



This chapter describes Avalon® Memory-Mapped (Avalon-MM) interface bus byte ordering and provides recommendations for representing data in your system. Understanding byte ordering in both hardware and software is important when using intellectual property (IP) cores that interpret data differently.

 Altera recommends understanding the following documents before proceeding:

- The *System Interconnect Fabric for Memory-Mapped Interfaces* chapter of the *SOPC Builder User Guide*.
- The *Avalon Interface Specifications*.


Endianness

The term *endian* describes data byte ordering in both hardware and software. The two most common forms of data byte ordering are little endian and big endian. Little endian means that the least significant portion of a value is presented first and stored at the lowest address in memory. Big endian means the most significant portion of a value is presented first and stored at the lowest address in memory. For example, consider the value 0x1234. In little endian format, the 4 is the first digit presented or stored. In big endian format, the 1 is the first digit presented or stored.

 Endianness typically refers only to byte ordering. Bit ordering within each byte is a separate subject covered in “PowerPC Bus Byte Ordering” on page 11–9 and “ARM BE-32 Bus Byte Ordering” on page 11–10.

Hardware Endianness

Hardware developers can map the data bits of an interface in any order. There must be coordination and agreement among developers so that the data bits of an interface correctly map to address offsets for any master or slave interface connected to the interconnect. Consistent hardware endianness or *bus byte ordering* is especially important in systems that contain IP interfaces of varying data widths because the interconnect performs the data width conversion between the master and slave interfaces. The key to simplifying bus byte ordering is to be consistent system-wide when connecting IP cores to the interconnect. For example, if all but one IP core in your system use little endian bus byte ordering, modify the interface of the one big endian IP core to conform to the rest of the system.

 The way an IP core presents data to the interconnect is not dependent on the internal representation of the data within the core. IP cores can map the data interface to match the bus data byte ordering specification independent of the internal arithmetic byte ordering.

Software Endianness

Software endianness or *arithmetic byte ordering* is the internal representation of data within an IP core, software compiler, and peripheral drivers. Processors can treat byte offset 0 of a variable as the most or least significant byte of a multibyte word. For example, the value 0x0A0B0C0D, which spans multiple bytes, can be represented different ways. A little endian processor considers the byte 0x0D of the value to be located at the lowest byte offset in memory, whereas a big endian processor considers the byte 0x0A of the value to be located at the lowest byte offset in memory.

Example 11-1 shows a C code fragment that illustrates the difference between the little endian and big endian arithmetic byte ordering used by most processors.

Example 11-1. Reading Byte Offset 0 of a 32-Bit Word

```
long * long_ptr;
char byte_value;
*long_ptr = 0x0A0B0C0D; // 32-bit store to 'long_ptr'
byte_value = *((char *)long_ptr); // 8-bit read from 'long_ptr'
```

In **Example 11-1**, the processor writes the 32-bit value 0x0A0B0C0D to memory, then reads the first byte of the value back. A little endian processor such as the Nios II processor, which considers memory byte offset 0 to be the least significant byte of a word, stores byte 0x0D to byte offset 0 of pointer location `long_ptr`. A processor such as a PowerPC®, which considers memory byte offset 0 to be the most significant byte of a word, stores byte 0x0A to byte offset 0 of pointer location `long_ptr`. As a result, the variable `byte_value` is loaded with 0x0D if this code executes on a little endian Nios II processor and 0x0A if this code executes on a big endian PowerPC processor.



Arithmetic byte ordering is not dependent on the bus byte ordering used by the processor data master that accesses memory. However, word and halfword accesses sometimes require byte swapping in software to correctly interpret the data internally by the processor. For more information, refer to “[Arithmetic Byte Reordering](#)” on page 11-12 and “[System-Wide Arithmetic Byte Reordering in Software](#)” on page 11-15.

Avalon-MM Interface Ordering

To ensure correct data communication, the Avalon-MM interface specification requires that each master or slave port of all components in your system pass data in descending bit order with data bits 7 down to 0 representing byte offset 0. This bus byte ordering is a little endian ordering. Any IP core that you add to your system must comply with the Avalon-MM interface specification. This ordering ensures that when any master accesses a particular byte of any slave port, the same physical byte lanes are accessed using a consistent bit ordering.



For more information, refer to the [Avalon Interface Specifications](#).

The interconnect handles dynamic bus sizing for narrow to wide and wide to narrow transfers when the master and slave port widths connected together do not match. When a wide master accesses a narrow slave, the interconnect serializes the data by presenting the lower bytes to the slave first. When a narrow master accesses a wide slave, the interconnect performs byte lane realignment to ensure that the master accesses the appropriate byte lanes of the slave.



For more information, refer to the *System Interconnect Fabric for Memory-Mapped Interfaces* chapter of the *SOPC Builder User Guide*.

Dynamic Bus Sizing DMA Examples

A direct memory access (DMA) engine moves memory contents from a source location to a destination location. Because Qsys and SOPC Builder support dynamic bus sizing, the data widths of the source and destination memory in the examples do not need to match the width of the DMA engine. The DMA engine reads data from a source base address and sequentially increases the address until the last read completes. The DMA engine also writes data to a destination base address and sequentially increases the address until the last write completes.

Figure 11-2. 32-Bit to 64-Bit Memory DMA Transfer

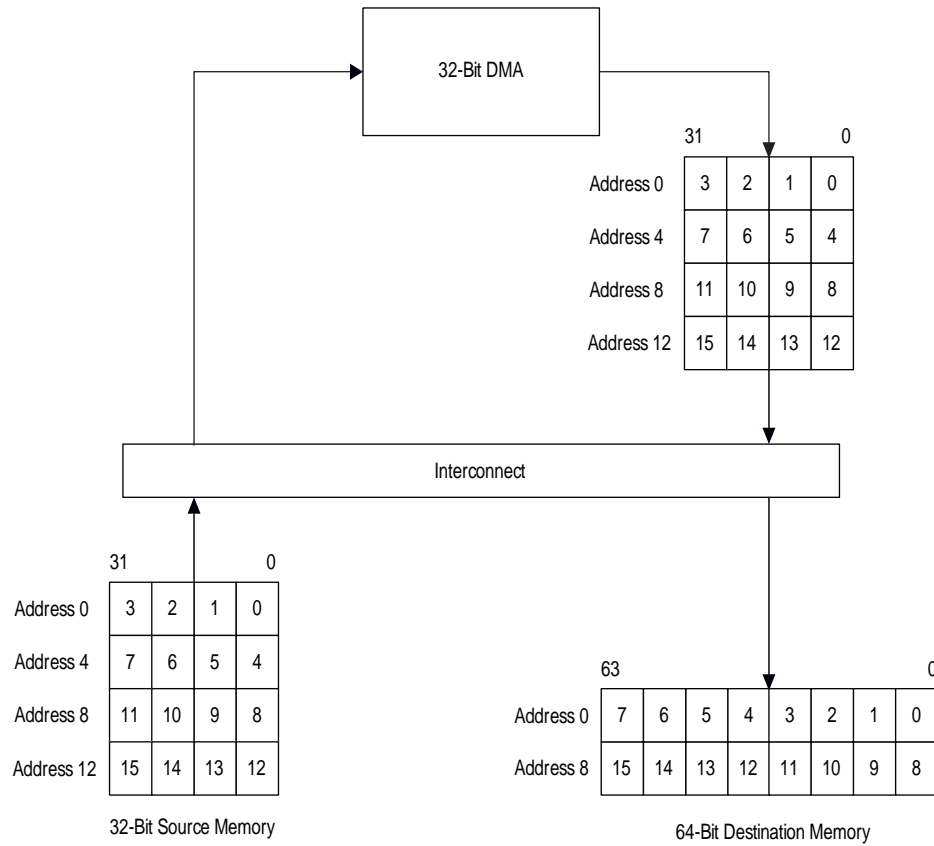
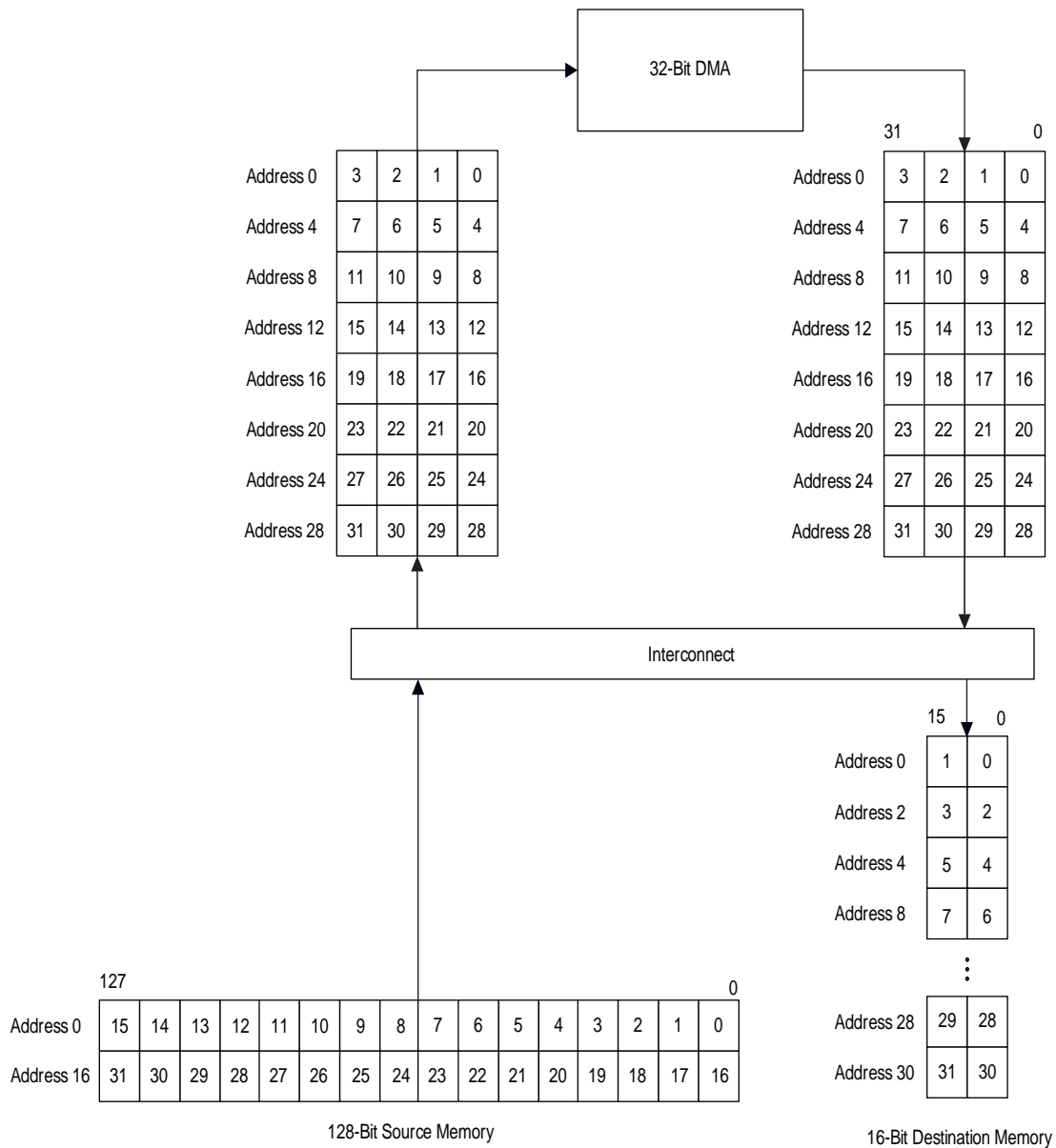


Figure 11-3. 128-Bit to 16-Bit Memory DMA Transfer

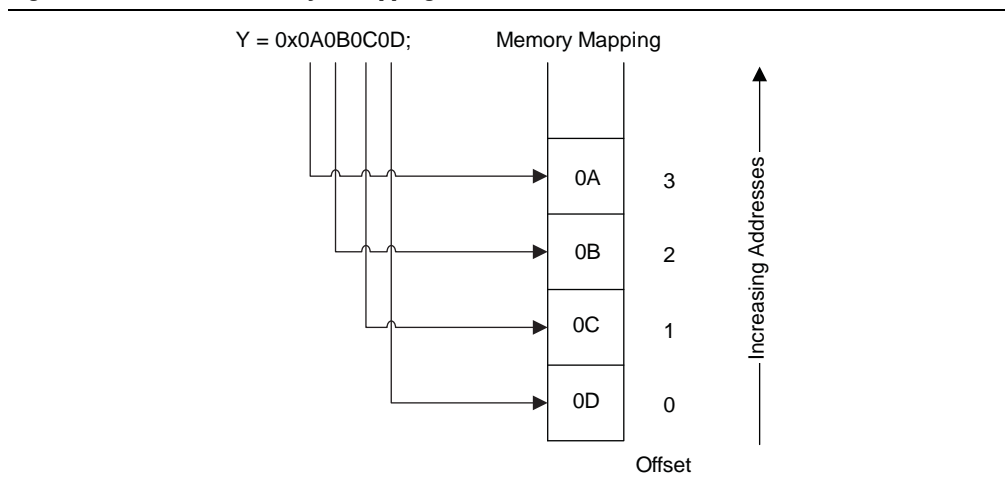


Nios II Processor Data Accesses

In the Nios II processor, the internal arithmetic byte ordering and the bus byte ordering are both little endian. Internally, the processor and its compiler map the least significant byte of a value to the lowest byte offset in memory.

For example, Figure 11-4 shows storing the 32-bit value 0x0A0B0C0D to the variable Y. The action maps the least significant byte 0x0D to offset 0 of the memory used to store the variable.

Figure 11-4. Nios II 32-Bit Byte Mapping



The Nios II processor is a 32-bit processor. For data larger than 32 bits, the same mapping of the least significant byte to lowest offset occurs. For example, if the value 0x0807060504030201 is stored to a 64-bit variable, the least significant byte 0x01 of the variable is stored to byte offset 0 of the variable in memory. The most significant byte 0x08 of the variable is stored to byte offset 7 of the variable in memory. The processor writes the least significant four bytes 0x04030201 of the variable to memory first, followed by the most significant four bytes 0x08070605.

The master interfaces of the Nios II processor comply with Avalon-MM bus byte ordering by providing read and write data to the interconnect in descending bit order with bits 7 down to 0 representing byte offset 0. Because the Nios II processor uses a 32-bit data path, the processor can access the interconnect with seven different aligned accesses. Table 11-1 shows the seven valid write accesses that the Nios II processor can present to the interconnect.

Table 11-1. Nios II Write Data Byte Mapping

Access Size (Bits)	Offset (Bytes)	Value	Byte Enable (Bits 3:0)	Write Data (Bits 31:24)	Write Data (Bits 23:16)	Write Data (Bits 15:8)	Write Data (Bits 7:0)
8	0	0x0A	0001	—	—	—	0x0A
8	1	0x0A	0010	—	—	0x0A	—
8	2	0x0A	0100	—	0x0A	—	—
8	3	0x0A	1000	0x0A	—	—	—
16	0	0x0A0B	0011	—	—	0x0A	0x0B
16	2	0x0A0B	1100	0x0A	0x0B	—	—
32	0	0x0A0B0C0D	1111	0x0A	0x0B	0x0C	0x0D

The code fragment shown in [Example 11-2](#) generates all seven of the accesses described in [Table 11-1](#) in the order presented in the table, where BASE is a location in memory aligned to a four-byte boundary.

Example 11-2. Nios II Write Data Byte Mapping Code

```
IOWR_8DIRECT(BASE, 0, 0x0A);
IOWR_8DIRECT(BASE, 1, 0x0A);
IOWR_8DIRECT(BASE, 2, 0x0A);
IOWR_8DIRECT(BASE, 3, 0x0A);
IOWR_16DIRECT(BASE, 0, 0x0A0B);
IOWR_16DIRECT(BASE, 2, 0x0A0B);
IOWR_32DIRECT(BASE, 0, 0x0A0B0C0D);
```

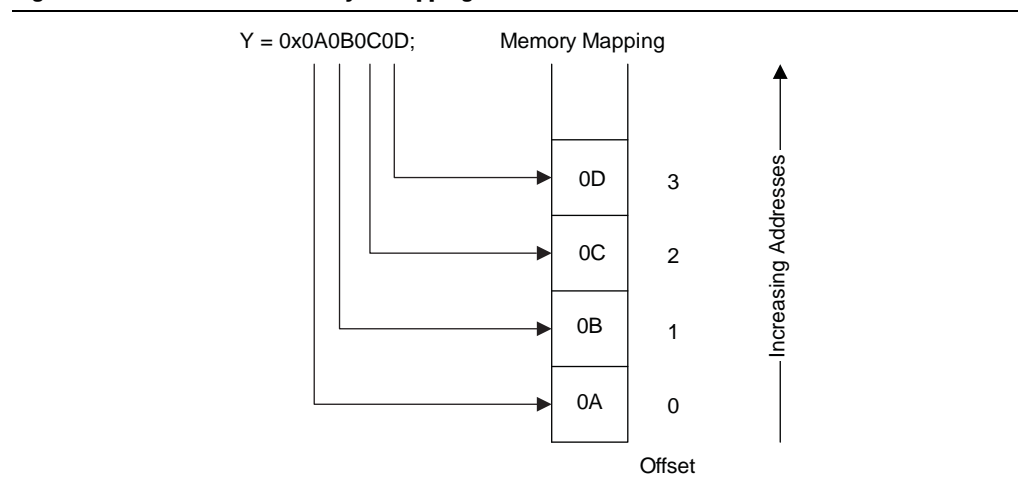
Adapting Processor Masters to be Avalon-MM Compliant

Because the way the Nios II processor presents data to the interconnect is Avalon-MM compliant, no extra effort is required to connect the processor to the interconnect. This section describes how to modify non-Avalon-MM compliant processor masters to achieve Avalon-MM compliance.

Some processors use a different arithmetic byte ordering than the Nios II processor uses, and as a result, typically use a different bus byte ordering than the Avalon-MM interface specification supports. When connecting one of these processors directly to the interconnect in a system containing other masters such as a Nios II processor, accesses to the same address result in accessing different physical byte lanes of the slave port. Mixing masters and slaves that conform to different bus byte ordering becomes nearly impossible to manage at a system level. These mixed bus byte ordering systems are difficult to maintain and debug. Altera requires that the master interfaces of any processors you add to your system are Avalon-MM compliant.

Processors that use a big endian arithmetic byte ordering, which is opposite to what the Nios II processor implements, map the most significant byte of the variable to the lowest byte offset of the variable in memory. For example, [Figure 11-5](#) shows how a PowerPC processor core stores the 32-bit value 0x0A0B0C0D to the memory containing the variable Y. The PowerPC stores the most significant byte, 0x0A, to offset 0 of the memory containing the variable.

Figure 11-5. Power PC 32-Bit Byte Mapping



This arithmetic byte ordering is the opposite of the ordering shown in “Nios II Processor Data Accesses” on page 11-6. Because the arithmetic byte ordering internal to the processor is independent of data bus byte ordering external to the processor, you can adapt processor masters with non-Avalon-MM compliant bus byte ordering to present Avalon-MM compliant data to the interconnect.

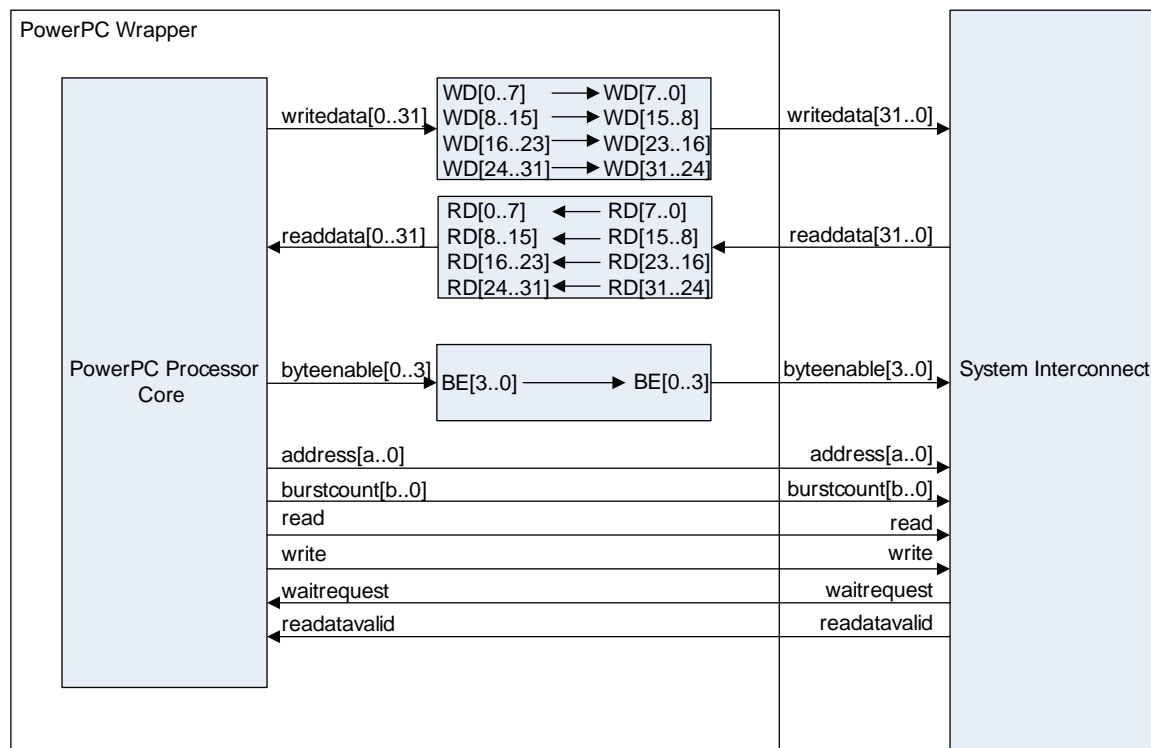
The following sections describe the bus byte ordering for the two most common processors that are not Avalon-MM compliant:

- “PowerPC Bus Byte Ordering”
- “ARM BE-32 Bus Byte Ordering”

PowerPC Bus Byte Ordering

The byte positions of the PowerPC bus byte ordering are aligned with the byte positions of the Avalon-MM interface specification; however, the bits within each byte are misaligned. PowerPC processor cores use an ascending bit ordering when the masters are connected to the interconnect. For example, a 32-bit PowerPC core labels the bus data bits 0 up to 31. A PowerPC core considers bits 0 up to 7 as byte offset 0. This layout differs from the Avalon-MM interface specification, which defines byte offset 0 as data bits 7 down to 0. To connect a PowerPC processor to the interconnect, you must rename the bits in each byte lane as Figure 11-6 shows.

Figure 11-6. PowerPC Bit-Renaming Wrapper



In [Figure 11-6](#), bit 0 is renamed to bit 7, bit 1 is renamed to bit 6, bit 2 is renamed to bit 5, and so on. By renaming the bits in each byte lane, byte offset 0 remains in the lower eight data bits. You must rename the bits in each byte lane separately. Renaming the bits by reversing all 32 bits creates a result that is not Avalon-MM compliant. For example, byte offset 0 would shift to data bits 31 down to 24, not 7 down to 0 as required.



Because the bits are simply renamed, this additional hardware does not occupy any additional FPGA resources nor impact the f_{MAX} of the data interface.

ARM BE-32 Bus Byte Ordering

Some ARM® cores use a bus byte ordering commonly referred to as *big endian 32* (BE-32). BE-32 processor cores use a descending bit ordering when the masters are connected to the interconnect. For example, an ARM BE-32 processor core labels the data bits 31 down to 0. Such a processor core considers bits 31 down to 24 as byte offset 0. This layout differs from the Avalon-MM specification, which defines byte 0 as data bits 7 down to 0.

A BE-32 processor core accesses memory using the bus mapping shown in [Table 11-2](#).

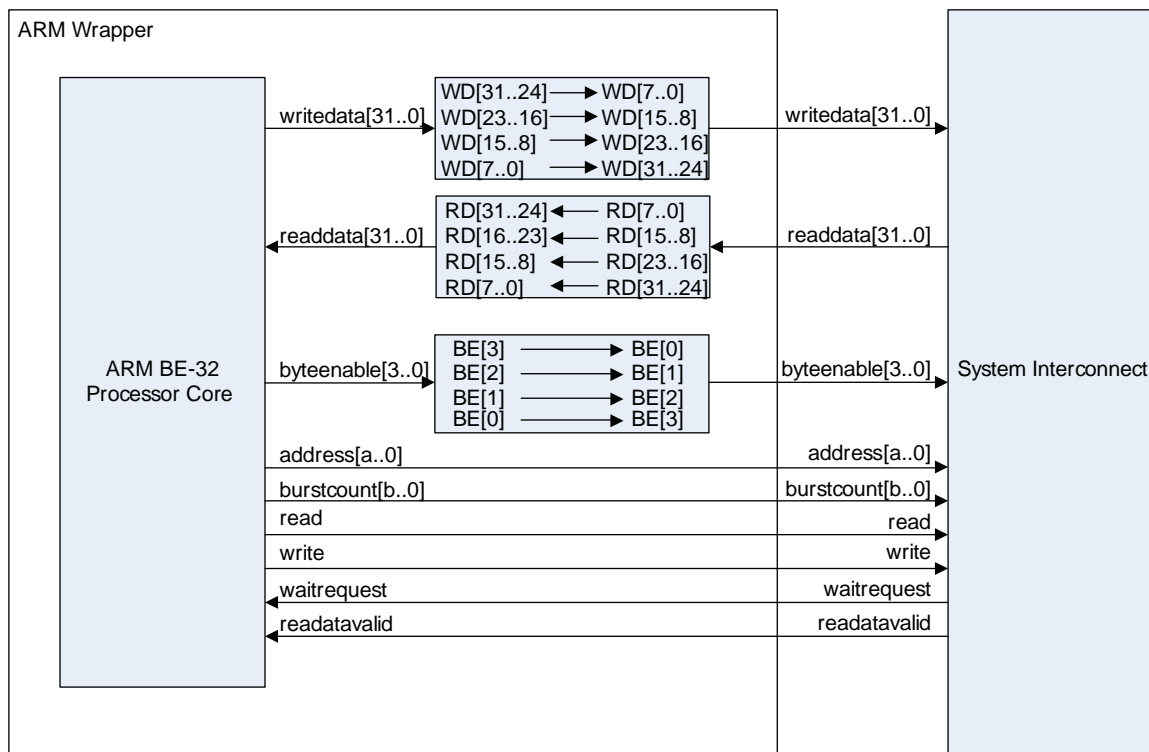
Table 11-2. ARM BE-32 Write Data Mapping


Access Size (Bits)	Offset (Bytes)	Value	Byte Enable (Bits 3:0)	Write Data (Bits 31:24)	Write Data (Bits 23:16)	Write Data (Bits 15:8)	Write Data (Bits 7:0)
8	0	0x0A	1000	0x0A	—	—	—
8	1	0x0A	0100	—	0x0A	—	—
8	2	0x0A	0010	—	—	0x0A	—
8	3	0x0A	0001	—	—	—	0x0A
16	0	0x0A0B	1100	0x0A	0x0B	—	—
16	2	0x0A0B	0011	—	—	0x0A	0x0B
32	0	0x0A0B0C0D	1111	0x0A	0x0B	0x0C	0x0D

The write access behavior of the BE-32 processor shown in [Table 11-2](#) differs greatly from the Nios II processor behavior shown in [Table 11-1](#). The only consistent access is the full 32-bit write access. In all the other cases, each processor accesses different byte lanes of the interconnect.

To connect a processor with BE-32 bus byte ordering to the interconnect, rename each byte lane as [Figure 11-7](#) shows.

Figure 11-7. ARM BE-32 Byte-Renaming Wrapper



 As in the case of the PowerPC wrapper logic, the ARM BE-32 wrapper does not consume any FPGA logic resources or degrade the f_{MAX} of the interface.

ARM BE-8 Bus Byte Ordering

Newer ARM processor cores offer a mode called *big endian 8* (BE-8). BE-8 processor master interfaces are Avalon-MM compliant. Internally, the BE-8 core uses a big endian arithmetic byte ordering; however, at the bus level, the core maps the data to the interconnect with the little endian orientation the Avalon-MM interface specification requires.

 This byte reordering sometimes requires special attention. For more information, refer to [“Arithmetic Byte Reordering”](#) on page 11-12.

Other Processor Bit and Byte Orders

There are numerous other ways to order the data leaving or entering a processor master interface. For those cases, the approach to achieving Avalon-MM compliance is the same. In general, apply the following three steps to any processor core to ensure Avalon-MM compliance:

1. Identify the bit order.
2. Identify the location of byte offset 0 of the master.

3. Create a wrapper around the processor core that renames the data signals so that byte 0 is located on data 7 down to 0, byte 1 is located on data 15 down to 8, and so on.

Arithmetic Byte Reordering

Altering your system to conform to Avalon-MM byte ordering modifies the internal arithmetic byte ordering of multibyte values as seen by the software. For example, an Avalon-MM compliant big endian processor core such as an ARM BE-8 processor accesses memory using the bus mapping shown in [Table 11-3](#).

Table 11-3. ARM BE-8 Write Data Mapping

Access Size (Bits)	Offset (Bytes)	Value	Byte Enable (Bits 3:0)	Write Data (Bits 31:24)	Write Data (Bits 23:16)	Write Data (Bits 15:8)	Write Data (Bits 7:0)
8	0	0x0A	0001	—	—	—	0x0A
8	1	0x0A	0010	—	—	0x0A	—
8	2	0x0A	0100	—	0x0A	—	—
8	3	0x0A	1000	0x0A	—	—	—
16	0	0x0A0B	0011	—	—	0x0B	0x0A
16	2	0x0A0B	1100	0x0B	0x0A	—	—
32	0	0x0A0B0C0D	1111	0x0D	0x0C	0x0B	0x0A

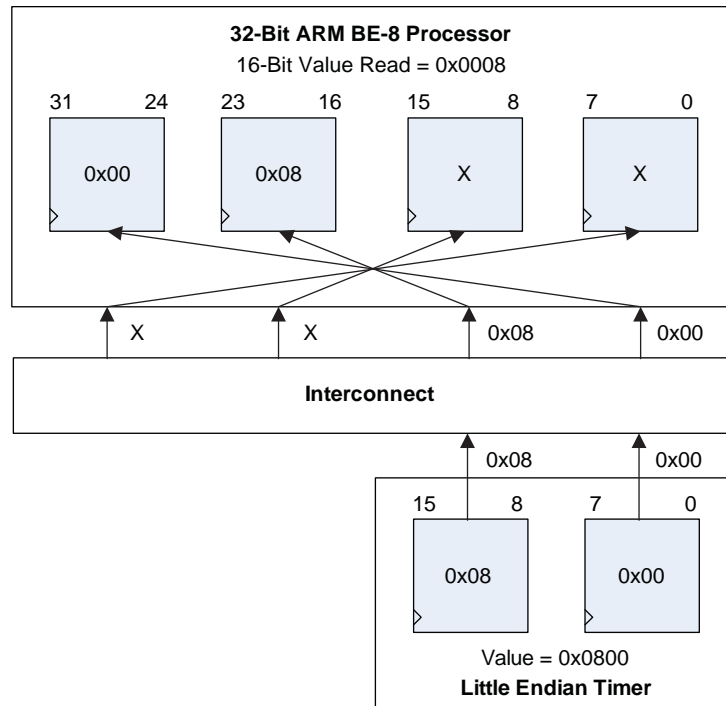
The big endian ARM BE-8 mapping ([Table 11-3](#)) matches the little endian Nios II processor mapping ([Table 11-1](#)) for all single byte accesses. If you ensure that your processor is Avalon-MM compliant, you can easily share individual bytes of data between big and little endian processors and peripherals.

However, making sure that the processor data master is Avalon-MM compliant only ensures that single byte accesses map to the same physical byte lanes of a slave port. In the case of multibyte accesses, the same byte lanes are accessed between the BE-8 and little endian processor; however, the value is not interpreted consistently. This mismatch is only important when the internal arithmetic byte ordering of the processor differs from other peripherals and processors in your system.

To correct the mismatch, you must perform arithmetic byte reordering in software for multibyte accesses. Interpretation of the data by the processor can vary based on the arithmetic byte ordering used by the processor and other processors and peripherals in the system.

For example, consider a 32-bit ARM BE-8 processor core that reads from a 16-bit little endian timer peripheral by performing a 16-bit read access. The ARM processor treats byte offset 0 as the most significant byte of any word. The timer treats byte offset 0 as the least significant byte of the 16-bit value. When the processor reads a value from the timer, the bytes of the value, as seen by software, are swapped. Figure 11-8 shows the swapping. A timer counter value of 0x0800 (2,048 clock ticks) is interpreted by the processor as 0x0008 (8 clock ticks) because the arithmetic byte ordering of the processor does not match the arithmetic byte ordering of the timer component.

Figure 11-8. ARM BE-8 Processor Accessing a Little Endian Peripheral



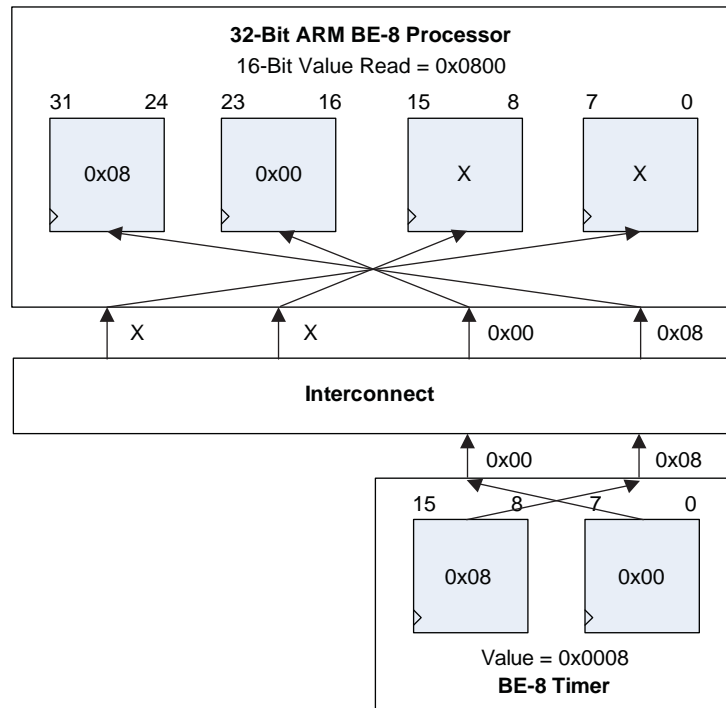
For the values to be interpreted accurately, the processor must either read each byte lane individually and then combine the two byte reads into a single 16-bit value in software, or read the single 16-bit value and swap the bytes in software.

The same issue occurs when you apply a bus-level renaming wrapper to an ARM BE-32 or PowerPC core. Both processor cores treat byte offset 0 as the most significant byte of any value. As a result, you must handle any mismatch between arithmetic byte ordering of data used by the processor and peripherals in your system.

On the other hand, if the timer in Figure 11-8 were to treat the most significant byte of the 16-bit value as byte 0 (big endian ordering), the data would arrive at the processor master in the same arithmetic byte ordering used by the processor. If the processor and the component internally implement the same arithmetic byte ordering, no software swapping of bytes is necessary for multibyte accesses.

Figure 11-9 shows how the value 0x0800 of a big endian timer is read by the processor. The value is retained without the need to perform any byte swapping in software after the read completes.

Figure 11-9. ARM BE-8 Processor Accessing a BE-8 Peripheral



System-Wide Design Recommendations

In the previous sections, we discussed arithmetic and bus byte ordering from a processor perspective. The same concepts directly apply to any component in your system. Any component containing Avalon-MM slave ports must also adhere to the Avalon-MM specification, which states that the data bits be defined in descending order with byte offset 0 positioned at bits 7 down to 0. As long as the component's slave port is Avalon-MM compliant, you can use any arithmetic byte ordering within the component.

System-Wide Arithmetic Byte Ordering

Typically, the most convenient arithmetic byte ordering to use throughout a system is the ordering the processor uses, if one is present. If the processor uses a different arithmetic byte ordering than the rest of the system, you must write software that rearranges the ordering for all multibyte accesses.

The majority of the IP provided by Altera that contains an Avalon-MM master or slave port uses little endian arithmetic byte ordering. If your system consists primarily of components provided by Altera, it is much easier to make the remainder of your system use the same little endian arithmetic byte ordering. When the entire system uses components that use the same arithmetic byte ordering and Avalon-MM bus byte ordering, arithmetic byte reordering within the processor or any component performing data accesses is not necessary.

Altera recommends writing your driver code to handle both big and little endian arithmetic byte ordering. For example, if the peripheral is little endian, write the peripheral driver to execute on both big and little endian processors. For little endian processors, no byte swapping is necessary. For big endian processors, all multibyte accesses requires a byte swap. Driver code selection is controlled at compile time or run time depending on the application and the peripheral.

System-Wide Arithmetic Byte Reordering in Software

If you cannot modify your system so that all the components use the same arithmetic byte ordering, you must implement byte reordering in software for multibyte accesses. Many processors today include instructions to accelerate this operation. If your processor does not have dedicated byte-reordering instructions, [Example 11-3](#) shows how you can implement byte reordering in software by leveraging the macros for 16-bit and 32-bit data.

Example 11-3. Software Arithmetic Byte Reordering

```
/* Perform 16-bit byte reordering */
#define SW_16_BIT_ARITHMETIC_REORDERING (data) ( \
    ((data) << 8) & 0xFF00 | \
    ((data) >> 8) & 0x00FF) \
)

/* Perform 32-bit byte reordering */
#define SW_32_BIT_ARITHMETIC_REORDERING (data) ( \
    ((data) << 24) & 0xFF000000 | \
    ((data) << 8) & 0x00FF0000 | \
    ((data) >> 8) & 0x0000FF00 | \
    ((data) >> 24) & 0x000000FF) \
)
```



Choose the appropriate instruction or macro to perform the byte reordering based on the width of the value that requires arithmetic byte reordering. Because arithmetic byte ordering only applies to individual values stored in memory or peripherals, you must reverse the bytes of the value without disturbing the data stored in neighboring memory locations. For example, if you load a 16-bit value from a peripheral that uses a different arithmetic byte ordering, you must swap two bytes in software. If you attempt to load two 16-bit values as a packed 32-bit read access, you must swap the individual 16-bit values independently. If you attempt to swap all four bytes at once, the two individual 16-bit values are swapped, which is not the original intent of the software developer.

Document Revision History

Table 11-4 shows the revision history for this document.

Table 11-4. Document Revision History

Date	Version	Changes
January 2011	1.0	Initial release.