9. SPI Core

Core Overview

SPI is an industry-standard serial protocol commonly used in embedded systems to connect microprocessors to a variety of off-chip sensor, conversion, memory, and control devices. The SPI core with Avalon® interface implements the SPI protocol and provides an Avalon Memory-Mapped (Avalon-MM) interface on the back end.

The SPI core can implement either the master or slave protocol. When configured as a master, the SPI core can control up to 16 independent SPI slaves. The width of the receive and transmit registers are configurable between 1 and 16 bits. Longer transfer lengths (for example, 24-bit transfers) can be supported with software routines. The SPI core provides an interrupt output that can flag an interrupt whenever a transfer completes.

The SPI core is SOPC Builder ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- “Functional Description”
- “Instantiating the SPI Core in SOPC Builder” on page 9–7
- “Device and Tools Support” on page 9–10
- “Software Programming Model” on page 9–10

Functional Description

The SPI core communicates using two data lines, a control line, and a synchronization clock:

- Master Out Slave In (mosi)—Output data from the master to the inputs of the slaves
- Master In Slave Out (miso)—Output data from a slave to the input of the master
- Serial Clock (sclk)—Clock driven by the master to slaves, used to synchronize the data bits
- Slave Select (ss_n)—Select signal (active low) driven by the master to individual slaves, used to select the target slave

The SPI core has the following user-visible features:

- A memory-mapped register space comprised of five registers: rxdata, txdata, status, control, and slaveselect
- Four SPI interface ports: sclk, ss_n, mosi, and miso
The registers provide an interface to the SPI core and are visible via the Avalon-MM slave port. The \texttt{sclk}, \texttt{ss_n}, \texttt{mosi}, and \texttt{miso} ports provide the hardware interface to other SPI devices. The behavior of \texttt{sclk}, \texttt{ss_n}, \texttt{mosi}, and \texttt{miso} depends on whether the SPI core is configured as a master or slave.

Figure 9–1 shows a block diagram of the SPI core in master mode.

**Figure 9–1. SPI Core Block Diagram**

The SPI core logic is synchronous to the clock input provided by the Avalon-MM interface. When configured as a master, the core divides the Avalon-MM clock to generate the SCLK output. When configured as a slave, the core’s receive logic is synchronized to SCLK input. The core’s Avalon-MM interface is capable of Avalon-MM transfers with flow control. The SPI core can be used in conjunction with a DMA controller with flow control to automate continuous data transfers between, for example, the SPI core and memory. See the *Timer Core* chapter for details.

**Example Configurations**

Two possible configurations are shown below. In Figure 9–2, the SPI core provides a slave interface to an off-chip SPI master.
In Figure 9–3 the SPI core provides a master interface driving multiple off-chip slave devices. Each slave device in Figure 9–3 must tristate its \texttt{miso} output whenever its select signal is not asserted.

The \texttt{ss}_{n} signal is active-low. However, any signal can be inverted inside the FPGA, allowing the slave-select signals to be either active high or active low.
Transmitter Logic

The SPI core transmitter logic consists of a transmit holding register (txdata) and transmit shift register, each n bits wide. The register width n is specified at system generation time, and can be any integer value from 1 to 16. After a master peripheral writes a value to the txdata register, the value is copied to the shift register and then transmitted when the next operation starts.

The shift register and the txdata register provide double buffering during data transmission. A new value can be written into the txdata register while the previous data is being shifted out of the shift register. The transmitter logic automatically transfers the txdata register to the shift register whenever a serial shift operation is not currently in process.

In master mode, the transmit shift register directly feeds the mosi output. In slave mode, the transmit shift register directly feeds the miso output. Data shifts out least-significant bit (LSB) first or most-significant bit (MSB) first, depending on the configuration of the SPI core.

Receiver Logic

The SPI core receive logic consists of a receive holding register (rxdata) and receive shift register, each n bits wide. The register width n is specified at system generation time, and can be any integer value from 1 to 16. A master peripheral reads received data from the rxdata register after the shift register has captured a full n-bit value of data.

The shift register and the rxdata register provide double buffering during data receiving. The rxdata register can hold a previously received data value while subsequent new data is shifting into the shift register. The receiver logic automatically transfers the shift register content to the rxdata register when a serial shift operation completes.

In master mode, the shift register is fed directly by the miso input. In slave mode, the shift register is fed directly by the mosi input. The receiver logic expects input data to arrive least-significant bit (LSB) first or most-significant bit (MSB) first, depending on the configuration of the SPI core.

Master and Slave Modes

At system generation time, the designer configures the SPI core in either master mode or slave mode. The mode cannot be switched at runtime.
Master Mode Operation

In master mode, the SPI ports behave as shown in Table 9–1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mosi</td>
<td>output</td>
<td>Data output to slave(s)</td>
</tr>
<tr>
<td>miso</td>
<td>input</td>
<td>Data input from slave(s)</td>
</tr>
<tr>
<td>sclk</td>
<td>output</td>
<td>Synchronization clock to all slaves</td>
</tr>
<tr>
<td>ss_nM</td>
<td>output</td>
<td>Slave select signal to slave M, where M is a number between 0 and 15.</td>
</tr>
</tbody>
</table>

Only an SPI master can initiate an operation between master and slave. In master mode, an intelligent host (e.g., a microprocessor) configures the SPI core using the control and slaveselect registers, and then writes data to the txdata buffer to initiate a transaction. A master peripheral can monitor the status of the transaction by reading the status register. A master peripheral can enable interrupts to notify the host whenever new data is received (i.e., a transfer has completed), or whenever the transmit buffer is ready for new data.

The SPI protocol is full duplex, so every transaction both sends and receives data at the same time. The master transmits a new data bit on the mosi output and the slave drives a new data bit on the miso input for each active edge of sclk. The SPI core divides the Avalon-MM system clock using a clock divider to generate the sclk signal.

When the SPI core is configured to interface with multiple slaves, the core has one ss_n signal for each slave, up to a maximum of sixteen slaves. During a transfer, the master asserts ss_n to each slave specified in the slaveselect register. Note that there can be no more than one slave transmitting data during any particular transfer, or else there will be a conflict on the miso input. The number of slave devices is specified at system generation time.
**Slave Mode Operation**

In slave mode, the SPI ports behave as shown in Table 9–2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mosi</td>
<td>input</td>
<td>Data input from the master</td>
</tr>
<tr>
<td>miso</td>
<td>output</td>
<td>Data output to the master</td>
</tr>
<tr>
<td>sclk</td>
<td>input</td>
<td>Synchronization clock</td>
</tr>
<tr>
<td>ss_n</td>
<td>input</td>
<td>Select signal</td>
</tr>
</tbody>
</table>

In slave mode, the SPI core simply waits for the master to initiate transactions. Before a transaction begins, the slave logic is continuously polling the ss_n input. When the master asserts ss_n (drives it low), the slave logic immediately begins sending the transmit shift register contents to the miso output. The slave logic also captures data on the mosi input, and fills the receive shift register simultaneously. Thus, a read and write transaction are carried out simultaneously.

An intelligent host (e.g., a microprocessor) writes data to the txdata registers, so that it will be transmitted the next time the master initiates an operation. A master peripheral reads received data from the rxdata register. A master peripheral can enable interrupts to notify the host whenever new data is received, or whenever the transmit buffer is ready for new data.

**Multi-Slave Environments**

When ss_n is not asserted, typical SPI cores set their miso output pins to high impedance. The Altera®-provided SPI slave core drives an undefined high or low value on its miso output when not selected. Special consideration is necessary to avoid signal contention on the miso output, if the SPI core in slave mode will be connected to an off-chip SPI master device with multiple slaves. In this case, the ss_n input should be used to control a tristate buffer on the miso signal. Figure 9–4 shows an example of the SPI core in slave mode in an environment with two slaves.
Avalon-MM Interface

The SPI core’s Avalon-MM interface consists of a single Avalon-MM slave port. In addition to fundamental slave read and write transfers, the SPI core supports Avalon-MM read and write transfers with flow control.

Instantiating the SPI Core in SOPC Builder

Designers use the MegaWizard® interface for the SPI core in SOPC Builder to configure the hardware feature set. The following sections describe the available options.

Master/Slave Settings

The designer can select either master mode or slave mode to determine the role of the SPI core. When master mode is selected, the following options are available: Generate Select Signals; SPI Clock Rate; and Specify Delay.

Generate Select Signals

This setting specifies how many slaves the SPI master will connect to. The acceptable range is 1 to 16. The SPI master core presents a unique ss_n signal for each slave.
**SPI Clock (sclk) Rate**

This setting determines the rate of the sclk signal that synchronizes data between master and slaves. The target clock rate can be specified in units of Hz, kHz or MHz. The SPI master core uses the Avalon-MM system clock and a clock divisor to generate sclk.

The actual frequency of sclk may not exactly match the desired target clock rate. The achievable clock values are:

\[
<\text{Avalon-MM system clock frequency}> / [2, 4, 6, 8, ...]
\]

The actual frequency achieved will not be greater than the specified target value. For example, if the system clock frequency is 50 MHz and the target value is 25 MHz, then the clock divisor is 2 and the actual sclk frequency achieves exactly 25 MHz. However, if the target frequency is 24 MHz, then the clock divisor is 4 and the actual sclk frequency becomes 12.5 MHz.

**Specify Delay**

Turning on this option causes the SPI master to add a time delay between asserting the ss_n signal and shifting the first bit of data. This delay is required by certain SPI slave devices. If the delay option is on, the designer must also specify the delay time in units of ns, us or ms. An example is shown in Figure 9–5.

![Figure 9–5. Time Delay Between Asserting ss_n and Toggling sclk](image)

The delay generation logic uses a granularity of half the period of sclk. The actual delay achieved is the desired target delay rounded up to the nearest multiple of half the sclk period, as shown in the following equations:

\[
p = \frac{1}{2} \times <\text{period of sclk}>
\]

\[
\text{actual delay} = \text{ceiling}( <\text{desired delay}> / p ) \times p
\]
Data Register Settings

The data register settings affect the size and behavior of the data registers in the SPI core. There are two data register settings:

- **Width**—This setting specifies the width of rxdata, txdata, and the receive and transmit shift registers. Acceptable values are from 1 to 16.
- **Shift direction**—This setting determines the direction that data shifts (MSB first or LSB first) into and out of the shift registers.

Timing Settings

The timing settings affect the timing relationship between the ss_n, sclk, mosi and miso signals. In this discussion the mosi and miso signals are referred to generically as “data”. There are two timing settings:

- **Clock polarity**—This setting can be 0 or 1. When clock polarity is set to 0, the idle state for sclk is low. When clock polarity is set to 1, the idle state for sclk is high.
- **Clock phase**—This setting can be 0 or 1. When clock phase is 0, data is latched on the leading edge of sclk, and data changes on trailing edge. When clock phase is 1, data is latched on the trailing edge of sclk, and data changes on the leading edge.

Figures 9–6 through 9–9 demonstrate the behavior of signals in all possible cases of clock polarity and clock phase.

**Figure 9–6. Clock Polarity = 0, Clock Phase = 0**

**Figure 9–7. Clock Polarity = 0, Clock Phase = 1**
Device and Tools Support

The SPI core can target all Altera FPGAs.

Software Programming Model

The following sections describe the software programming model for the SPI core, including the register map and software constructs used to access the hardware. For Nios® II processor users, Altera provides the HAL system library header file that defines the SPI core registers. The SPI core does not match the generic device model categories supported by the HAL, so it cannot be accessed via the HAL API or the ANSI C standard library. Altera provides a routine to access the SPI hardware that is specific to the SPI core.

Hardware Access Routines

Altera provides one access routine, `alt_avalon_spi_command()`, that provides general-purpose access to an SPI core configured as a master.
alt_avalon_spi_command()

Prototype: int alt_avalon_spi_command(alt_u32 base, alt_u32 slave, 
                                        alt_u32 write_length, 
                                        const alt_u8* wdata, 
                                        alt_u32 read_length, 
                                        alt_u8* read_data, 
                                        alt_u32 flags)

Thread-safe: No.
Available from ISR: No.
Include: <altera_avalon_spi.h>

Description: alt_avalon_spi_command() is used to perform a control sequence on 
the SPI bus. This routine is designed for SPI masters of 8-bit data width or less. 
Currently, it does not support SPI hardware with data-width greater than 8 bits. A 
single call to this function writes a data buffer of arbitrary length out the MOSI port, 
and then reads back an arbitrary amount of data from the MISO port. The function 
performs the following actions:

1. Asserts the slave select output for the specified slave. The first slave select 
   output is numbered 0, the next is 1, etc.
2. Transmits write_length bytes of data from wdata through the SPI 
   interface, discarding the incoming data on MISO.
3. Reads read_length bytes of data, storing the data into the buffer 
   pointed to by read_data. MOSI is set to zero during the read transaction.
4. Deasserts the slave select output, unless the flags field contains the value 
   ALT_AVALON_SPI_COMMAND_MERGE. If you want to transmit from 
   scattered buffers then you can call the function multiple times, specifying the 
   merge flag on all the accesses except the last.

This function is not thread safe. If you want to access the SPI bus from more than 
one thread, then you should use a semaphore or mutex to ensure that only one 
thread is executing within this function at any time.

Returns: The number of bytes stored in the read_data buffer.
Software Files

The SPI core is accompanied by the following software files. These files provide a low-level interface to the hardware.

- *altera_avalon_spi.h*—This file defines the core’s register map, providing symbolic constants to access the low-level hardware.
- *altera_avalon_spi.c*—This file implements low-level routines to access the hardware.

Legacy SDK Routines

The SPI core is also supported by the legacy SDK routines for the first-generation Nios processor. For details about these routines, refer to the SPI documentation that accompanied the first-generation Nios processor. For details about upgrading programs based on the legacy SDK to the HAL system library API, refer to AN 350: Upgrading Nios Processor Systems to the Nios II Processor.

Register Map

An Avalon-MM master peripheral controls and communicates with the SPI core via the six 16-bit registers, shown in Table 9–3. The table assumes an $n$-bit data width for rxdata and txdata.

<table>
<thead>
<tr>
<th>Internal Address</th>
<th>Register Name</th>
<th>15...11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rxdata (1)</td>
<td></td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>txdata (1)</td>
<td></td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>status (2)</td>
<td></td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td>RRDY</td>
<td>TRDY</td>
<td>TMT</td>
</tr>
<tr>
<td>3</td>
<td>control</td>
<td></td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sso (3)</td>
<td>IE</td>
<td>IRRDY</td>
<td>ITRDY</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
<td></td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>slaveselect</td>
<td></td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>(3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ITOE</td>
<td>IROE</td>
</tr>
</tbody>
</table>

Notes to Table 9–3:

(1) Bits 15 to $n$ are undefined when $n$ is less than 16.
(2) A write operation to the status register clears the roe, toe and e bits.
(3) Present only in master mode.

Reading undefined bits returns an undefined value. Writing to undefined bits has no effect.
rxdata Register

A master peripheral reads received data from the rxdata register. When the receive shift register receives a full \( n \) bits of data, the status register’s rrdy bit is set to 1 and the data is transferred into the rxdata register. Reading the rxdata register clears the rrdy bit. Writing to the rxdata register has no effect.

New data is always transferred into the rxdata register, whether or not the previous data was retrieved. If rrdy is 1 when data is transferred into the rxdata register (i.e., the previous data was not retrieved), a receive-overrun error occurs and the status register’s roe bit is set to 1. In this case, the contents of rxdata are undefined.

taxdata Register

A master peripheral writes data to be transmitted into the txdata register. When the status register’s trdy bit is 1, it indicates that the txdata register is ready for new data. The trdy bit is set to 0 whenever the txdata register is written. The trdy bit is set to 1 after data is transferred from the txdata register into the transmitter shift register, which readies the txdata holding register to receive new data.

A master peripheral should not write to the txdata register until the transmitter is ready for new data. If trdy is 0 and a master peripheral writes new data to the txdata register, a transmit-overrun error occurs and the status register’s toe bit is set to 1. In this case, the new data is ignored, and the content of txdata remains unchanged.

As an example, assume that the SPI core is idle (i.e., the txdata register and transmit shift register are empty), when a CPU writes a data value into the txdata holding register. The trdy bit is set to 0 momentarily, but after the data in txdata is transferred into the transmitter shift register, trdy returns to 1. The CPU writes a second data value into the txdata register, and again the trdy bit is set to 0. This time the shift register is still busy transferring the original data value, so the trdy bit remains at 0 until the shift operation completes. When the operation completes, the second data value is transferred into the transmitter shift register and the trdy bit is again set to 1.

status Register

The status register consists of bits that indicate status conditions in the SPI core. Each bit is associated with a corresponding interrupt-enable bit in the control register, as discussed in “control Register” on page 9–14.
A master peripheral can read status at any time without changing the value of any bits. Writing status does clear the roe, toe and e bits. Table 9–4 describes the individual bits of the status register.

### Table 9–4. status Register Bits

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 3  | ROE   | Receive-overrun error  
The ROE bit is set to 1 if new data is received while the rxdata register is full (that is, while the RRDY bit is 1). In this case, the new data overwrites the old. Writing to the status register clears the ROE bit to 0. |
| 4  | TOE   | Transmitter-overrun error  
The TOE bit is set to 1 if new data is written to the txdata register while it is still full (that is, while the TRDY bit is 0). In this case, the new data is ignored. Writing to the status register clears the TOE bit to 0. |
| 5  | TMT   | Transmitter shift-register empty  
The TMT bit is set to 0 when a transaction is in progress and set to 1 when the shift register is empty. |
| 6  | TRDY  | Transmitter ready  
The TRDY bit is set to 1 when the txdata register is empty. |
| 7  | RRDY  | Receiver ready  
The RRDY bit is set to 1 when the rxdata register is full. |
| 8  | E     | Error  
The E bit is the logical OR of the TOE and ROE bits. This is a convenience for the programmer to detect error conditions. Writing to the status register clears the E bit to 0. |

**control Register**

The control register consists of data bits to control the SPI core’s operation. A master peripheral can read control at any time without changing the value of any bits.

Most bits (IROE, ITOE, ITRDY, IRRDY, and IE) in the control register control interrupts for status conditions represented in the status register. For example, bit 1 of status is ROE (receiver-overrun error), and bit 1 of control is IROE, which enables interrupts for the ROE condition. The SPI core asserts an interrupt request when the corresponding bits in status and control are both 1.
The control register bits are shown in Table 9–5.

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>IROE</td>
<td>Setting IROE to 1 enables interrupts for receive-overrun errors.</td>
</tr>
<tr>
<td>4</td>
<td>ITOE</td>
<td>Setting ITOE to 1 enables interrupts for transmitter-overrun errors.</td>
</tr>
<tr>
<td>6</td>
<td>ITRDY</td>
<td>Setting ITRDY to 1 enables interrupts for the transmitter ready condition.</td>
</tr>
<tr>
<td>7</td>
<td>IRRDY</td>
<td>Setting IRRDY to 1 enables interrupts for the receiver ready condition.</td>
</tr>
<tr>
<td>8</td>
<td>IE</td>
<td>Setting IE to 1 enables interrupts for any error condition.</td>
</tr>
<tr>
<td>10</td>
<td>SSO</td>
<td>Setting SSO to 1 forces the SPI core to drive its ss_n outputs, regardless of whether a serial shift operation is in progress or not. The slaveselect register controls which ss_n outputs are asserted. SSO can be used to transmit or receive data of arbitrary size (i.e., greater than 16 bits).</td>
</tr>
</tbody>
</table>

After reset, all bits of the control register are set to 0. All interrupts are disabled and no ss_n signals are asserted after reset.

**slaveselect Register**

The slaveselect register is a bit mask for the ss_n signals driven by an SPI master. During a serial shift operation, the SPI master selects only the slave device(s) specified in the slaveselect register.

The slaveselect register is only present when the SPI core is configured in master mode. There is one bit in slaveselect for each ss_n output, as specified by the designer at system generation time. For example, to enable communication with slave device 3, set bit 3 of slaveselect to 1.

A master peripheral can set multiple bits of slaveselect simultaneously, causing the SPI master to simultaneously select multiple slave devices as it performs a transaction. For example, to enable communication with slave devices 1, 5, and 6, set bits 1, 5, and 6 of slaveselect. However, consideration is necessary to avoid signal contention between multiple slaves on their miso outputs.

Upon reset, bit 0 is set to 1, and all other bits are cleared to 0. Thus, after a device reset, slave device 0 is automatically selected.
Table 9–6 shows the revision history for this chapter.

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 2007 v7.1.0</td>
<td>● Chapter 9 was formerly chapter 7.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>● Added table of contents to Overview section.</td>
<td></td>
</tr>
<tr>
<td>March 2007 v7.0.0</td>
<td>No change from previous release.</td>
<td></td>
</tr>
<tr>
<td>November 2006 v6.1.0</td>
<td>● Updated Avalon terminology because of changes to Avalon technologies</td>
<td>For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.</td>
</tr>
<tr>
<td></td>
<td>● Changed old “Avalon switch fabric” term to “system interconnect fabric”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>● Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface”</td>
<td></td>
</tr>
<tr>
<td>May 2006 v6.0.0</td>
<td>No change from previous release.</td>
<td></td>
</tr>
<tr>
<td>December 2005 v5.1.1</td>
<td>Changed Avalon “streaming” terminology to “flow control” based on a change to the Avalon Interface Specification.</td>
<td></td>
</tr>
<tr>
<td>October 2005 v5.1.0</td>
<td>No change from previous release.</td>
<td></td>
</tr>
<tr>
<td>May 2005 v5.0.0</td>
<td>No change from previous release. Previously in the Nios II Processor Reference Handbook.</td>
<td></td>
</tr>
<tr>
<td>September 2004 v1.1</td>
<td>Updates for Nios II 1.01 release.</td>
<td></td>
</tr>
<tr>
<td>May 2004 v1.0</td>
<td>Initial release.</td>
<td></td>
</tr>
</tbody>
</table>