Generic Serial Flash Interface Intel® FPGA IP User Guide

Updated for Intel® Quartus® Prime Design Suite: 20.3

IP Version: 20.0.0
1. Generic Serial Flash Interface Intel® FPGA IP User Guide

The Generic Serial Flash Interface Intel® FPGA IP provides access to Serial Peripheral Interface (SPI) flash devices. The Generic Serial Flash Interface IP is a more efficient alternative compared to the ASMI Parallel Intel FPGA IP and ASMI Parallel II Intel FPGA IP. The Generic Serial Flash Interface Intel FPGA IP supports Intel configuration devices as well as flash from different vendors. Intel recommends you to use the Generic Serial Flash Interface Intel FPGA IP for new designs.

You can use the Generic Serial Flash Interface Intel FPGA IP to write the following data to the flash device:

- Configuration memory\(^{(1)}\)—configuration data for Active Serial (AS) configuration scheme.
- General purpose memory—application-specific data.

The Generic Serial Flash Interface IP supports the following features:

- Single, dual or quad I/O mode.
- Direct flash access via the Avalon® memory-mapped slave interface which allows a processor such as Nios® II to directly execute codes from the flash.
- Up to 3 flash device support (Intel Arria® 10 devices, Intel Cyclone® 10 GX devices, and other FPGA devices with flashes that are connected to the FPGA GPIO pins).
- IP control register for accessing flash control and status registers.
- Programmable clock generator with run-time baud rate change for flash device clock.
- Programmable chip select delay.
- Read data capturing logic when running with high frequency.
- FPGA active serial memory interface (ASMI) block atom connection to the active serial (AS) pins or export to FPGA I/O pins.

Related Information

- Generic Serial Flash Interface Intel FPGA IP Reference Design on page 16
- Generic Serial Flash Interface Intel FPGA IP Core Reference Design Files
- How do I enable Micron’s MT25Q device support in replacement to End Of Life (EOL) EPCQ(>=256Mb) and EPCQ-L devices?
- Configuration Devices

\(^{(1)}\) The supported flash devices for configuration memory are, EPCQ, EPCQ-A, EPCQ-L, and Micron* MT25Q (256Mb to 2Gb) devices.
• Using the Generic Serial Flash Interface (ODEVGSFI) Training Course

1.1. Release Information

IP versions are the same as the Intel Quartus® Prime Design Suite software versions up to v19.1. From Intel Quartus Prime Design Suite software version 19.2 or later, IP cores have a new IP versioning scheme.

The IP version (X.Y.Z) number may change from one Intel Quartus Prime software version to another. A change in:
• X indicates a major revision of the IP. If you update your Intel Quartus Prime software, you must regenerate the IP.
• Y indicates the IP includes new features. Regenerate your IP to include these new features.
• Z indicates the IP includes minor changes. Regenerate your IP to include these changes.

Table 1. Generic Serial Flash Interface Intel FPGA IP Release Information

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP Version</td>
<td>20.0.0</td>
</tr>
<tr>
<td>Intel Quartus Prime Pro Edition Version</td>
<td>20.3</td>
</tr>
<tr>
<td>Release Date</td>
<td>2020.09.28</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Quartus Prime Standard Edition Version</td>
<td>20.1</td>
</tr>
<tr>
<td>Release Date</td>
<td>2020.04.13</td>
</tr>
</tbody>
</table>

Note: The new IP versioning scheme is only available for the Generic Serial Flash Interface Intel FPGA IP in the Intel Quartus Prime Pro Edition software.

1.2. Device Family Support

The Generic Serial Flash Interface IP is supported in the following devices:
• Intel Agilex™
• Intel Stratix® 10
• Intel Arria 10
• Intel Cyclone 10 GX
• Intel Cyclone 10 LP
• Intel MAX® 10 (For general purpose memory only)
• Stratix V
• Arria V
• Cyclone V
• Stratix IV
• Cyclone IV
• Arria II

Note: For Intel Agilex, Intel Stratix 10, and Intel MAX 10 devices, export the flash pin by enabling **Enable SPI pins interface** parameter of this IP. For Intel Agilex and Intel Stratix 10 devices, the IP can only access flash that is connected to FPGA GPIO pins. The IP cannot be used to access flash that is connected to SDM for configuration purpose.

**Related Information**
**Configuration Devices**
Provides more information about the third-party flash support.

### 1.3. Signals

**Figure 1. Signal Block Diagram**
The inclusion and width of some signals depend on the features selected.

**Table 2. Ports Description**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avalon Memory-Mapped Slave Interface for CSR (<strong>avl_csr</strong>)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>avl_csr_addr</strong></td>
<td>6</td>
<td>Input</td>
<td>Avalon memory-mapped address bus. The address bus is in word addressing.</td>
</tr>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Signal</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>avl_csr_read</td>
<td>1</td>
<td>Input</td>
<td>Avalon memory-mapped read control to the CSR.</td>
</tr>
<tr>
<td>avl_csr_rdata</td>
<td>32</td>
<td>Output</td>
<td>Avalon memory-mapped read data bus from the CSR.</td>
</tr>
<tr>
<td>avl_csr_write</td>
<td>1</td>
<td>Input</td>
<td>Avalon memory-mapped write control to the CSR.</td>
</tr>
<tr>
<td>avl_csr_wrdata</td>
<td>32</td>
<td>Input</td>
<td>Avalon memory-mapped write data bus to CSR.</td>
</tr>
<tr>
<td>avl_csr_waitrequest</td>
<td>1</td>
<td>Output</td>
<td>Avalon memory-mapped waitrequest control from the CSR.</td>
</tr>
<tr>
<td>avl_csr_rddata_valid</td>
<td>1</td>
<td>Output</td>
<td>Avalon memory-mapped read data valid that indicates the CSR read data is available.</td>
</tr>
<tr>
<td>avl_csr_byteenable</td>
<td>4</td>
<td>Input</td>
<td>Avalon memory-mapped byteenable control to the CSR. Available when you enable the \textit{Use byteenable for CSR} parameter.</td>
</tr>
</tbody>
</table>

### Avalon Memory-Mapped Slave Interface for Memory Access (avl_mem)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>avl_mem_write</td>
<td>1</td>
<td>Input</td>
<td>Avalon memory-mapped write control to the memory</td>
</tr>
<tr>
<td>avl_mem_burstcount</td>
<td>7</td>
<td>Input</td>
<td>Avalon memory-mapped burst count for the memory. The value range from 1 to 64 (Max page size).</td>
</tr>
<tr>
<td>avl_mem_waitrequest</td>
<td>1</td>
<td>Output</td>
<td>Avalon memory-mapped waitrequest control from the memory.</td>
</tr>
<tr>
<td>avl_mem_read</td>
<td>1</td>
<td>Input</td>
<td>Avalon memory-mapped read control to the memory</td>
</tr>
<tr>
<td>avl_mem_addr</td>
<td>N</td>
<td>Input</td>
<td>Avalon memory-mapped address bus. The address bus is in word addressing. The width of the address depends on the flash memory density. If you are using Intel Arria 10, and Intel Cyclone 10 GX or any supported devices with general purpose I/O with multiples flashes, write the CSR to select the chip select. The IP targets the selected flash when being accessed via this address.</td>
</tr>
<tr>
<td>avl_mem_wrdata</td>
<td>32</td>
<td>Input</td>
<td>Avalon memory-mapped write data bus to the memory</td>
</tr>
<tr>
<td>avl_mem_rddata</td>
<td>32</td>
<td>Output</td>
<td>Avalon memory-mapped read data bus from the memory</td>
</tr>
<tr>
<td>avl_mem_rddata_valid</td>
<td>1</td>
<td>Output</td>
<td>Avalon memory-mapped read data valid that indicates the memory read data is available.</td>
</tr>
<tr>
<td>avl_mem_byteenable</td>
<td>4</td>
<td>Input</td>
<td>Avalon memory-mapped write data enable bus to memory. During bursting mode, byteenable bus will be logic high, 4'b1111.</td>
</tr>
</tbody>
</table>

### Clock and Reset

<table>
<thead>
<tr>
<th>Signal</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>1</td>
<td>Input</td>
<td>Input clock to clock the IP.</td>
</tr>
<tr>
<td>reset</td>
<td>1</td>
<td>Input</td>
<td>Asynchronous reset to reset the IP.</td>
</tr>
</tbody>
</table>

### Interrupt

<table>
<thead>
<tr>
<th>Signal</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irg</td>
<td>1</td>
<td>Output</td>
<td>Interrupt signal that indicate if there is an illegal write or illegal erase.</td>
</tr>
</tbody>
</table>

### Conduit Interface\(^{(2)}\)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Width</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flash_data</td>
<td>4</td>
<td>Bidirectional</td>
<td>Input or output port to feed data from the flash device.</td>
</tr>
<tr>
<td>flash_dclk</td>
<td>1</td>
<td>Output</td>
<td>Provides clock signal to the flash device.</td>
</tr>
<tr>
<td>flash_ncs</td>
<td>1/3</td>
<td>Output</td>
<td>Provides the ncs signal to the flash device.</td>
</tr>
</tbody>
</table>

\(^{(2)}\) Available when you enable the \textit{Enable SPI pins interface} parameter.
1.4. Parameters

Table 3. Parameter Settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Legal Values</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Density</td>
<td>1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048</td>
<td>Density of the flash device used in Mb.</td>
</tr>
<tr>
<td>Disable dedicated Active Serial</td>
<td>—</td>
<td>Routes the signals to the top level of your design. Enable this when you</td>
</tr>
<tr>
<td>interface</td>
<td></td>
<td>want to include the Serial Flash Loader Intel FPGA IP in your design.</td>
</tr>
<tr>
<td>Enable SPI pins interface</td>
<td>—</td>
<td>Translates the signals to the SPI pin interface.</td>
</tr>
<tr>
<td>Number of Chip Select used</td>
<td>1, 2, 3</td>
<td>Selects the number of chip select connected to the flash.</td>
</tr>
<tr>
<td>Enable flash simulation model</td>
<td>—</td>
<td>Uses the default EPCQ1024 simulation model for simulation. When disabled,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>refer to AN-720: Simulating the ASMI Block in Your Design for creating a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wrapper to use with other flash simulation model.</td>
</tr>
<tr>
<td>Use byteenable for CSR</td>
<td>—</td>
<td>Turns on byteenable for CSR writedata interface.</td>
</tr>
</tbody>
</table>

Related Information
AN-720: Simulating the ASMI Block in Your Design

1.5. Register Map

Table 4. Register Map

- Each address offset in the following table represents 1 word of memory address space.
- IP_CLK is the clock that drives the IP.
- SCLK is the clock that drives the flash device.

<table>
<thead>
<tr>
<th>Offset (Hex)</th>
<th>Register Name</th>
<th>R/W</th>
<th>Field Name</th>
<th>Bit</th>
<th>Default Value (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Control Register</td>
<td>R/W</td>
<td>Addressing</td>
<td>8</td>
<td>0x0</td>
<td>Addressing mode for read and write operation:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mode</td>
<td></td>
<td></td>
<td>• 0x0: 3-bytes addressing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 0x1: 4-bytes addressing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For 4-byte addressing mode, you must enable 4-byte address by sending</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>command to the flash.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>This bit affects direct access to memory via the Avalon memory-mapped</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>interface for both write and read operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/W</td>
<td>Chip select</td>
<td>7:4</td>
<td>0x0</td>
<td>Selects the flash device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 0x0: To select first device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 0x1: To select second device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 0x2: To select third device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved</td>
<td>3:1</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>Offset (Hex)</td>
<td>Register Name</td>
<td>R/W</td>
<td>Field Name</td>
<td>Bit</td>
<td>Default Value (Hex)</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
<td>-----</td>
<td>---------------------</td>
<td>-----</td>
<td>---------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>R/W</td>
<td></td>
<td>Enable</td>
<td>0</td>
<td>0x1</td>
<td>Set this bit to 0 to disable the output of the IP and put all output signal to high impedance state. This can be used to share bus with other devices.</td>
</tr>
<tr>
<td>1</td>
<td>SPI Clock Baud-rate Register</td>
<td>Reserved</td>
<td>31:5</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R/W</td>
<td></td>
<td>Baud rate divisor</td>
<td>4:0</td>
<td>0x10</td>
<td>The IP has an internal clock divider to generate the clock that connects to the flash device. The possible divisor value is from 2 to 32 with the increment of 2. So, the maximum clock that the flash run is half of the clock of the IP. Ex if the IP is run with 100 Mhz clock, then the clock of the flash is at 50 Mhz. By default, the clock is set to the lowest clock (/32) to ensure that the IP works in most cases. Divisor values: • 0x1 : /2 • 0x2 : /4 • 0x3 : /6 • ... • 0xF : /30 • 0x10 : /32</td>
</tr>
<tr>
<td>2</td>
<td>CS Delay Setting Register</td>
<td>Reserved</td>
<td>31:12</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R/W</td>
<td></td>
<td>tSHSL</td>
<td>11:8</td>
<td></td>
<td>This register setting controls the tSHSL. • 0: tSHSL is 3 IP_CLK. • n: tSHSL is 3+n IP_CLK.</td>
</tr>
<tr>
<td></td>
<td>R/W</td>
<td></td>
<td>CS de-assert</td>
<td>7:4</td>
<td>0x0</td>
<td>Sets the chip select de-assertion delay. • 0: Chip select is de-asserted at the last falling edge of SCLK. • n: Chip select is de-asserted n number of clocks after the last falling edge of SCLK.</td>
</tr>
<tr>
<td></td>
<td>R/W</td>
<td></td>
<td>CS assert</td>
<td>3:0</td>
<td>0x0</td>
<td>Sets the chip select assertion delay. • 0: Chip select is asserted half flash clock period before the first rising edge of SCLK. • n: Chip select is asserted half flash clock period plus n number of IP_CLK. (3)</td>
</tr>
<tr>
<td>3</td>
<td>Read Capturing Register</td>
<td>Reserved</td>
<td>31:4</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R/W</td>
<td></td>
<td>Read delay</td>
<td>3:0</td>
<td>0x0</td>
<td>The clock to output timing of the flash plus the board trace, I/O pin timing can contribute to high value of delay to the data arriving at the IP logic. The delay capture provides a way for the IP to delay its reading logic to compensate for those delays.</td>
</tr>
</tbody>
</table>

(3) Intel recommends that you set the chip select assertion delay to 5 if you are running the IP clock at 100 MHz.
<table>
<thead>
<tr>
<th>Offset (Hex)</th>
<th>Register Name</th>
<th>R/W</th>
<th>Field Name</th>
<th>Bit</th>
<th>Default Value (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Operating Protocols</td>
<td>R/W</td>
<td>Reserved</td>
<td>31:18</td>
<td></td>
<td>Delay the read data logic by a value of the IP_CLK cycles.</td>
</tr>
<tr>
<td></td>
<td>Setting Register</td>
<td></td>
<td>Read data out transfer mode</td>
<td>17:16</td>
<td>0x0</td>
<td>Transfer mode for read data output.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Reserved</td>
<td>15:14</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/W</td>
<td>Read address transfer mode</td>
<td>13:12</td>
<td>0x0</td>
<td>Transfer mode for read address input Description as bit 1:0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Reserved</td>
<td>11:10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/W</td>
<td>Write Data in transfer mode</td>
<td>9:8</td>
<td>0x0</td>
<td>Transfer mode for write data input Description as bit 1:0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Reserved</td>
<td>7:6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/W</td>
<td>Write address transfer mode</td>
<td>5:4</td>
<td>0x0</td>
<td>Transfer mode for write address input Description as bit 1:0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Reserved</td>
<td>3:2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/W</td>
<td>Instruction transfer mode</td>
<td>1:0</td>
<td>0x0</td>
<td>Transfer mode for opcode: • 0x0: Standard SPI mode – command input is sent on DQ0. • 0x1: Dual I/O mode – command input is sent on DQ[1:0]. • 0x2: Quad I/O mode – command input is sent on DQ[3:0]. This setting affects the flash command register. For example, if this field is set to 0x1, flash common operations (such as read id, read status, write status register) uses 0x1 as well.</td>
</tr>
<tr>
<td>5</td>
<td>Read Instruction Register</td>
<td>R/W</td>
<td>Dummy cycles</td>
<td>12:8</td>
<td>0xA</td>
<td>Number of default dummy cycles used for read operation. Refer to the respective flash device datasheet.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/W</td>
<td>Read opcode</td>
<td>7:0</td>
<td>0x03</td>
<td>The opcode for read operation. Refer to the respective flash device datasheet to select the correct opcode according to the transfer mode setting.</td>
</tr>
<tr>
<td>6</td>
<td>Write Instruction Register</td>
<td>R/W</td>
<td>Polling opcode</td>
<td>15:8</td>
<td>0x05</td>
<td>The opcode to check if the write operation has been completed. After write operation is completed, the IP releases the wait request of the Avalon memory-mapped interface. In applicable devices, you can set as the status register or flag status register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/W</td>
<td>Write opcode</td>
<td>7:0</td>
<td>0x02</td>
<td>The opcode for write operation. Refer to the respective flash device datasheet to select the correct opcode according to the transfer mode setting.</td>
</tr>
<tr>
<td>7</td>
<td>Flash Command</td>
<td>Reserved</td>
<td>31:21</td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

*continued...*
<table>
<thead>
<tr>
<th>Offset (Hex)</th>
<th>Register Name</th>
<th>R/W</th>
<th>Field Name</th>
<th>Bit</th>
<th>Default Value (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Setting Register(4)</td>
<td>R/W</td>
<td>Number of dummy cycles</td>
<td>20:16</td>
<td>0x0</td>
<td>The number of dummy cycles. Set to 0 when the operation does not require any dummy cycles. Refer to the respective flash device datasheet for dummy clock requirements.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/W</td>
<td>Number of data bytes</td>
<td>15:12</td>
<td>0x08</td>
<td>The number of write or read data. This works together with bit 11. If the value is Set to 0 if the operation has no write or read data, for example, write enable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/W</td>
<td>Data type</td>
<td>11</td>
<td>0x01</td>
<td>Indicates the type of data (bit [15:12]). • 0: Number of byte declared in [15:12] is write data to flash device • 1: Number of byte declared in [15:12] is read data from flash device</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/W</td>
<td>Number of address bytes</td>
<td>10:8</td>
<td>0x0</td>
<td>Number of address bytes to send to the flash device. Either 3 or 4 bytes If this is set to zero then the operation does not carry any address byte.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/W</td>
<td>Opcode</td>
<td>7:0</td>
<td>0x05</td>
<td>The opcode of the operation.</td>
</tr>
<tr>
<td>8</td>
<td>Flash Command Control Register</td>
<td></td>
<td>Reserved</td>
<td>31:1</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>W</td>
<td>Start</td>
<td>0</td>
<td>0x0</td>
<td>Write 1 to this bit to start the operation.</td>
</tr>
<tr>
<td>9</td>
<td>Flash Command Address Register</td>
<td>R/W</td>
<td>Stating address</td>
<td>31:0</td>
<td>31:0</td>
<td>Address of flash command.</td>
</tr>
<tr>
<td>A</td>
<td>Flash Command Write Data 0 Register</td>
<td>R/W</td>
<td>Lower 4 bytes write data</td>
<td>31:0</td>
<td>0x0</td>
<td>The first 4-byte of write data to flash device.</td>
</tr>
<tr>
<td>B</td>
<td>Flash Command Write Data 1 Register</td>
<td>R/W</td>
<td>Upper 4 bytes write data</td>
<td>31:0</td>
<td>0x0</td>
<td>The last 4-byte of write data to the flash device.</td>
</tr>
<tr>
<td>C</td>
<td>Flash Command Read Data 0 Register</td>
<td>R</td>
<td>Lower 4 bytes read data</td>
<td>31:0</td>
<td>0x0</td>
<td>The first 4-byte of read data from flash device.</td>
</tr>
<tr>
<td>D</td>
<td>Flash Command Read Data 1 Register</td>
<td>R</td>
<td>Upper 4 bytes read data</td>
<td>31:0</td>
<td>0x0</td>
<td>The last 4-byte of read data from the flash device.</td>
</tr>
</tbody>
</table>

1.6. Using Generic Serial Flash Interface Intel FPGA IP

The Generic Serial Flash Interface Intel FPGA IP core interfaces are Avalon memory-mapped compliant. For more details, refer to the Avalon specification.

(4) Default setting is for read status command.
Note:
- For operations that require write value to flash, you must perform write enable operation first.
- You must read the flag status register every time you issue a write or erase command.
- In case of support multiples flash devices, you must write chip select register to select the correct flash device before performing any operation to the specific flash device.

1.6.1. Control Status Register Byte Enable

The byte enable for the Control Status Register (CSR) interface feature allows you to write to all CSRs, from 0x0 to 0xD, in the Generic Serial Flash Interface Intel FPGA IP while selecting only certain bytes to write.

The `avl_csr_byteenable` port provides support for this feature.

- For normal CSRs (all CSRs except write data registers, 0xA and 0xB), the CSRs retain their values if a particular byteenable is not enabled, and only write the new value for enabled bytes.
- The writedata CSRs (0xA and 0xB) store the write data to target flash. The IP only writes enabled bytes with the corresponding data and does not retain the old values for disabled bytes.
  - The IP writes the writedata for all valid enabled bytes into the flash at a specified starting address as the first data.
  - Set the correct number of data bytes register in bit[15:12] in the flash command setting register (0x7). If you intend to write both write data 0 and write data 1 into the flash, you must write the number of data bytes to 8, regardless of how many CSR byteenable bits are set. The IP writes the enabled byte as the first data into the starting address for write data 0, and then fills the rest of the bytes for write data 0 as FF. The IP does the same for write data 1. If you intend to write one write data 0 or write data 1 into flash, you must set the number of data bytes to the same as `avl_csr_byteenable` bytes. In other words, if you enable 2 bytes in `avl_csr_byteenable` (4'b0110), the number of data bytes to be sent is also 2 bytes. The IP writes only the enabled data directly into the address that you specified in the flash command address register (0x9).

You have the option to turn on Use byteenable for CSR to enable the byte enable for the CSR interface feature in the parameter editor of the Generic Serial Flash Interface Intel FPGA IP.

The following steps are the programming flow using the CSR to write data to flash (Macronix® flash):
1. Write the address that you intend to write data into. Write the address in the flash command address register (0x9). For example, write 0x2001.

2. Write the writedata in the flash command write data 0 register (0xA) or flash command write data 1 register (0xB) using avl_csr_byteenable to select the desired bytes only. For example, the CSR byteenable of 4'b0110) for write data 0 is 44332211 and write data 1 is 88776655.

3. Start the setup of the write operation opcode in the IP by writing to flash command setting register (0x7) with ‘write enable’ opcode (h06) and then set the control bit to 1 in flash control register (0x8).

4. Next, write the write status register opcode (h01) in the same register (0x7). Set the data type as ‘write’ and write the number of data bytes as 8. For example, both write data 0 and write data 1 have 2 bytes enabled. You must write all the bytes (that is, 8 bytes) even if the intended total bytes to be written is 4.

The resulting write operation will write 77663322 (4 bytes) into the flash address 0x2001.

Table 5. Visualization of CSR byteenable for Write Data into Flash Memory

<table>
<thead>
<tr>
<th>CSR Enabled Bytes in Write Data</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write data 0</td>
<td>44</td>
<td>33 (5)</td>
<td>22 (5)</td>
<td>11</td>
</tr>
<tr>
<td>Write data 1</td>
<td>88</td>
<td>77 (5)</td>
<td>66 (5)</td>
<td>55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write into Flash Memory</th>
<th>0x2007</th>
<th>0x2006</th>
<th>0x2005</th>
<th>0x2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write data 1</td>
<td>FF</td>
<td>77 (5)</td>
<td>66 (5)</td>
<td>FF</td>
</tr>
<tr>
<td>Flash address</td>
<td>0x2003</td>
<td>0x2002</td>
<td>0x2001</td>
<td>0x2000</td>
</tr>
<tr>
<td>Write data 0</td>
<td>FF</td>
<td>33 (5)</td>
<td>22 (5)</td>
<td>FF</td>
</tr>
</tbody>
</table>

1.6.2. Memory Operations

During flash memory access, the IP performs the following steps to allow you to perform any direct read or write operation:

- Write enable for write operation
- Check flag status register to make sure the operation has been completed at the flash
- Release waitrequest signal when operation completed

Memory operations are Avalon memory-mapped operations. You must set the correct address on the address bus, write data if it is write transaction, drive burst count bus 1 if single transaction or desired burst count value and trigger the write or read signal.

Note: For multiple flash device setup, the address bus is extended to include the chip select value.

(5) New data written into the flash.
1. Generic Serial Flash Interface Intel® FPGA IP User Guide

1.6.3. Byte Enabling

1.6.3.1. Byte Enabling Supported Patterns

<table>
<thead>
<tr>
<th>Byte Enable Pattern</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>4'b0000</td>
<td>Supported (when burst is more than 1)</td>
</tr>
<tr>
<td>4'b0001</td>
<td>Supported</td>
</tr>
<tr>
<td>4'b0010</td>
<td>Supported</td>
</tr>
<tr>
<td>4'b0100</td>
<td>Supported</td>
</tr>
<tr>
<td>4'b1000</td>
<td>Supported</td>
</tr>
<tr>
<td>4'b0011</td>
<td>Supported</td>
</tr>
<tr>
<td>4'b0110</td>
<td>Supported</td>
</tr>
<tr>
<td>4'b1100</td>
<td>Supported</td>
</tr>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Byte Enable Pattern</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>4'b0111</td>
<td>Supported</td>
</tr>
<tr>
<td>4'b1110</td>
<td>Supported</td>
</tr>
<tr>
<td>4'b1111</td>
<td>Supported</td>
</tr>
</tbody>
</table>

All write bursts greater than 1 is set to byte enable of 4'b1111, in which all byte enablers are asserted through all the words of the burst. When a master wider than 32 bits is used to connect to the IP, the interconnect fabric of the Platform Designer produces multi-word bursts to adapt the wide master into the narrow 32-bit slave (the IP). Choose to use the byte enabling patterns in the Byte Enabling Supported Patterns table if the wide master intends to write only certain bytes in the entire transaction. You must ensure that the byte enabling pattern is contiguous for burst writes.

**Note:**
For burst writes, the IP writes all bytes (4'b1111) into the flash even with your selected byte enable pattern. If you have not enabled the data, the IP writes 0xFF. The performance of the IP is still the same as writing 8 bytes for a 64-bit wide master, even if you have enabled only 1 byte using byte enable.

### 1.6.4. Constraining the I/O Pins

The Intel Quartus Prime software does not automatically generate the I/O timing constraints for the Generic Serial Flash Interface Intel FPGA IP file. To enable the SPI pin interface using the general purpose I/O pins, you must manually enter the timing constraints. Follow the timing guidelines and examples to ensure that the Timing Analyzer analyzes the I/O timing correctly.

- **Constrain the input clock of the Generic Serial Flash Interface Intel FPGA IP.**
  **Example:**
  ```
  # Create Clock
  #******************************************************************************
  # constrain the base clock using create_clock, this is typically a clock coming into the device on an input clock pin
  # here, clk_clk is a 10ns clock with a 50 percent duty cycle, where the first rising edge occurs at 0ns applied to port clk_clk
  create_clock -name {clk_clk} -period 10.000 -waveform { 0.000 5.00 } [get_ports {clk_clk}]
  ```

- **Create a timing constraint to dclk, which is the SPI output clock from Generic Serial Flash Interface Intel FPGA IP.** The maximum SPI clock is half of the input clock. **Example:**
  ```
  #******************************************************************************
  # Create Generated Clock
  #******************************************************************************
  # constrain the generated clock dclk. The input clock of GSFI IP is used and created a counter logic to generate a slower DCLK that is used as SPI clock
  # here, we set the maximum dclk, which is half of the input clk_clk
  create_generated_clock -name {dclk_int} -source [get_ports {clk_clk}] -divide_by 2 [get_pins {u0|intel_generic_serial_flash_interface_top_0|intel_generic_serial_flash_interface_top_0|qspi_inf_inst|flash_clk_reg|q}]
  ```

---

Generic Serial Flash Interface Intel® FPGA IP User Guide

14
• Set a multicycle path to change the setup and hold clock relationship between the input clock and the SPI clock. Example:

```vhdl
#************************************************************
# Set Multicycle Path
#************************************************************
# For a divide by 2 DCLK
# SPI latched data on rising edge dclk and FPGA driven data output on rising edge clk_clk
set_multicycle_path -setup -start -from [get_clocks {clk_clk}] -to [get_clocks {dclk}] 2
set_multicycle_path -hold -start -from [get_clocks {clk_clk}] -to [get_clocks {dclk}] 1
# SPI driven data on falling edge of dclk and FPGA latched data on second rising edge of clk_clk
set_multicycle_path -setup -end -from [get_clocks {dclk}] -to [get_clocks {clk_clk}] 2
set_multicycle_path -hold -end -from [get_clocks {dclk}] -to [get_clocks {clk_clk}] 1
```

• Set the input and output delays for the quad serial peripheral interface (QSPI) IO pin. Example:

```vhdl
#************************************************************
# Set Input Delay
#************************************************************
#$input_delay is determined by Tco values and board parameters (outside of FPGA)
#$input_delay max = data_trace_max - clk_trace_min + ext_tco_max
#$input_delay min = data_trace_min - clk_trace_max + ext_tco_min
set_input_delay -clock { dclk } -max -clock_fall -add_delay $input_delay [get_ports {intel_generic_serialflash_interface_top_0_qspi_pins_data[0]}]
set_input_delay -clock { dclk } -min -clock_fall -add_delay $input_delay [get_ports {intel_generic_serialflash_interface_top_0_qspi_pins_data[0]}]
```

```vhdl
#************************************************************
# Set Output Delay
#************************************************************
#$output_delay is determined by Th and Tsu values and board parameters (outside of FPGA)
#$output_delay max = data_trace_max + Tsu - clk_trace_min
#$output_delay min = data_trace_min - Th - clk_trace_max
set_output_delay -clock { dclk } -max -add_delay $output_delay [get_ports {intel_generic_serialflash_interface_top_0_qspi_pins_data[0]}]
set_output_delay -clock { dclk } -min -add_delay $output_delay [get_ports {intel_generic_serialflash_interface_top_0_qspi_pins_data[0]}]
```

1.7. Generic Serial Flash Interface Intel FPGA IP Reference Design

The reference design implements the Generic Serial Flash Interface Intel FPGA IP to perform the following general-purpose memory operations:

- Read device ID
- Enable sector protect
- Perform sector erase
- Read and write data from and to flash devices

Related Information

- Generic Serial Flash Interface Intel FPGA IP User Guide on page 3
- Generic Serial Flash Interface Intel FPGA IP Core Reference Design Files

1.7.1. Hardware and Software Requirements

The following are the hardware and software requirements for the design example:

- Cyclone V E FPGA Development Kit
- Intel Quartus Prime Standard Edition software version 18.0 with Nios II Software Build Tools for Eclipse
- Intel FPGA Download Cable II
- Tested flash devices:
  - Cypress* S70FL01G
  - Micron MT25Q01G
  - Micron MT25Q512
  - EPCQ256
1.7.2. Functional Description

1.7.2.1. Reference Design Components

Table 7. Reference Design Components Descriptions

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JTAG UART Intel FPGA IP</td>
<td>Enables communication between the Nios II processor and the host computer.</td>
</tr>
<tr>
<td>Nios II Processor</td>
<td>Runs application program by executing data and instruction.</td>
</tr>
<tr>
<td>On-Chip Memory Intel FPGA IP</td>
<td>• Stores code and data.</td>
</tr>
<tr>
<td></td>
<td>• Connects the Nios II instruction master to the on-chip memory block.</td>
</tr>
<tr>
<td>Generic Serial Flash Interface Intel FPGA IP</td>
<td>Controls vendor-independent flash device to perform flash interaction.</td>
</tr>
</tbody>
</table>
1.7.2.2. Reference Design Application Program

Figure 6. Reference Design Application Program Flow Diagram

Flow diagram sequence description:

1. The application program starts with identifying the flash device attached to the FPGA.

   Note: The flash devices serve as samples to demonstrate this reference design only.

2. The application program performs sector protection and erases the protected sector:

   a. To perform sector protect, the application program:

      i. Performs write enable command.

      ii. Performs write status register command to set block protect (BP) bit and Top/Bottom(TB) bit.

      iii. Polls write in progress (WIP) bit (bit 0 of status register) until it returns a 0 (ready).

      iv. Performs read status register command to check if sector protect operation succeeded or failed.

   b. To perform sector erase, the application program:
i. Performs write enable command.
ii. Performs sector erase command.
iii. Polls in progress (WIP) bit (bit 0 of status register) until it returns a 0 (ready).
iv. Performs read status register to check whether erase operation succeeded or failed.

3. Erase error occurred because the sector is protected. The application program clears the error bit through:
   - Clear flag status register command (EPCQ-L or Micron).
   - Clear status register command (Cypress).

4. The application program disables the sector protect:
   a. Performs write enable command.
   b. Performs write status register command to clear BP bit and TB bit.
   c. Polls WIP bit (bit 0 of status register) until it returns a 0 (ready).
   d. Performs read status register command to check whether BP bit and TB bit has succeeded clear.

5. The application program performs flash device programming after the sector is not protected. The application program:
   a. Performs write memory into the address with empty memory.
   b. Polls WIP bit (bit 0 of status register) until it returns a 0 (ready).
   c. Performs read back memory of the address to confirm the address has programmed.

6. Repeat Step 2 and read back memory of the address. Memory is not erased because the sector is protected.

1.7.3. Creating Nios II Hardware System

1. In the Intel Quartus Prime software, go to File ➤ New Project Wizard.
2. Create a new Intel Quartus Prime Prime project named generic_flash_access in a new directory and select the Cyclone V E 5CEFA7F3117 device.
3. Select Tools ➤ Platform Designer, and save the file as generic_flash_access.qsys.
4. Double-click on the clock source clk_0 and change the Clock frequency to 100000000 Hz (100MHz).
5. Right click on clk_0 and rename it as sys_clk.
6. Add a Nios II processor:
   a. Go to Processor and Peripherals ➤ Embedded Processors ➤ Nios II Processor, and click Add.
   b. Click Finish to add the Nios II processor to the design and rename it as nios2.
      Note: Ignore any messages about parameters that have not been specified yet.
7. Add a Generic Serial Flash Interface IP:
a. Select Basic Functions ➤ Configuration and Programming ➤ Generic Serial Flash Interface Intel FPGA IP, and click Add. Rename this component as intel_generic_serial_flash_interface_top0.

b. Set the device density.
   
   Note: This reference design uses 1024MB flash device density.

c. Connect data_master of processor to avl_mem and avl_csr, and instruction_master of processor to only avl_mem of this component.

8. Add an On-chip Memory IP:
   a. Select Basic Functions ➤ On Chip Memory ➤ On-Chip Memory (RAM or ROM) Intel FPGA IP.
   b. Set the Total Memory Size to 40960 bytes (40 KBytes).
   c. Click Finish and rename as main_memory.
   d. Connect its slave to data_master and instruction_master of processor.

9. Add a JTAG UART IP:
   a. Go to Interface Protocols ➤ Serial ➤ JTAG UART Intel FPGA IP, and click Add.
   b. Click Finish and rename it as jtag_uart.
   c. Connect its avalon_jtag_slave port to the data_master port of the processor.
   d. In the IRQ column, connect the interrupt sender port from the Avalon_jtag_slave port to the interrupt receiver port of the processor and type 0.

10. Connect clock input of sys_clk to clock input of all other components.

11. Resolve all Nios II processor error messages before generating the Platform Designer system:
   a. Double click the Nios II processor nios2.
   b. Click Vectors, change both the Reset vector memory and Exception vector memory to main_memory.s1.
   c. Click System tab and click on the drop-down menu System and click Assign Base Address to auto assign base addresses for all the components.
   d. Under the same menu, click Create Global Reset Network to connect the reset signals to form a global reset network.
12. Generate the system:
   a. Click Generate HDL on the bottom of the window.
   b. When completed, the Platform Designer displays Generate: Completed successfully.

1.7.4. Integrating Modules into Intel Quartus Prime Project

1. In the Intel Quartus Prime software, select Assignment ➤ Settings.
2. In the Settings window, add generic_flash_access.qys file located in the synthesis folder and click Apply.
3. The generic_flash_access.qys file is shown under Files directory. Right click the file and choose Set as Top-Level Entity.
4. Go to Processing ➤ Start ➤ Start Analysis and Elaboration to allow the hardware system to determine input and output pins.
5. Start pin assignment by going to Assignments ➤ Pin Planner, and assign PIN_L14 as clk_clk and PIN_AA26 as reset_reset_n.
6. Go to Assignments ➤ Device ➤ Device and Pin Options ➤ Configuration, and change the Configuration scheme to Active Serial x1.
7. Processing ➤ Start ➤ Start Analysis and Synthesis to perform full hardware system compilation.

1.7.5. Programming the .sof File

1. In the Intel Quartus Prime Programmer, click on Hardware setup and choose the correct USB chain connecting your FPGA.
2. Click on Auto Detect and 5CEFA7F31 appears, and change the file to top.sof.
3. Enable Program/ Configure, and click Start.
1.7.6. Building Application Software System using Nios II Software Build Tools

1. In the Intel Quartus Prime, go to Tools ➤ Nios II Software Build Tools for Eclipse.
2. Browse to your workspace directory.
3. In the Nios II Software Build Tools for Eclipse, go to File ➤ New ➤ Nios II Application and BSP from Template.
4. In the SOPC Information File name field, select generic_flash_access.sopcinfo from your project directory and click Open.
5. For Project Name, set to generic_flash_access, choose Hello World Small project template and click Finish.
6. In the generic_flash_access project directory and replace the hello_world_small.c file with main.c and operation.c files attached in the reference design.
7. Select the main.c file and go to Project ➤ Build Project to create the generic_flash_access.elf file.
8. Select the generic_flash_access.elf file and go to Run ➤ Run As ➤ Nios II Hardware.
9. The Nios II Console prints the following results.

1.7.6.1. Reference Design Results

Cypress S70FL01G:

Flash Device: Cypress flash S70FL01G
Device ID: 4d210201
All sectors in this flash device is not protected
Now performing sector protection...
All sectors in this flash device is now successfully protected
Trying to erase sector 0...
ERASE ERROR as sector is protected!
Now perform sector unprotect...
Sector unprotect successfully! :)
Reading data at address 0...
Memory content at address 0: abcd1234
Trying to erase sector 0...
Sector erase successfully. Sector 0 is now empty.
Writing data to address 0...
Read back data from address 0...
Current memory in address 0: abcd1234
Read data match with data written. Write memory successful. Now performing sector protection...
All sectors in this flash device is now successfully protected
Trying to erase sector 0...
ERASE ERROR as sector is protected!
Current memory in address 0: abcd1234
Read data match with data written previously. Sector erase does not perform during sector is protected.

Micron MT25Q01G:

Flash Device: Micron flash MT25Q01G
Device ID: 1021ba20
All sectors in this flash device is not protected
Now performing sector protection...
All sectors in this flash device is now successfully protected
Trying to erase sector 0...
<table>
<thead>
<tr>
<th>Micron MT25Q512:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash Device: Micron flash MT25Q512</td>
</tr>
<tr>
<td>Device ID: 1020ba20</td>
</tr>
<tr>
<td>All sectors in this flash device is not protected</td>
</tr>
<tr>
<td>Now performing sector protection...</td>
</tr>
<tr>
<td>All sectors in this flash device is now successfully protected</td>
</tr>
<tr>
<td>Trying to erase sector 0...</td>
</tr>
<tr>
<td>Erase Error as erase is not allow during sector is protected!</td>
</tr>
<tr>
<td>Now perform sector unprotect...</td>
</tr>
<tr>
<td>Sector unprotect successfully! :)</td>
</tr>
<tr>
<td>Reading data at address 0...</td>
</tr>
<tr>
<td>Memory content at address 0: abcd1234</td>
</tr>
<tr>
<td>Address 0 containing data, it is not empty.</td>
</tr>
<tr>
<td>Trying to erase sector 0...</td>
</tr>
<tr>
<td>Sector erase successfully. Sector 0 is now empty.</td>
</tr>
<tr>
<td>Memory not containing data...</td>
</tr>
<tr>
<td>Writing data to address 0...</td>
</tr>
<tr>
<td>Read back data from address 0...</td>
</tr>
<tr>
<td>Current memory in address 0: abcd1234</td>
</tr>
<tr>
<td>Read data match with data written. Write memory successful.</td>
</tr>
<tr>
<td>Now performing sector protection...</td>
</tr>
<tr>
<td>All sectors in this flash device is now successfully protected</td>
</tr>
<tr>
<td>Trying to erase sector 0...</td>
</tr>
<tr>
<td>ERASE ERROR as sector is protected!</td>
</tr>
<tr>
<td>Current memory in address 0: abcd1234</td>
</tr>
<tr>
<td>Read data match with data written previously. Sector erase does not perform during sector is protected.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EPCQ256:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash Device: EPCQ256</td>
</tr>
<tr>
<td>Device ID: 1019ba20</td>
</tr>
<tr>
<td>All sectors in this flash device is not protected</td>
</tr>
<tr>
<td>Now performing sector protection...</td>
</tr>
<tr>
<td>All sectors in this flash device is now successfully protected</td>
</tr>
<tr>
<td>Trying to erase sector 0...</td>
</tr>
<tr>
<td>Erase Error as erase is not allow during sector is protected!</td>
</tr>
<tr>
<td>Now perform sector unprotect...</td>
</tr>
<tr>
<td>Sector unprotect successfully! :)</td>
</tr>
<tr>
<td>Reading data at address 0...</td>
</tr>
<tr>
<td>Memory content at address 0: abcd1234</td>
</tr>
<tr>
<td>Address 0 containing data, it is not empty.</td>
</tr>
<tr>
<td>Trying to erase sector 0...</td>
</tr>
<tr>
<td>Sector erase successfully. Sector 0 is now empty.</td>
</tr>
<tr>
<td>Memory not containing data...</td>
</tr>
<tr>
<td>Writing data to address 0...</td>
</tr>
<tr>
<td>Read back data from address 0...</td>
</tr>
<tr>
<td>Current memory in address 0: abcd1234</td>
</tr>
<tr>
<td>Read data match with data written. Write memory successful.</td>
</tr>
</tbody>
</table>
Now performing sector protection...
All sectors in this flash device is now successfully protected
Trying to erase sector 0...
ERASE ERROR as sector is protected!
Current memory in address 0: abcd1234
Read data match with data written previously. Sector erase does not perform during sector is protected.

1.8. Flash Access Using the Generic Serial Flash Interface Intel FPGA IP

This section provides information on how to use the registers of this IP to perform flash access. To begin, build the Platform Designer system with a few components (clock, jtag master, pll, and this IP) as shown below. Then, use the flash operations in the next example.

Figure 8. Example of Creating Flash Access Using the Generic Serial Flash Interface Intel FPGA IP

Note: You must set the MSEL pins of the FPGA devices to the AS configuration mode. For Intel MAX 10 devices, you must enable the Enable SPI Pins Interface parameter of this IP.

Flash operations are divided into several categories. Example of operations, registers to use, and sample .tcl scripts for each category are provided.

1.8.1. Flash Operations that Require Operation Code

The following flash operations require an operation code:
- Write enable
- Enter 4-byte addressing mode
- Exit 4-byte addressing mode
- Clear flag status register
- Clear status register

The following registers are used for operations that require an operation code:
- Flash command setting register
- Flash command control register
Example 1. Perform the Write Enable Operation for the Flash

```plaintext
class proc write_enable { } {
    global mp flash_cmd_setting flash_cmd_ctrl flash_cmd_write_data_0
    master_write_32 $mp $flash_cmd_setting 0x00000006
    master_write_32 $mp $flash_cmd_ctrl 0x1
}
```

To perform the write enable operation for the flash, follow these steps:
1. Define the global variables.
2. Customize the write enable operation by writing to the flash command setting register.
   a. Set bit \([7:0]\) of this register to \(06\) as \(06\)h is the operation code of the write enable operation.
3. Write 1 to bit 0 of the flash command control register to start the write enable operation.

1.8.2. Flash Operations to Read Flash Registers

The following flash operations are used to read flash registers:
- Read device ID
- Read status register
- Read flag status register
- Read configuration register
- Read bank register
- Read enhanced volatile configuration register

The following registers are used to read the status of a register:
- Flash command setting register
- Flash command control register
- Flash command read data 0 register

Example 2. Perform the Read Device ID Operation

```plaintext
class proc read_device_id { } {
    global mp flash_cmd_setting flash_cmd_ctrl flash_cmd_read_data_0
    master_write_32 $mp $flash_cmd_setting 0x0000489F
    master_write_32 $mp $flash_cmd_ctrl 0x1
    set device_id [master_read_32 $mp $flash_cmd_read_data_0 1]
    puts $device_id
}
```
To perform the read device ID operation, follow these steps:

1. Define the global variables.
2. Customize the read device ID operation by writing to the flash command setting register.
   a. Set bit [7:0] of this register to 9F as 9Fh is the operation code of the read device ID operation.
   b. Set bit [10:8] to 0 as this operation does not carry any address byte.
   c. Set bit 11 to 1 as the number of byte declared in bit [15:12] is the read data from the flash device.
   d. Set bit [15:12] to 4 as you will be reading 4 bytes device ID data from the flash.
3. Write 1 to bit 0 of the flash command control register to start the read device ID operation.
4. Read the device ID from the flash command read data 0 register.

1.8.3. Flash Operations to Write Flash Registers

The following flash operations are used to write flash registers:
- Write enhanced volatile configuration register
- Write bank register
- Write status register
- Write configuration register

*Note:* You must execute the write enable operation before you start these operations.

The following registers are used to write the status of a register:
- Flash command setting register
- Flash command control register
- Flash command write data 0 register

**Example 3. Perform the Write Status Register Operation to Protect Sector of Flash**

```
proc write_status_register { } {
    global mp flash_cmd_setting flash_cmd_write_data_0 flash_cmd_ctrl
    master_write_32 $mp $flash_cmd_setting 0x00001001
    master_write_32 $mp $flash_cmd_write_data_0 0x00000007c
    master_write_32 $mp $flash_cmd_ctrl 0x1
}
```

To perform the write status register operation, follow these steps:

1. Define the global variables.
2. Customize the write status register operation by writing to the flash command setting register.
a. Set bit [7:0] of this register to 01 as 01h is the operation code of the write status register operation.

b. Set bit [10:8] to 0 as this operation does not carry any address byte.

c. Set bit 11 to 0 as the number of byte declared in bit [15:12] is the write data to the flash device.

d. Set bit [15:12] to 1 as you will be writing 1 byte (8 bits) of data into the status register.

3. Write the data to set the sector protection into the flash command write data 0 register.

a. Bit 6 and bit [4:2] of the status register are the block protect bits and bit 5 is the Top/Bottom bit. In this example, protection is required for all sectors from the bottom of the memory array. For more information, refer to the respective flash datasheet.

4. Write 1 to bit 0 of the flash command control register to start the write status register for the sector protect operation.

1.8.4. Flash Operations that Require An Address

The following flash operations require an address:

- Sector erase
- Bulk erase
- Die erase

*Note:* You must execute the write enable operation before you start these operations.

The following registers are used for operations that require an address:

- Flash command setting register
- Flash command control register
- Flash command address register

**Example 4. Perform the Flash Sector Erase Operation**

```plaintext
proc erase_sector { } {
    global mp flash_cmd_setting flash_cmd_ctrl flash_cmd_addr_register
    master_write_32 $mp $flash_cmd_setting 0x000004D8
    master_write_32 $mp $flash_cmd_addr_register 0x00001000
    master_write_32 $mp $flash_cmd_ctrl 0x1
}
```

To perform the flash sector erase operation, follow these steps:

1. Define the global variables.

2. Customize the sector erase operation by writing to the flash command setting register.
a. Set bit \([7:0]\) of this register to D8 as D8h is the operation code of the sector erase operation.

b. Set bit \([10:8]\) to 4 as 4 bytes of address will be sent to the flash device.

c. Set bit 11 to 0 as the number of byte declared in bit \([15:12]\) is the write data to the flash device.

3. Specify any address within the sector that you want to erase and write it to the flash command address register.
   a. In this example, we are performing the erase sector operation for address 00001000.

4. Write 1 to bit 0 of the flash command control register to start the sector erase operation.

This IP core supports flash in the extended, dual, and quad I/O protocols. Currently, the protocols supported by this IP core is a single-transfer rate (STR) only. This IP core supports both the 3-byte and 4-byte addressing modes. Different protocols and addressing modes to read memory and program operations are explained in the following sections.

1.8.5. Read Memory from the Flash

The following registers are used to perform the read memory:

- Operating protocols setting register
- Control register
- Read instruction register

Example 5. **Perform the Read Memory (Extended Mode)**

```plaintext
proc read { } {
    global mp operating_protocols_setting control_register read_instr

    master_write_32 $mp $operating_protocols_setting 0x00000000
    master_write_32 $mp $control_register 0x00000001
    master_write_32 $mp $read_instr 0x00000003
    master_read_32 $mp 0x01000000 0x1
}
```

To perform the read memory for the extended mode, follow these steps:

1. Define the global variables.

2. Write to the operating protocols setting register to set the transfer mode of the read memory operation. In this example, the transfer mode for read is (1-1-1).
   a. Set the instruction transfer mode \([1:0]\) to 0, read address transfer mode \([13:12]\) to 0, and read data out transfer mode \([17:16]\) to 0.

3. Write to the control register to choose the byte addressing mode of the read memory operation.
Example 6. Perform the Dual-Output Fast Read (Dual-SPI Mode)

```bash
proc dual_output_fast_read { } {
    global mp operating_protocols_setting control_register read_instr
    master_write_32 $mp $operating_protocols_setting 0x00011001
    master_write_32 $mp $control_register 0x00000101
    master_write_32 $mp $read_instr 0x00000A3B
    master_read_32 $mp 0x00000100 0x1
}
```

To perform the dual-output fast read mode, follow these steps:

1. Define the global variables.
2. Write to the operating protocols setting register to set the transfer mode of the read memory operation. In this example, the transfer mode for read is (2-2-2).
   a. Set the instruction transfer mode [1:0] to 1, read address transfer mode [13:12] to 1, and read data out transfer mode [17:16] to 1.
3. Write to the control register to choose the byte addressing mode of the read memory operation.
   a. This example is using the 4-byte addressing mode. Set bit 8 to 1.
4. Write to the read instruction register to customize the read memory operation.
   a. Set the read operation code [7:0] to 3B as 3Bh is the operation code for the dual-output fast read.
   b. Set the dummy cycles [12:8] to A as the dual-output fast read operation contains 10 dummy cycles.
5. After setting the registers, you can perform dual-output fast read memory content in the address.
   a. In this example, the memory content is read from address 0x00000100.
1.8.6. Program Flash

The following registers are used to perform program flash:
- Operating protocols setting
- Control register
- Write instruction

Example 7. Perform Page Program (Extended Mode)

```bash
proc page_program { } {
    global mp operating_protocols_setting control_register write_instr
    master_write_32 $mp $operating_protocols_setting 0x00000000
    master_write_32 $mp $control_register 0x00000001
    master_write_32 $mp $write_instr 0x00007002
    master_write_32 $mp 0x00001000 0x1234abcd
}
```

To perform the page program for the extended mode, follow these steps:

1. Define the global variables.
2. Write to the operating protocols setting register to set the transfer mode of the program operation. In this example, the transfer mode for read is (1-1-1).
   a. Set the instruction transfer mode [1:0] to 1, write address transfer mode [5:4] to 1, and write data in transfer mode [9:8] to 1.
3. Write to the control register to choose the byte addressing mode of the write operation.
   a. This example is using the 3-byte addressing mode. Set bit 8 to 0.
4. Write to the write instruction register to customize the program operation.
   a. Set the write operation code [7:0] to 02 as 02h is the operation code for page program.
   b. Set the polling operation code [15:8] to 70 as 70h is the operation code for the read flag status register. After completing the write operation, the IP core releases the wait request of the Avalon memory-mapped interface. For the flash to have the read flag status register, you can use the read status register (05h).
5. After setting the registers, you can start to program the memory into the address.
   a. In this example, 1234abcdh is written to the memory address 0x00001000.
**Example 8. Perform 4-byte Quad Input Fast Program (Quad SPI Mode)**

```plaintext
proc fourbyte_quad_input_fast_program { } {
    global mp operating_protocols_setting control_register write_instr

    master_write_32 $mp $operating_protocols_setting 0x00000222
    master_write_32 $mp $control_register 0x00000101
    master_write_32 $mp $write_instr 0x00007034
    master_write_32 $mp 0x00002000 0xabcd1234
}
```

To perform the 4-byte quad input fast program, follow these steps:

1. Define the global variables.
2. Write to the operating protocols setting register to set the transfer mode of the program operation. In this example, the transfer mode for 4-byte quad input fast program is (4-4-4).
3. Write to the control register to choose the byte addressing mode of the write operation.
   a. This example is using the 4-byte addressing mode. Set bit 8 to 1.
4. Write to the write instruction register to customize the program operation.
   a. Set the write operation code [7:0] to 34 as 34h is the operation code for the 4-byte quad input fast program.
   b. Set the polling operation code [15:8] to 70 as 70h is the operation code for the read flag status register. After completing the write operation, the IP core releases the wait request of the Avalon memory-mapped interface. For the flash to have the read flag status register, you can use the read status register (05h).
5. After setting the registers, you can start to program the memory into the address.
   a. In this example, 1234abcdh is written to the memory address 0x00002000.

**1.9. Generic Serial Flash Interface Intel FPGA IP User Guide**

IP versions are the same as the Intel Quartus Prime Design Suite software versions up to v19.1. From Intel Quartus Prime Design Suite software version 19.2 or later, IP cores have a new IP versioning scheme.

<table>
<thead>
<tr>
<th>Intel Quartus Prime Version</th>
<th>IP Core Version</th>
<th>User Guide</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.1</td>
<td>19.2.1</td>
<td>Generic Serial Flash Interface Intel FPGA IP User Guide</td>
</tr>
<tr>
<td>19.4</td>
<td>19.1.1</td>
<td>Generic Serial Flash Interface Intel FPGA IP User Guide</td>
</tr>
</tbody>
</table>

**Archives**

If an IP core version is not listed, the user guide for the previous IP core version applies.
1.10. Document Revision History for the Generic Serial Flash Interface Intel FPGA IP User Guide

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>IP Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2020.09.28       | 20.3                        | 20.0.0     | • Added a new register setting—tSHSL.  
• Added a new section—Constraining the I/O Pins.  
• Updated the description for Enable flash simulation model in Table: Parameter Settings.  
• Removed Control Status Register Operations.  
• Updated the following topics:  
  — Generic Serial Flash Interface Intel FPGA IP User Guide  
  — Release Information  
  — Memory Operations  
• Updated the Hardware and Software Requirements of the Generic Serial Flash Interface Intel FPGA IP Reference Design section.  
• Updated the description of On-Chip Memory Intel FPGA IP in Table: Reference Design Components Descriptions.  
• Updated Creating Nios II Hardware System:  
  — Updated the description in step 7c.  
  — Updated Figure: Completed Platform Designer Connection.  
• Made minor editorial updates throughout the document. |
| 2020.05.08       | 20.1                        | 19.2.1     | • Added new sections—Release Information and Control Status Register Byte Enable.  
• Updated Table: Parameter Settings to include a new parameter—Use byteenable for CSR.  
• Added a new signal—avl_csr_byteenable.  
• Updated Figure: Signal Block Diagram.  
• Updated the note to the Device Family Support topic. |
• Added the Byte Enabling section.  
• Added a note to the Device Family Support topic.  
• Updated the Memory Operations topic.  
• Updated for latest branding standards. |
| 2019.11.27       | 19.3                        | 19.1       | Added a note to the Memory Operations topic. |
| 2019.09.30       | 19.3                        | 19.1       | • Added support for Intel Agilex devices.  
• Updated the Device Family Support topic.  
• Made minor editorial updates to the document. |

continued...
<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>IP Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2018.11.09       | 18.1                        | 18.1       | • Added the Flash Access Using the Generic Serial Flash Interface Intel FPGA IP Core section.  
• Added the Generic Serial Flash Interface Intel FPGA IP Core User Guide Archives section.  
• Updated the Generic Serial Flash Interface Intel FPGA IP Core User Guide section to provide more information on the Generic Serial Flash Interface Intel FPGA IP core.  
• Updated the signal names of the Signal Block Diagram figure.  
• Updated the Conduit Interface signal names in the Ports Description table.  
• Updated the description of the write opcode field name of the write instruction register in the Register Map table. |
| 2018.05.16       | 18.0                        | 18.0       | • Updated the Generic Serial Flash Interface Intel FPGA IP Core Reference Design Files link.  
• Added Flash Command Address Register in the Register Map. |
| 2018.05.07       | 18.0                        | 18.0       | Initial release. |