 16-bit resulting values to all ones, otherwise it sets them to all zeros.

C.6.9 _mm_cmpgt_pu32

Syntax: `__m64 _mm_cmpgt_pu32 (__m64 m1, __m64 m2)`

Description:
If the respective 32-bit values in `m1` are unsigned greater than the respective 32-bit values in `m2`, the function sets the respective 32-bit resulting values to all ones, otherwise it sets them all to zeros.

C.7 Wireless MMX™ Technology PACK/UNPACK Intrinsics

Table C-5. Overview of Wireless MMX™ Technology PACK/UNPACK Intrinsics (Sheet 1 of 2)

Intrinsic Name	Wireless MMX™ Instruction	MMX™ technology/ SSE Instruction	Operation	Signed	Saturation
_mm_packs_pi16	WPACKHSS	PACKSSWB	Pack	Yes	Yes
_mm_packs_pi32	WPACKWSS	PACKSSDW	Pack	Yes	Yes
_mm_packs_pu16	WPACKHUS	PACKUSWB	Pack	No	Yes
_mm_unpackhi_pi8	WUNPCKIHB	PUNPCKHBW	Interleave	--	--

Table C-5. Overview of Wireless MMX™ Technology PACK/UNPACK Intrinsics (Sheet 2 of 2)

Intrinsic Name	Wireless MMX™ Instruction	MMX™ technology/ SSE Instruction	Operation	Signed	Saturation
_mm_unpackhi_pi16	WUNPCKIHH	PUNPCKHWD	Interleave	--	--
_mm_unpackhi_pi32	WUNPCKIHW	PUNPCKHDQ	Interleave	--	--
_mm_unpacklo_pi8	WUNPCKILB	PUNPCKLBW	Interleave	--	--
_mm_unpacklo_pi16	WUNPCKILH	PUNPCKLWD	Interleave	--	--
_mm_unpacklo_pi32	WUNPCKILW	PUNPCKLDQ	Interleave	--	--
_mm_packs_si64	WPACKDSS	--	Pack	Yes	Yes
_mm_packs_su64	WPACKDUS	--	Pack	No	Yes
_mm_packs_pu32	WPACKWUS	--	Pack	No	Yes
_mm_unpackeh_pi8	WUNPCKEHSB	--	Signed extended	Yes	No
_mm_unpackeh_pi16	WUNPCKEHSH	--	Signed extended	Yes	No
_mm_unpackeh_pi32	WUNPCKEHSW	--	Signed extended	Yes	No
_mm_unpackeh_pu8	WUNPCKEHUB	--	Unsigned extended	No	No
_mm_unpackeh_pu16	WUNPCKEHUH	--	Unsigned extended	No	No
_mm_unpackeh_pu32	WUNPCKEHUW	--	Unsigned extended	No	No
_mm_unpackel_pi8	WUNPCKELSB	--	Signed extended	Yes	No
_mm_unpackel_pi16	WUNPCKELSH	--	Signed extended	Yes	No
_mm_unpackel_pi32	WUNPCKELSW	--	Signed extended	Yes	No
_mm_unpackel_pu8	WUNPCKELUB	--	Unsigned extended	No	No
_mm_unpackel_pu16	WUNPCKELUH	--	Unsigned extended	No	No
_mm_unpackel_pu32	WUNPCKELUW	--	Unsigned extended	No	No

C.7.1 _mm_packs_pi16

Syntax: `__m64 _mm_packs_pi16 (__m64 m1, __m64 m2)`

Description:
Packs the four 16-bit values from `m1` into the lower four 8-bit values of the result with signed saturation, and packs the four 16-bit values from `m2` into the upper four 8-bit values of the result with signed saturation.

C.7.2 _mm_packs_pi32

Syntax: `__m64 _mm_packs_pi32 (__m64 m1, __m64 m2)`

Description:
Packs the two 32-bit values from `m1` into the lower two 16-bit values of the result with signed saturation, and packs the two 32-bit values from `m2` into the upper two 16-bit values of the result with signed saturation.

C.7.3 _mm_packs_pu16

Syntax: `__m64 _mm_packs_pu16 (__m64 m1, __m64 m2)`

Description:
Packs the four 16-bit values from `m1` into the lower four 8-bit values of the result with unsigned saturation, and packs the four 16-bit values from `m2` into the upper four 8-bit values of the result with unsigned saturation.

C.7.4 _mm_unpackhi_pi8

Syntax: `__m64 _mm_unpackhi_pi8 (__m64 m1, __m64 m2)`

Description:
Interleaves the four 8-bit values from the high half of `m1` with the four values from the high half of `m2`. The interleaving begins with the data from `m1`.

C.7.5 _mm_unpackhi_pi16

Syntax: `__m64 _mm_unpackhi_pi16 (__m64 m1, __m64 m2)`

Description:
Interleaves the two 16-bit values from the high half of `m1` with the two values from the high half of `m2`. The interleaving begins with the data from `m1`.

C.7.6 _mm_unpackhi_pi32

Syntax: `__m64 _mm_unpackhi_pi32 (__m64 m1, __m64 m2)`

Description:
Interleaves the 32-bit value from the high half of `m1` with the 32-bit value from the high half of `m2`. The interleaving begins with the data from `m1`.

C.7.7 _mm_unpacklo_pi8

Syntax: `__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)`

Description:
Interleaves the four 8-bit values from the low half of `m1` with the four values from the low half of `m2`. The interleaving begins with the data from `m1`.

C.7.8 _mm_unpacklo_pi16

Syntax: `__m64 _mm_unpacklo_pi16 (__m64 m1, __m64 m2)`

Description:
Interleaves the two 16-bit values from the low half of `m1` with the two values from the low half of `m2`. The interleaving begins with the data from `m1`.

C.7.9 _mm_unpacklo_pi32

Syntax: `__m64 _mm_unpacklo_pi32 (__m64 m1, __m64 m2)`

Description:
Interleaves the 32-bit value from the low half of `m1` with the 32-bit value from the low half of `m2`. The interleaving begins with the data from `m1`.

C.7.10 _mm_packs_si64

Syntax: `__m64 _mm_packs_si64 (__m64 m1, __m64 m2)`

Description:
Packs the 64-bit value from `m1` into the lower 32-bit value of the result with signed saturation, and packs the one 32-bit value from `m2` into the upper 32-bit value of the result with signed saturation.

C.7.11 _mm_packs_su64

Syntax: `__m64 _mm_packs_su64 (__m64 m1, __m64 m2)`

Description:
Packs the 64-bit values from `m1` into the lower 32-bit values of the result with signed saturation, and packs the upper 32-bit values from `m2` into the upper 32-bit values of the result with signed saturation.

C.7.12 _mm_packs_pu32

Syntax: `__m64 _mm_packs_pu32 (__m64 m1, __m64 m2)`

Description:
Packs the two 32-bit values from `m1` into the lower two 16-bit values of the result with unsigned saturation, and packs the two 32-bit values from `m2` into the upper two 16-bit values of the result with unsigned saturation.

C.7.13 _mm_unpackeh_pi8

Syntax: `__m64 _mm_unpackeh_pi8 (__m64 m1)`

Description:
Unpacks the four 8-bit values from the high half of `m1` and sign extended each value of result.

C.7.14 _mm_unpackeh_pi16

Syntax: `__m64 _mm_unpackeh_pi16 (__m64 m1)`

Description:
Unpacks the two 16-bit values from the high half of `m1` and sign extended each value of result.

C.7.15 _mm_unpackeh_pi32

Syntax: `__m64 _mm_unpackeh_pi32 (__m64 m1)`

Description:
Unpacks the 32-bit value from the high half of `m1` and sign extended each value of result.

C.7.16 _mm_unpackeh_pu8

Syntax: `__m64 _mm_unpackeh_pu8 (__m64 m1)`

Description:
Unpacks the four 8-bit values from the high half of `m1` and zero extended each value of result.

C.7.17 _mm_unpackeh_pu16

Syntax: `__m64 _mm_unpackeh_pu16 (__m64 m1)`

Description:
Unpacks the two 16-bit values from the high half of `m1` and zero extended each value of result.

C.7.18 _mm_unpackeh_pu32

Syntax: `__m64 _mm_unpackeh_pu32 (__m64 m1)`

Description:
Unpacks the 32-bit value from the high half of `m1` and zero extended each value of result.

C.7.19 _mm_unpackel_pi8

Syntax: `__m64 _mm_unpackel_pi8 (__m64 m1)`

Description:
Unpacks the four 8-bit values from the low half of `m1` and sign extended each value of result.

C.7.20 _mm_unpackel_pi16

Syntax: `__m64 _mm_unpackel_pi16 (__m64 m1)`

Description:
Unpacks the two 16-bit values from the low half of `m1` and sign extended each value of result.

C.7.21 _mm_unpackel_pi32

Syntax: `__m64 _mm_unpackel_pi32 (__m64 m1)`

Description:
Unpacks the 32-bit value from the low half of `m1` and sign extended each value of result.

C.7.22 _mm_unpackel_pu8

Syntax: `__m64 _mm_unpackel_pu8 (__m64 m1)`

Description:
Unpacks the four 8-bit values from the low half of `m1` and zero extended each value of result.

C.7.23 _mm_unpackel_pu16

Syntax: `__m64 _mm_unpackel_pu16 (__m64 m1)`

Description:
Unpacks the two 16-bit values from the low half of `m1` and zero extended each value of result.

C.7.24 _mm_unpackel_pu32

Syntax: `__m64 _mm_unpackel_pu32 (__m64 m1)`

Description:
Unpacks the 32-bit value from the low half of `m1` and zero extended each value of result.

C.8 Wireless MMX™ Technology Set Intrinsics

Table C-6. Overview of Wireless MMX™ Technology Set Intrinsics

Intrinsic Name	Operation	Number of Elements	Instruction	Element Bit Size	Signed	Reverse Order
_mm_setzero_si64	Set to zero	1	WZERO	64	No	No
_mm_set_pi32	Set integer values	2	Composite	32	No	No
_mm_set_pi16	Set integer values	4	Composite	16	No	No
_mm_set_pi8	Set integer values	8	Composite	8	No	No
_mm_set1_pi32	Set integer values	2	TBCSTW	32	Yes	No
_mm_set1_pi16	Set integer values	4	TBCSTH	16	Yes	No
_mm_set1_pi8	Set integer values	8	TBCSTB	8	Yes	No
_mm_setr_pi32	Set integer values	2	Composite	32	No	Yes
_mm_setr_pi16	Set integer values	4	Composite	16	No	Yes
_mm_setr_pi8	Set integer values	8	Composite	8	No	Yes
_mm_setwcx	Set control register	--	TMCR	--	--	--
_mm_getwcx	Get control register	--	TMRC	--	--	--

Note: In the following descriptions regarding the bits of the Wireleass MMX™ technology register, bit 0 is the least significant and bit 63 is the most significant.

C.8.1 _mm_setzero_si64

Syntax: `__m64 _mm_setzero_si64 ()`

Description:
Sets the 64-bit value to zero.

If `r = _mm_setzero_si64 ()`, the action will be

```
r : = 0x0
```

C.8.2 _mm_set_pi32

Syntax: `__m64 _mm_set_pi32 (int i1, int i0)`

Description:
Sets the 2 signed 32-bit integer values.

If `r = _mm_set_pi32(i1, i0)`, the action will be

```
r0 : = i0;
r1 : = i1;
```

C.8.3 _mm_set_pi16

Syntax: `__m64 _mm_set_pi16 (short w3, short w2, short w1, short w0)`

Description:
Sets the 4 signed 16-bit integer values.

If `r = _mm_set_pi16 (w3, w2, w1, w0)`, the action will be

```
r0 := w0;
r1 := w1;
r2 := w2;
r3 := w3;
```

C.8.4 _mm_set_pi8

Syntax:
```
__m64 _mm_set_pi8 (char b7, char b6,
                   char b5, char b4,
                   char b3, char b2,
                   char b1, char b0)
```

Description:
Sets the 8 signed 8-bit integer values.

If `r = _mm_set_pi8 (b7, b6, b5, b4, b3, b2, b1, b0)`, the action will be

```
r0 := b0;
r1 := b1;
...
r7 := b7;
```

C.8.5 _mm_set1_pi32

Syntax: `__m64 _mm_set1_pi32 (int i)`

Description:
Sets the 2 signed 32-bit integer values to `i`.

If `r = _mm_set1_pi32 (i)`, the action will be

```
r0 := i;
r1 := i;
```

C.8.6 _mm_set1_pi16

Syntax: `__m64 _mm_set1_pi16 (short w)`

Description:
Sets the 4 signed 16-bit integer values to `w`.

If `r = _mm_set1_pi16 (w)`, action will be

```
r0 := w;
r1 := w;
r2 := w;
r3 := w;
```

C.8.7 _mm_set1_pi8

Syntax: `__m64 _mm_set1_pi8 (char b)`

Description:
Sets the 8 signed 8-bit integer values to `b`.

If `r = _mm_set1_pi8 (b)`, the action will be

```
r0 := b;
r1 := b;
...
r7 := b;
```

C.8.8 _mm_setr_pi32

Syntax: `__m64 _mm_setr_pi32 (int i0, int i1)`

Description:
Sets the 2 signed 32-bit integer values in reverse order.

If `r = _mm_setr_pi32 (i0, i1)`, the action will be

```
r0 := i0;
r1 := i1;
```

C.8.9 _mm_setr_pi16

Syntax: `__m64 _mm_setr_pi16 (short w0, short w1,short w2, short w3)`

Description:
Sets the 4 signed 16-bit integer values in reverse order.

If `r = _mm_setr_pi16 (w0, w1, w2, w3)`, the action will be

```
r0 := w0;
r1 := w1;
r2 := w2;
r3 := w3;
```

C.8.10 _mm_setr_pi8

Syntax: `__m64 _mm_setr_pi8 (char b0, char b1,char b2, char b3,`
`char b4, char b5,`
`char b6, char b7)`

Description:
Sets the 8 signed 8-bit integer values in reverse order.

If `r = _mm_setr_pi8 (b0, b1, b2, b3, b4, b5, b6, b7)`, the action will be

```
r0 := b0;
r1 := b1;
...
r7 := b7;
```

C.8.11 _mm_setwcx

Syntax: `void __mm_setwcx(int value, int number)`

Description:
Sets specified Wireless MMX™ technology control register with the contents of `value`. Where `number` is the coprocessor register number.

C.8.12 _mm_getwcx

Syntax: `int __mm_getwcx(int number)`

Description:
Returns contents of Wireless MMX™ technology control register, which is specified with `number`. Where `number` is the coprocessor register number.

C.9 Wireless MMX™ Technology General Support Intrinsics

Table C-7. Overview of Wireless MMX™ Technology General Support Intrinsics (Sheet 1 of 2)

Intrinsic Name	Operation	Wireless MMX™ Instruction	MMX™ technology/ SSE Instruction	Sign
_mm_extract_pi8	Extract on of eight bytes	TEXTRMSB	--	Yes
_mm_extract_pi16	Extract on of four half words	TEXTRMSH	PEXTRW	Yes
_mm_extract_pi32	Extract on of two words	TEXTRMSW	--	Yes
_mm_extract_pu8	Extract on of eight bytes	TEXTRMUB	--	No
_mm_extract_pu16	Extract on of four half words	TEXTRMUH	--	No
_mm_extract_pu32	Extract on of two words	TEXTRMUW	--	No
_mm_insert_pi8	Insert a byte	TINSRB	--	--
_mm_insert_pi16	Insert a half word	TINSRH	PINSRW	--
_mm_insert_pi32	Insert a word	TINSRW	--	--
_mm_max_pi8	Compute the maximum	WMAXSB	--	
_mm_max_pi16	Compute the maximum	WMAXSH	PMAXSW	
_mm_max_pi32	Compute the maximum	WMAXSW	--	
_mm_max_pu8	Compute the maximum, unsigned	WMAXUB	PMAXUB	
_mm_max_pu16	Compute the maximum, Unsigned	WMAXUH	--	
_mm_max_pu32	Compute the maximum, unsigned	WMAXUW	--	
_mm_min_pi8	Compute the minimum	WMINSB	--	
_mm_min_pi16	Compute the minimum	WMINSH	PMINSW	
_mm_min_pi32	Compute the minimum	WMINSW	--	
_mm_min_pu8	Compute the minimum, unsigned	WMINUB	PMINUB	
_mm_min_pu16	Compute the minimum, unsigned	WMINUH	--	
_mm_min_pu32	Compute the minimum, unsigned	WMINUW	--	
_mm_movemask_pi8	Create an eight-bit mask	TMOVMSKB	PMOVMSKB	
_mm_movemask_pi16	Create an four half word mask	TMOVMSKH	--	
_mm_movemask_pi32	Create an two word mask	TMOVMSKW	--	
_mm_shuffle_pi16	Return a combination of four words	WSHUFH	PSHUFW	
_mm_avg_pu8	Compute rounded average	WAVG2BR	PAVGB	
_mm_avg_pu16	Compute rounded average	WAVG2HR	PAVGW	

Table C-7. Overview of Wireless MMX™ Technology General Support Intrinsics (Sheet 2 of 2)

Intrinsic Name	Operation	Wireless MMX™ Instruction	MMX™ technology/ SSE Instruction	Sign
_mm_avg2_pu8	Compute rounded average without rounding +1	WAVG2B	--	
_mm_avg2_pu16	Compute rounded average without rounding +1	WAVG2H	--	
_mm_sad_pu8	Compute sum of absolute differences	WSADB	PSADBW	
_mm_sad_pu16	Compute sum of absolute differences	WSADH	--	
_mm_sadz_pu8	Accumulate sum of absolute differences	WSADBZ		
_mm_sadz_pu16	Accumulate sum of absolute differences	WSADHZ	--	
_mm_align_si64	Extract a 64-bit value	WALIGNI/ WALIGNR	--	--
_mm_cvtsi64_m64	Transfer	TMCRR	--	--
_mm_cvtm64_si64	Transfer	TMRRC	--	--

C.9.1 _mm_extract_pi8

Syntax: `int _mm_extract_pi8(__m64 a, const int n)`

Description:
Extracts one of the eight bytes of `a`. The selector `n` must be an immediate.

If `r = _mm_extract_pi8(a, n)`, the action will be

```
r[7:0] = a[Byte n[2:0]];
r[31:8] = SingReplicate(a[Byte n[2:0]], 24);
```

C.9.2 _mm_extract_pi16

Syntax: `int _mm_extract_pi16(__m64 a, const int n)`

Description:
Extracts one of the four half words of `a`. The selector `n` must be an immediate.

If `r = _mm_extract_pi16(a, n)`, the action will be

```
r[15:0] = a[Halfword n[1:0]];
r[31:16] = SingReplicate(a[Byte n[1:0]], 16);
```

C.9.3 _mm_extract_pi32

Syntax: `int _mm_extract_pi32(__m64 a, const int n)`

Description:
Extracts one of the two words of a. The selector n must be an immediate.

If `r = _mm_extract_pi32(a, n)`, the action will be

```
r[31:0] = a[Byte n[0]];
```

C.9.4 _mm_extract_pu8

Syntax: `int _mm_extract_pu8(__m64 a, const int n)`

Description:
Extracts one of the eight bytes of a. The selector n must be an immediate.

If `r = _mm_extract_pu8(a, n)`, the action will be

```
r[7:0] = a[Byte n[2:0]];
r[31:8] = 0;
```

C.9.5 _mm_extract_pu16

Syntax: `int _mm_extract_pu16(__m64 a, const int n)`

Description:
Extracts one of the four half words of a. The selector n must be an immediate.

If `r = _mm_extract_pu16(a, n)`, the action will be

```
r [15:0] = a[Halfword n[1:0]];
r[31:16] = 0;
```

C.9.6 _mm_extract_pu32

Syntax: `int _mm_extract_pu32(__m64 a, const int n)`

Description:
This provides same functionality as `_mm_extract_pi32`.

C.9.7 _mm_insert_pi8

Syntax: `__m64 _mm_insert_pi8(__m64 a, int d, int n)`

Description:
Inserts byte d into one of eight bytes of a. The selector n must be an immediate.

If `r = _mm_insert_pi8(a, d, n)`, the action will be

```
r0 := (n==0) ? d[7:0] : a0;
r1 := (n==1) ? d[7:0] : a1;
r2 := (n==2) ? d[7:0] : a2;
r3 := (n==3) ? d[7:0] : a3;
r4 := (n==4) ? d[7:0] : a4;
r5 := (n==5) ? d[7:0] : a5;
r6 := (n==6) ? d[7:0] : a6;
r7 := (n==7) ? d[7:0] : a7;
```

C.9.8 _mm_insert_pi16

Syntax: `__m64 _mm_insert_pi16(__m64 a, int d, int n)`

Description:
Inserts half word d into one of four half words of a. The selector n must be an immediate.

If `r = _mm_insert_pi16(a, d, n)`, the action will be

```
r0 := (n==0) ? d[15:0] : a0;
r1 := (n==1) ? d[15:0] : a1;
r2 := (n==2) ? d[15:0] : a2;
r3 := (n==3) ? d[15:0] : a3;
```

C.9.9 _mm_insert_pi32

Syntax: `__m64 _mm_insert_pi32(__m64 a, int d, int n)`

Description:
Inserts word d into one of two half words of a. The selector n must be an immediate.

If `r = _mm_insert_pi32(a, d, n)`, the action will be

```
r0 := (n==0) ? d[31:0] : a0;
r1 := (n==1) ? d[31:0] : a1;
```

C.9.10 _mm_max_pi8

Syntax: `__m64 _mm_max_pi8(__m64 a, __m64 b)`

Description:
Computes the element-wise maximum of the bytes in a and b.

If `r = _mm_max_pi8(a, b)`, the action will be

```
r0 := max(a0, b0);
r1 := max(a1, b1);
...
r7 := max(a7, b7);
```

C.9.11 _mm_max_pi16

Syntax: `__m64 _mm_max_pi16(__m64 a, __m64 b)`

Description:
Computes the element-wise maximum of the half words in `a` and `b`.

If `r = _mm_max_pi16(a, b)`, the action will be

```
r0 := max(a0, b0);
r1 := max(a1, b1);
r2 := max(a2, b2);
r3 := max(a3, b3);
```

C.9.12 _mm_max_pi32

Syntax: `__m64 _mm_max_pi32(__m64 a, __m64 b)`

Description:
Computes the element-wise maximum of the words in `a` and `b`.

If `r = _mm_max_pi32(a, b)`, the action will be

```
r0 := max(a0, b0);
r1 := max(a1, b1);
```

C.9.13 _mm_max_pu8

Syntax: `__m64 _mm_max_pu8(__m64 a, __m64 b)`

Description:
Computes the element-wise maximum of the unsigned bytes in `a` and `b`.

If `r = _mm_max_pu8(a, b)`, the action will be

```
r0 := max(a0, b0);
r1 := max(a1, b1);
...
r7 := max(a7, b7);
```

C.9.14 _mm_max_pu16

Syntax: `__m64 _mm_max_pu16(__m64 a, __m64 b)`

Description:
Computes the element-wise maximum of the unsigned half words in `a` and `b`.

If `r = _mm_max_pu16(a, b)`, the action will be

```
r0 := max(a0, b0);
r1 := max(a1, b1);
r2 := max(a2, b2);
r3 := max(a3, b3);
```

C.9.15 _mm_max_pu32

Syntax: `__m64 _mm_max_pu32(__m64 a, __m64 b)`

Description:
Computes the element-wise maximum of the unsigned words in a and b.

If `r = _mm_max_pu32(a, b)`, the action will be

```
r0 := max(a0, b0);
r1 := max(a1, b1);
```

C.9.16 _mm_min_pi8

Syntax: `__m64 _mm_min_pi8(__m64 a, __m64 b)`

Description:
Computes the element-wise minimum of the bytes in a and b.

If `r = _mm_min_pi8(a, b)`, the action will be

```
r0 := min(a0, b0);
r1 := min(a1, b1);
...
r7 := min(a7, b7);
```

C.9.17 _mm_min_pi16

Syntax: `__m64 _mm_min_pi16(__m64 a, __m64 b)`

Description:
Computes the element-wise minimum of the half words in a and b.

If `r = _mm_min_pi16(a, b)`, the action will be

```
r0 := min(a0, b0);
r1 := min(a1, b1);
r2 := min(a2, b2);
r3 := min(a3, b3);
```

C.9.18 _mm_min_pi32

Syntax: `__m64 _mm_min_pi32(__m64 a, __m64 b)`

Description:
Computes the element-wise minimum of the words in a and b.

If `r = _mm_min_pi32(a, b)`, the action will be

```
r0 := min(a0, b0);
r1 := min(a1, b1);
```

C.9.19 _mm_min_pu8

Syntax: `__m64 _mm_min_pu8(__m64 a, __m64 b)`

Description:
Computes the element-wise minimum of the unsigned bytes in a and b.

If `r = _mm_min_pu8(a, b)`, the action will be

```
r0 := min(a0, b0);
r1 := min(a1, b1);
...
r7 := min(a7, b7);
```

C.9.20 _mm_min_pu16

Syntax: `__m64 _mm_min_pu16(__m64 a, __m64 b)`

Description:
Computes the element-wise minimum of the unsigned half words in a and b.

If `r = _mm_min_pu16(a, b)`, the action will be

```
r0 := min(a0, b0);
r1 := min(a1, b1);
r2 := min(a2, b2);
r3 := min(a3, b3);
```

C.9.21 _mm_min_pu32

Syntax: `__m64 _mm_min_pu32(__m64 a, __m64 b)`

Description:
Computes the element-wise minimum of the unsigned words in a and b.

If `r = _mm_min_pu32(a, b)`, the action will be

```
r0 := min(a0, b0);
r1 := min(a1, b1);
```

C.9.22 _mm_movemask_pi8

Syntax: `int _mm_movemask_pi8(__m64 a)`

Description:
Creates an 8-bit mask from the most significant bits of the bytes in `a`.

If `r = _mm_movemask_pi8(a)`, the action will be

```
r = 0;
r := sign(a7)<<7 | sign(a6)<<6 |... | sign(a0);
```

C.9.23 _mm_movemask_pi16

Syntax: `int _mm_movemask_pi16(__m64 a)`

Description:
Creates an 4-bit mask from the most significant bits of the half words in `a`.

If `r = _mm_movemask_pi16(a)`, the action will be

```
r = 0;
r := sign(a3)<<3 | sign(a2)<<2 |... | sign(a0);
```

C.9.24 _mm_movemask_pi32

Syntax: `int _mm_movemask_pi32(__m64 a)`

Description:
Creates an 2-bit mask from the most significant bits of the bytes in `a`.

If `r = _mm_movemask_pi32(a)`, the action will be

```
r = 0;
r := sign(a1)<<1 | sign(a0);
```

C.9.25 _mm_shuffle_pi16

Syntax: `__m64 _mm_shuffle_pi16(__m64 a, int n)`

Description:
Returns a combination of the four half words of a. The selector n must be an immediate.

If `r = _mm_shuffle_pi16(a, n)`, the action will be

```
r0 := Half word (n&0x3) of a
r1 := Half word ((n>>2)&0x3) of a
r2 := Half word ((n>>4)&0x3) of a
r3 := Half word ((n>>6)&0x3) of a
```

C.9.26 _mm_avg_pu8

Syntax: `__m64 _mm_avg_pu8(__m64 a, __m64 b)`

Description:
Computes the (rounded) averages of the unsigned bytes in a and b.

If `r = _mm_avg_pu8(a, b)`, the action will be

```
t = (unsigned short)a0 + (unsigned short)b0;
r0 = (t >> 1) | (t & 0x01);
...
t = (unsigned short)a7 + (unsigned short)b7;
r7 = (unsigned char)((t >> 1) | (t & 0x01));
```

C.9.27 _mm_avg_pu16

Syntax: `__m64 _mm_avg_pu16(__m64 a, __m64 b)`

Description:
Computes the (rounded) averages of the unsigned words in a and b.

If `r = _mm_avg_pu16(a, b)`, the action will be

```
t = (unsigned int)a0 + (unsigned int)b0;
r0 = (t >> 1) | (t & 0x01);
...
t = (unsigned word)a7 + (unsigned word)b7;
r7 = (unsigned short)((t >> 1) | (t & 0x01));
```

C.9.28 _mm_avg2_pu8

Syntax: `__m64 _mm_avg2_pu8(__m64 a, __m64 b)`

Description:
Computes the without rounded averages of the unsigned bytes in a and b.

If `r = _mm_avg2_pu8(a, b)`, the action will be

```
t = (unsigned byte)a0 + (unsigned byte)b0;
r0 = (t >> 1);
...
t = (unsigned byte)a7 + (unsigned byte)b7;
r7 = (unsigned char)((t >> 1);
```

C.9.29 _mm_avg2_pu16

Syntax: `__m64 _mm_avg2_pu16(__m64 a, __m64 b)`

Description:
Computes the with out rounded averages of the unsigned words in `a` and `b`.

If `r = _mm_avg2_pu16(a, b)`, the action will be

```
t = (unsigned half word)a0 + (unsigned half word)b0;
r0 = (t >> 1);
...
t = (unsigned half word)a3 + (unsigned half word)b3;
r3 = (unsigned short)(t >> 1);
```

C.9.30 _mm_sadz_pu8

Syntax: `__m64 _mm_sadz_pu8(__m64 a, __m64 b)`

Description:
Computes the sum of the absolute differences of the unsigned bytes in `a` and `b`, returning he value in the lower word. The upper three words are cleared.

If `r = _mm_sadz_pu8(a, b)`, the action will be

```
r0 = abs(a0-b0) +... + abs(a7-b7);
r1 = r2 = … = 0;
```

C.9.31 _mm_sadz_pu16

Syntax: `__m64 _mm_sadz_pu16(__m64 a, __m64 b)`

Description:
Computes the sum of the absolute differences of the unsigned half words in `a` and `b`, returning he value in the lower word. The upper three words are cleared.

If `r = _mm_sadz_pu16(a, b)`, the action will be

```
r0 = abs(a0-b0) +... + abs(a3-b3);
r1 = r2 … = 0;
```

C.9.32 _mm_sad_pu8

Syntax: `__m64 _mm_sad_pu8(__m64 a, __m64 b)`

Description:
Computes the sum of the absolute differences of the unsigned bytes in `a` and `b`, returning he value in the lower word. The upper three words are cleared.

If `r = _mm_sad_pu8(a, b)`, the action will be

```
r0 = abs(a0-b0) +... + abs(a7-b7) + r0;
r1 = r2 = … = 0;
```

C.9.33 _mm_sad_pu16

Syntax: `__m64 _mm_sad_pu16(__m64 a, __m64 b)`

Description:
Computes the sum of the absolute differences of the unsigned half words in `a` and `b`, returning the value in the lower word. The upper three words are cleared.

If `r = _mm_sad_pu16(a, b)`, the action will be

```
r0 = abs(a0-b0) +... + abs(a3-b3) + r0;
r1 = r2 … = 0;
```

C.9.34 __mm_align_si64

Syntax: `__m64 __mm_align_si64(__m64 m1, __m64 m2, int count)`

Description:
Extracts an 64-bit value from the two 64-bit input values `m1`, `m2` with `count` byte offset.

If `r = __mm_align_si64(m1,  m2, count)`, the action will be

```
r = Low_DB_word((m1, m2) >> (count * 8));
```

C.9.35 _mm_cvtsi64_m64

Syntax: `__m64 _mm_cvtsi64_m64 (__int64 i)`

Description:
Converts the `__int64` integer `i` to a 64-bit `__m64` object.

If `r = _cvtsi64_m64(a)`, action will be

```
r[31:0] = i0 (low word)
r[63:32] = i1 (high word)
```

C.9.36 _mm_cvtm64_si64

Syntax: `__int64 _mm_cvtm64_si64 (__m64 m)`

Description:
Converts the 64-bit `__m64` object `m` to `__int64` bit integer.

If `r = _cvtm64_si64(a)`, action will be

```
r0 = m[31:0]  (low word)
r1 = m[63:32] (high word)
```

Index