Intel XScale® Microarchitecture Programmer Model for Big Endian

Application Note

June 2001

Order Number: 278402-001

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

This document and the software described in it are furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document. Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's website at http://www.intel.com.

Copyright © Intel Corporation, 2001

Intel and Intel XScale are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Contents

1.0 Introduction 5

2.0 Endian Configuration 6

3.0 Data transfer 6

- 3.1 Word Accesses 7
  - 3.1.1 Word Load (LDR) 7
  - 3.1.2 Word Store (STR) 8
  - 3.1.3 Word Swap (SWP) 9
- 3.2 Half-Word Access 10
  - 3.2.1 Half Word Load (LDRH) 10
  - 3.2.2 Half Word Store (STRH) 11
- 3.3 Byte Access 12
  - 3.3.1 Byte Load (LDRB) 12
  - 3.3.2 Byte Store (STRB) 13
  - 3.3.3 Byte Swap (SWPB) 14
- 3.4 Endianness Test Code 14

Figures

1 Big Endian Byte, Half Word, and Word Memory Mapping 5

2 Little Endian Byte, Half Word, and Word Memory Mapping 6

1.0 Introduction

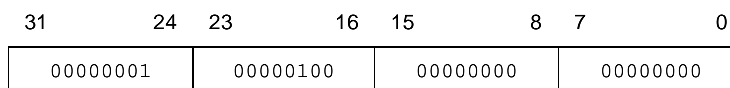
This document describes the configuration and behavior of the Intel XScale® Microarchitecture in big endian mode. The memory data transfer instruction sets being discussed are load, store, and swap in word, half word and byte with aligned and unaligned access. Section 3.4, “Endianness Test Code” on page 14 is an example code for system endianness testing.

Intel XScale® Microarchitecture has the option to operate in either little or big endian mode. This refers to the way how the memory is being accessed. Big endian byte ordering assigns the lowest byte address to the most significant byte of a 32-bit memory word, where the little endian byte order is the reverse. For an example, a decimal word of 1025 is represented as

00000000 00000000 00000100 00000001

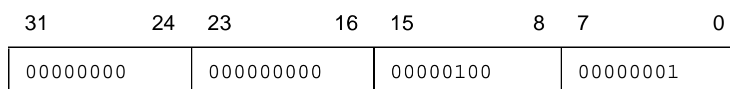
This is represented in the different modes:

- Big Endian



31 24	23 16	15 8	7 0
00000001	00000100	00000000	00000000

- Little Endian



31 24	23 16	15 8	7 0
00000000	000000000	00000100	00000001

Figures 1 and 2 show memory mapping for big and little endian modes respectivly. Both of these figures assume that address X is word aligned.

Figure 1. Big Endian Byte, Half Word, and Word Memory Mapping

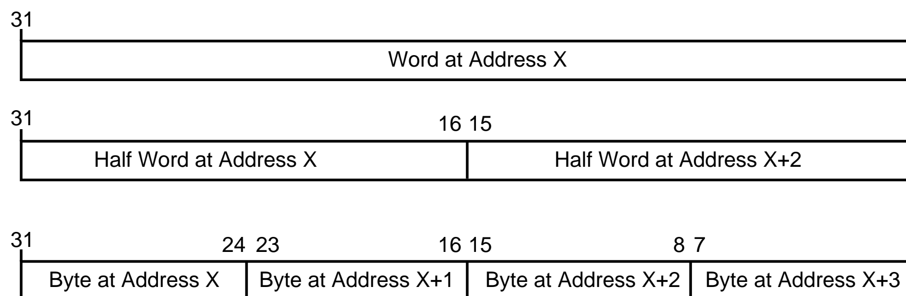
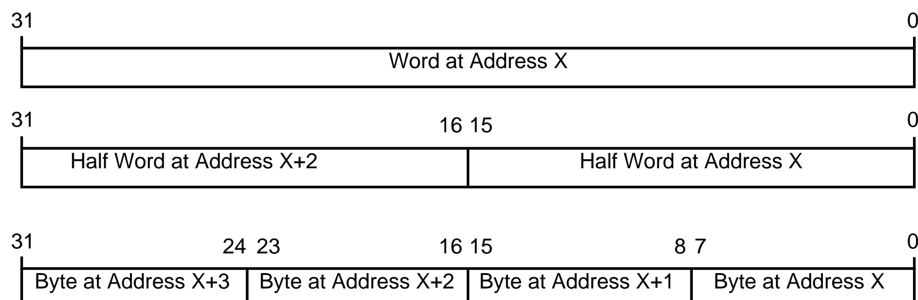


Figure 2. Little Endian Byte, Half Word, and Word Memory Mapping



2.0 Endian Configuration

The big and little endian modes are controlled by the B-bit of the Control Register (Coprocessor 15, register1, bit 7). The default mode at reset is little endian. To enable the big endian mode, the B bit must be set before performing any sub-word accesses to memory, or undefined results would occur. The bit takes effect even if the MMU is disabled. The following is assembly code to enable B-bit.

```
MACRO BIGENDIAN

MRC p15,0,a1,c1,c0,0

ORR a1,a1,#0x80 ;set bit7 of register1 cp15

MCR p15,0,a1,c1,c0,0

ENDM
```

The application code built to run on the system must be compiled to match the endianness. To produce the object code which is targeted for big endian system, the compiler must be specified to work in big endian mode. For example, *-mbig-endian* switch must be specified for GNU CC because the default is in little endian. For GNUPro assembler, *-EB* switch would assemble the code for big endian. The library being used must have been compiled in the big endian mode.

3.0 Data transfer

This section describes the behavior of the Big Endian Intel XScale® Microarchitecture in data transfer instructions.

3.1 Word Accesses

The endianness applies when byte or half-word accesses are made to memory. For a 32-bit word memory access, the bit pattern in the memory must match the bit pattern in the processor register, regardless if the system is big or little endian.

3.1.1 Word Load (LDR)

If the address is not word aligned, the loaded value is rotated right by 8 times the value of bits[1:0] of the address.

Before code execution:

Registers	Memory location	Byte 0	Byte1	Byte 2	Byte 3
• r4 = 0x0 • r5 = 0x0 • r6 = 0x0 • r7 = 0x0 • r10 = 0x0	0x0	AA	BB	CC	DD

Code segment:

```
LDR r4, [r10]           ; word load in aligned access
LDR r5, [r10, #1]       ; word load in unaligned access with 1 byte offset
LDR r6, [r10, #2]       ; word load in unaligned access with 2 byte offset
LDR r7, [r10, #3]       ; word load in unaligned access with 3 byte offset
```

After code execution:

Registers	Memory location	Byte 0	Byte1	Byte 2	Byte 3
• r4 = 0xAABBCCDD • r5 = 0xDDAABBCC • r6 = 0xCCDDAABB • r7 = 0xBBCCDDAA • r10 = 0x0	0x0	AA	BB	CC	DD

3.1.2 Word Store (STR)

In unaligned access, STR instruction ignores the least significant two bits of address.

Before code execution:

Registers	Memory location	Byte 0	Byte1	Byte 2	Byte 3
• r4 = 0xAABBCCDD • r10 = 0x0 • r11 = 0x4	0x0	00	00	00	00
	0x4	00	00	00	00

Code segment:

```
STR r4, [r10]             ;word store in aligned access

STR r4, [r11, #1]         ;word store in unaligned access
```

After code execution:

Registers	Memory location	Byte 0	Byte1	Byte 2	Byte 3
• r4 = 0xAABBCCDD • r10 = 0x0 • r11 = 0x4	0x0	AA	BB	CC	DD
	0x4	AA	BB	CC	DD

3.1.3 Word Swap (SWP)

A SWP instruction performs the load and store operation. If the address is not word aligned, the loaded value is rotated right by 8 times the bits [1:0] of the address. The stored value is not rotated.

Before code execution:

Registers	Memory location	Byte 0	Byte1	Byte 2	Byte 3
• r4 = 0x11223344 • r5 = 0x55667788 • r10 = 0x0 • r11 = 0x4	0x0	AA	BB	CC	DD
	0x4	EE	FF	00	99

Code segment:

```
SWP r4, r4, [r10]           ; word swap in aligned access

SWP r5, r5, [r11, #1]       ; word swap in unaligned access with 1 byte offset
```

After code execution:

Registers	Memory location	Byte 0	Byte1	Byte 2	Byte 3
• r4 = 0xAABBCCDD • r5 = 0x99EEFF00 • r10 = 0x0 • r11 = 0x4	0x0	11	22	33	44
	0x4	55	66	77	88

3.2 Half-Word Access

3.2.1 Half Word Load (LDRH)

The loaded half word is zero-extended to a 32-bit word for half word aligned memory address. For non half word aligned memory address, the loaded value is unpredictable.

Before code execution:

Registers	Memory location	Byte 0	Byte1	Byte 2	Byte 3
• r4 = 0x0 • r5 = 0x0 • r6 = 0x0 • r7 = 0x0 • r10 = 0x0	0x0	AA	BB	CC	DD

Code segment:

```
LDRH r4, [r10]              ; half word load in aligned access

LDRH r5, [r10, #1]          ; half word load in unaligned access

LDRH r6, [r10, #2]          ; half word load in aligned access

LDRH r7, [r10, #3]          ; half word load in unaligned access
```

After code execution:

Registers[a]	Memory location	Byte 0	Byte1	Byte 2	Byte 3
• r4 = 0x0000AABB • r5 = 0xXXXXXXXX • r6 = 0x0000CCDD • r7 = 0xXXXXXXXX • r10 = 0x0	0x0	AA	BB	CC	DD

a. XX means unpredictable

3.2.2 Half Word Store (STRH)

An unaligned half word memory access would cause the stored value to be unpredictable.

Before code execution:

Registers	Memory location	Byte 0	Byte1	Byte 2	Byte 3
• r4 = 0xAABBCCDD • r10 = 0x0 • r11 = 0x4 • r12 =0x8	0x0	00	00	00	00
	0x4	00	00	00	00
	0x8	00	00	00	00

Code segment:

```
STRH r4, [r10]              ; halfword store in aligned access
STRH r4, [r11, #1]          ; halfword store in unaligned access
STRH r4, [r12, #2]          ; halfword store in aligned access
```

After code execution:

Registers	Memory location	Byte 0	Byte1	Byte 2	Byte 3
• r4 = 0xAABBCCDD • r10 = 0x0 • r11 = 0x4 • r12 = 0x8	0x0	CC	DD	00	00
	0x4	XX	XX	XX	XX
	0x8	00	00	CC	DD

3.3 Byte Access

3.3.1 Byte Load (LDRB)

The selected byte is placed at the [7:0] of the register and [31:8] of the register is zero extended.

Before code execution:

Registers	Memory location	Byte 0	Byte1	Byte 2	Byte 3
• r4 =0x0 • r5 =0x0 • r6 = 0x0 • r7 =0x0 • r10 = 0x0	0x0	AA	BB	CC	DD

Code segment:

```
LDRB r4, [r10]          ; byte load in aligned access
LDRB r5, [r10, #1]      ; byte load in unaligned access with 1 byte offset
LDRB r6, [r10, #2]      ; byte load in unaligned access with 2 byte offset
LDRB r7, [r10, #3]      ; byte load in unaligned access with 3 byte offset
```

After code execution:

Registers	Memory location	Byte 0	Byte1	Byte 2	Byte 3
• r4 =0x000000AA • r5 = 0x000000BB • r6 = 0x000000CC • r7 = 0x000000DD • r10 = 0x0	0x0	AA	BB	CC	DD

3.3.2 Byte Store (STRB)

Before code execution:

<table>
<tr><th>Registers</th><th>Memory location</th><th>Byte 0</th><th>Byte1</th><th>Byte 2</th><th>Byte 3</th></tr>
<tr><td rowspan="5">r4 = 0xAABBCCDD
r8 = 0x0
r9 = 0x4
r10 = 0x8
r11= 0xA</td><td>0x0</td><td>00</td><td>00</td><td>00</td><td>00</td></tr>
<tr><td>0x4</td><td>00</td><td>00</td><td>00</td><td>00</td></tr>
<tr><td>0x8</td><td>00</td><td>00</td><td>00</td><td>00</td></tr>
<tr><td>0xA</td><td>00</td><td>00</td><td>00</td><td>00</td></tr>
<tr><td>0x0</td><td>00</td><td>00</td><td>00</td><td>00</td></tr>
</table>

Code segment:

```
STRB r4, [r8,#0]

STRB r4, [r9, #1]

STRB r4, [r10, #2]

STRB r4, [r11, #3]
```

After code execution

<table>
<tr><th>Registers</th><th>Memory location</th><th>Byte 0</th><th>Byte1</th><th>Byte 2</th><th>Byte 3</th></tr>
<tr><td rowspan="5">r4 = 0xAABBCCDD
r8 = 0x0
r9 = 0x4
r10 = 0x8
r11 = 0xA</td><td>0x0</td><td>DD</td><td>00</td><td>00</td><td>00</td></tr>
<tr><td>0x4</td><td>00</td><td>DD</td><td>00</td><td>00</td></tr>
<tr><td>0x8</td><td>00</td><td>00</td><td>DD</td><td>00</td></tr>
<tr><td>0xA</td><td>00</td><td>00</td><td>00</td><td>DD</td></tr>
<tr><td>0x0</td><td>DD</td><td>00</td><td>00</td><td>00</td></tr>
</table>

3.3.3 Byte Swap (SWPB)

A SWPB instruction works as the LDRB and STRB.

Before code execution:

Registers	Memory location	Byte 0	Byte1	Byte 2	Byte 3
• r4 = 0x11223344 • r10 = 0x0	0x0	AA	BB	CC	DD

Code segment:

```
SWPB r4, r4, [r10] ;byte swap
```

After code execution:

Registers	Memory location	Byte 0	Byte1	Byte 2	Byte 3
• r4 = 0x112233AA • r10 = 0x0	0x0	44	BB	CC	DD

3.4 Endianness Test Code

The following is simple C code to test the endianness of the system.

This code would place character AB or ABCD in the memory array. Pointer is used to read back the memory array. A big endian system would return the characters as the same byte order, where the little endian system would return the characters in reverse order.

```
/* This program store a value of Return value of 1 if the byte for int is neither 2
or 4 bytes, OR neither little or big endian*/

#include <stdio.h>

static int w[1];

static char * bytes;

int main(void)

{

 printf ("This is an Endianness test:\n");

 if (sizeof (int) == 2)

 {

 w[0] = 0x4142;

            bytes = (char *) w;

 if (strcmp(bytes, "AB") == 0)

printf ("big endian\n");
```

```
 else if (strcmp(bytes, "BA") == 0)
printf ("little endian\n");
 else
{
 printf ("Not big nor little endian\n");
 return 1;
}
 }
 else if (sizeof (int) == 4)
 {
 w[0] = 0x41424344;
        bytes = (char *) w;
 if (strcmp(bytes, "ABCD") == 0)
printf ("big endian\n");
 else if (strcmp(bytes, "DCBA") == 0)
printf ("little endian\n");
 else
{
 printf ("Not big nor little endian\n");
 return 1;
}
 }
 else
 {
 printf ("unexpected size of int\n");
 return 1;
 }
 return 0;
```