Generic Serial Flash Interface Intel® FPGA IP User Guide

Updated for Quartus® Prime Design Suite: **24.1**

IP Version: **20.2.5**

Online Version
Send Feedback

UG-20161

683419
2024.04.01

Contents

1. Generic Serial Flash Interface Intel® FPGA IP User Guide....3
1.1. Release Information....4
1.2. Device Family Support....4
1.3. Signals....6
1.4. Parameters....8
1.5. Register Map....9
1.6. Using Generic Serial Flash Interface Intel FPGA IP....12
1.6.1. Control Status Register Byte Enable....12
1.6.2. Memory Operations....14
1.6.3. Byte Enabling....15
1.6.4. Constraining the I/O Pins....16
1.7. Generic Serial Flash Interface Intel FPGA IP Reference Design....21
1.7.1. Hardware and Software Requirements....21
1.7.2. Functional Description....22
1.7.3. Creating Nios II Hardware System....24
1.7.4. Integrating Modules into Quartus Prime Project....26
1.7.5. Programming the .sof File....26
1.7.6. Building Application Software System using Nios II Software Build Tools....27
1.8. Flash Access Using the Generic Serial Flash Interface Intel FPGA IP....29
1.8.1. Flash Operations that Require Operation Code....31
1.8.2. Flash Operations to Read Flash Registers....31
1.8.3. Flash Operations to Write Flash Registers....32
1.8.4. Flash Operations that Require An Address....33
1.8.5. Read Memory from the Flash....34
1.8.6. Program Flash....36
1.9. Intel HAL Driver....39
1.9.1. Driver API....40
1.10. Generic Serial Flash Interface Intel FPGA IP User Guide Archives....43
1.11. Document Revision History for the Generic Serial Flash Interface Intel FPGA IP User Guide....43

683419 | 2024.04.01

Send Feedback

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 (Arria® 10 devices, 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 21
- Generic Serial Flash Interface Intel FPGA IP Core Reference Design Files
- KDB Answer: How do I enable Micron's MT25Q device support in replacement to End Of Life (EOL) EPCQ(>=256Mb) and EPCQ-L devices?
- Configuration Devices
  Provides more information on the third-party flash support.

(1) The supported flash devices for configuration memory are, EPCQ, EPCQ-A, EPCQ-L, and Micron* MT25Q (256Mb to 2Gb) devices.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

- Using the Generic Serial Flash Interface (ODEVGSFI) Training Course

1.1. Release Information

Intel FPGA IP versions match the Quartus® Prime Design Suite software versions until v19.1. Starting in Quartus Prime Design Suite software version 19.2, Intel FPGA IP has a new versioning scheme.

The Intel FPGA IP version (X.Y.Z) number can change with each Quartus Prime software version. A change in:

- X indicates a major revision of the IP. If you update the 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

Item	Description
IP Version	20.2.5
Quartus Prime Pro Edition Version	24.1
Release Date	2024.04.01

Item	Description
Quartus Prime Standard Edition Version	23.1
Release Date	2023.12.11

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

1.2. Device Family Support

The Generic Serial Flash Interface IP is supported in the following devices:

- Agilex™ 5[(2)][(3)]
- Agilex 7[(2)][(3)][(4)]
- Stratix® 10[(2)][(3)]

(2) Export the flash pin by enabling the `Enable SPI pins interface` parameter of this IP.

(3) The IP can only access flash that is connected to FPGA GPIO pins. You cannot use the IP to access flash that is connected to SDM for configuration purpose.

(4) For Agilex 7 F-Series and I-Series devices, do not place the exported conduit pins in the same x4 DQ group because they have different Output Enable (OE) signals. OE conflict during compilation can trigger Error(175005). Refer to the *Agilex 7 General Purpose I/O User Guide: F-Series and I-Series* and the KDB Answer *Error (175005): Could not find a location with: GPIO_SHARED_NOE0 of (locations affected)* for more information.

- Arria 10
- Cyclone 10 GX
- Cyclone 10 LP
- MAX® 10 (For general purpose memory only) (2)
- Stratix V
- Arria V
- Cyclone V
- Stratix IV
- Cyclone IV
- Arria II

Related Information

- Configuration Devices
  Provides more information about the third-party flash support.
- Agilex 7 General Purpose I/O User Guide: F-Series and I-Series
  More information about the I/O pin placement requirement and guidelines.
- KDB Answer: Error (175005): Could not find a location with: GPIO_SHARED_NOE0 of (locations affected)

1.3. Signals

Figure 1. Signal Block Diagram

The inclusion and width of some signals depend on the features selected.

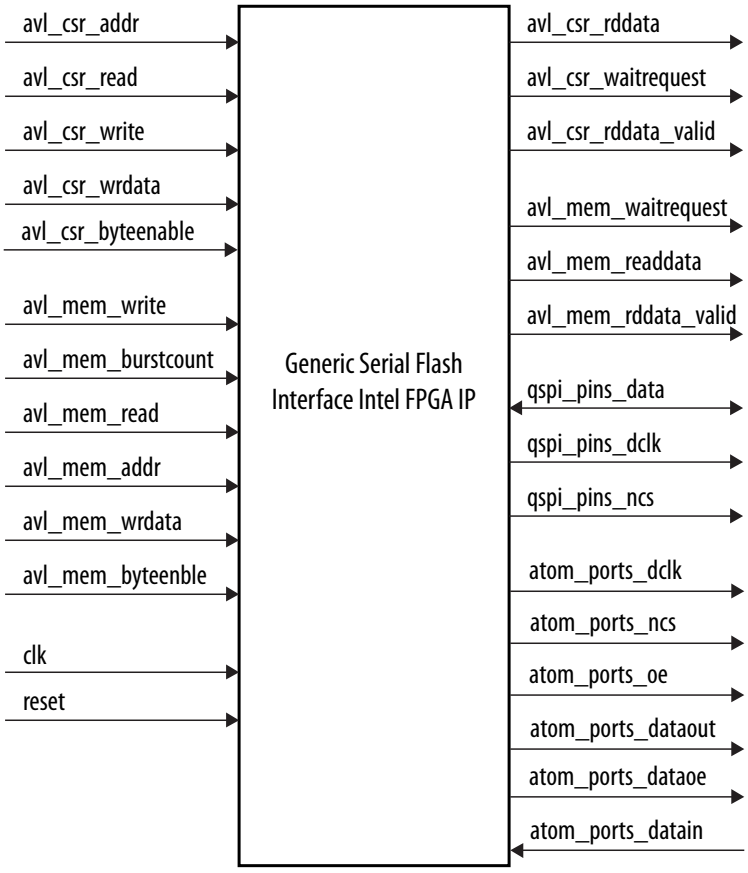


Table 2. Ports Description

Signal	Width	Direction	Description
Avalon Memory-Mapped Slave Interface for CSR (`avl_csr`) [5]			
`avl_csr_addr`	6	Input	Avalon memory-mapped address bus. The address bus is in word addressing.
`avl_csr_read`	1	Input	Avalon memory-mapped read control to the CSR.
`avl_csr_rddata`	32	Output	Avalon memory-mapped read data bus from the CSR.
`avl_csr_write`	1	Input	Avalon memory-mapped write control to the CSR.
`avl_csr_wrdata`	32	Input	Avalon memory-mapped write data bus to CSR.
`avl_csr_waitrequest`	1	Output	Avalon memory-mapped waitrequest control from the CSR.

continued...

(5) You can only access one port (`avl_csr` or `avl_mem`) at a time. The IP allows subsequent access after it deasserts the waitrequest signals (`avl_csr_waitrequest` and `avl_mem_waitrequest`).

Signal	Width	Direction	Description
avl_csr_rddata_valid	1	Output	Avalon memory-mapped read data valid that indicates the CSR read data is available.
avl_csr_byteenable	4	Input	Avalon memory-mapped byteenable control to the CSR. Available when you enable the **Use byteenable for CSR** parameter.
Avalon Memory-Mapped Slave Interface for Memory Access (avl_mem) [5]			
avl_mem_write	1	Input	Avalon memory-mapped write control to the memory.
avl_mem_burstcount	7	Input	Avalon memory-mapped burst count for the memory. The value range from 1 to 64 (maximum page size).
avl_mem_waitrequest	1	Output	Avalon memory-mapped waitrequest control from the memory.
avl_mem_read	1	Input	Avalon memory-mapped read control to the memory.
avl_mem_addr	N	Input	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 Arria 10, and 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.
avl_mem_wrdata	32	Input	Avalon memory-mapped write data bus to the memory.
avl_mem_readddata	32	Output	Avalon memory-mapped read data bus from the memory.
avl_mem_rddata_valid	1	Output	Avalon memory-mapped read data valid that indicates the memory read data is available.
avl_mem_byteenble	4	Input	Avalon memory-mapped write data enable bus to memory. During the bursting mode, the byteenable bus is at logic high, 4'b1111.
Clock and Reset			
clk	1	Input	Input clock to clock the IP.
reset	1	Input	Asynchronous reset to reset the IP.
Conduit Interface[6]			
qspi_pins_data	4	Bidirectional	Input or output port to feed data from the flash device.
qspi_pins_dclk	1	Output	Provides clock signal to the flash device.
qspi_pins_ncs	1/3	Output	Provides the ncs signal to the flash device.
ASMI Block Interface[7]			
atom_ports_dclk	1	Output	Clock signal from control-block based FPGA design to the external DCLK pin through the ASMI hard logic.
atom_ports_ncs	1/3	Output	Chip select signal from control-block based FPGA design to the external nCSO pin through the ASMI hard logic.
atom_ports_oe	1	Output	Active-low signal to enable DCLK and nCSO pins to reach the flash. The DCLK and nCSO are fixed to high when you set this signal to high, blocking the connection between control-block based FPGA and flash.

continued...

(6) Available when you enable the **Enable SPI pins interface** parameter.

(7) Available when you enable the **Disable dedicated Active Serial interface** parameter.

Signal	Width	Direction	Description
atom_ports_dataout	4	Output	Control signal from control-block based FPGA design to the Active Serial data pin for sending data into the serial configuration device.
atom_ports_dataoe	4	Output	Controls data pin either as input or output because the dedicated pins for active serial data is bidirectional.
atom_ports_datain	4	Input	Signal from the Active Serial data pin to control-block based FPGA design.

1.4. Parameters

Table 3. Parameter Settings - General Tab

Parameter	Legal Values	Descriptions
General Settings		
Device Density (Mb)	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048	Density of the flash device used in Mb.
Disable dedicated Active Serial interface	—	Routes the signals to the top level of your design. Enable this when you want to include the Serial Flash Loader Intel FPGA IP in your design.
Enable SPI pins interface	—	Translates the signals to the SPI pin interface.
Number of Chip Select used	1 2 3	Selects the number of chip select connected to the flash.
Default Settings		
Control Register	-	Expose the default settings for the Control Register. Refer to *Register Map* for the legal values and bit field descriptions.
Debug Mode		
Advance Mode	Disabled Enabled	Enable the **Advanced Mode** tab which expose the default settings for all registers.

Table 4. Parameter Settings - Simulation Tab

Parameter	Legal Values	Descriptions
Enable Simulation	-	Uses the default EPCQ1024 simulation model for simulation. When disabled, refer to AN-720: Simulating the ASMI Block in Your Design for creating a wrapper to use with other flash simulation model.

Note: When modifying the register settings in **Advanced Mode** tab, Refer to *Register Map* table for the legal values and parameter descriptions.

Related Information

- AN-720: Simulating the ASMI Block in Your Design
- Table 5 on page 9

1.5. Register Map

Table 5. 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.

Offset (Hex)	Register Name	R/W	Field Name	Bit	Default Value (Hex)	Description
0	Control Register	Reserved		31:9	Reserved	
		R/W	Addressing mode	8	0x0	Addressing mode for read and write operation: • 0x0: 3-bytes addressing. • 0x1: 4-bytes addressing. For 4-byte addressing mode, you must enable 4-byte address by sending command to the flash. This bit affects direct access to memory via the Avalon memory-mapped interface for both write and read operation.
		R/W	Chip select	7:4	0x0	Selects the flash device. • 0x0: To select first device. • 0x1: To select second device. • 0x2: To select third device.
		Reserved		3:1	Reserved	
		R/W	Enable	0	0x1	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.
1	SPI Clock Baud-rate Register	Reserved		31:5	Reserved	
		R/W	Baud rate divisor	4:0	0x10	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
2	CS Delay Setting Register	Reserved		31:12	Reserved	
		R/W	tSHSL (CS High Time)	11:8		This register setting controls the tSHSL. • 0: tSHSL is 3 IP_CLK. • n: tSHSL is 3+n IP_CLK.

continued...

Offset (Hex)	Register Name	R/W	Field Name	Bit	Default Value (Hex)	Description
		R/W	CS de-assert (CS Active Hold Time)	7:4	0x0	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.
		R/W	CS assert (CS Active Setup Time)	3:0	0x0	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. (8)
3	Read Capturing Register	Reserved		31:4	Reserved	
		R/W	Read delay	3:0	0x0	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. Delay the read data logic by a value of the IP_CLK cycles.
4	Operating Protocols Setting Register	Reserved		31:18	Reserved	
		R/W	Read data out transfer mode	17:16	0x0	Transfer mode for read data output.
		Reserved		15:14	Reserved	
		R/W	Read address transfer mode	13:12	0x0	Transfer mode for read address input Description as bit 1:0.
		Reserved		11:10	Reserved	
		R/W	Write Data in transfer mode	9:8	0x0	Transfer mode for write data input Description as bit 1:0.
		Reserved		7:6	Reserved	
		R/W	Write address transfer mode	5:4	0x0	Transfer mode for write address input Description as bit 1:0.
		Reserved		3:2	Reserved	
		R/W	Instruction transfer mode	1:0	0x0	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].

continued...

(8) Intel recommends that you set the chip select assertion delay to 5 if you are running the IP clock at 100 MHz.

Offset (Hex)	Register Name	R/W	Field Name	Bit	Default Value (Hex)	Description
						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.
5	Read Instruction Register	Reserved		31:13	Reserved	
		R/W	Dummy cycles	12:8	0x0	Number of default dummy cycles used for read operation. Refer to the respective flash device datasheet.
		R/W	Read opcode	7:0	0x03	The opcode for read operation. Refer to the respective flash device datasheet to select the correct opcode according to the transfer mode setting.
6	Write Instruction Register	Reserved		31:16	Reserved	
		R/W	Polling opcode	15:8	0x05	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.
		R/W	Write opcode	7:0	0x02	The opcode for write operation. Refer to the respective flash device datasheet to select the correct opcode according to the transfer mode setting.
7	Flash Command Setting Register(9)	Reserved		31:21	Reserved	
		R/W	Number of dummy cycles	20:16	0x0	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.
		R/W	Number of data bytes	15:12	0x08	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.
		R/W	Data type	11	0x01	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
		R/W	Number of address bytes	10:8	0x0	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.
		R/W	Opcode	7:0	0x05	The opcode of the operation.
8	Flash Command Control Register	Reserved		31:1	Reserved	
		W	Start	0	0x0	Write 1 to this bit to start the operation.

continued...

(9) Default setting is for read status command.

Offset (Hex)	Register Name	R/W	Field Name	Bit	Default Value (Hex)	Description
9	Flash Command Address Register	R/W	Stating address	31:0	31:0	Address of flash command.
A	Flash Command Write Data 0 Register	R/W	Lower 4 bytes write data	31:0	0x0	The first 4-byte of write data to flash device.
B	Flash Command Write Data 1 Register	R/W	Upper 4 bytes write data	31:0	0x0	The last 4-byte of write data to the flash device.
C	Flash Command Read Data 0 Register	R	Lower 4 bytes read data	31:0	0x0	The first 4-byte of read data from flash device.
D	Flash Command Read Data 1 Register	R	Upper 4 bytes read data	31:0	0x0	The last 4-byte of read data from the flash device.

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.

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_byteeanable` 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 writes 77663322 (4 bytes) into the flash address `0x2001`.

Table 6. Visualization of CSR byteenable for Write Data into Flash Memory

CSR Enabled Bytes in Write Data				
CSR byteenable for write data 1 and 0	0	1	1	0

continued...

Write data 0	44	33 (10)	22 (10)	11
Write data 1	88	77 (10)	66 (10)	55
Write into Flash Memory				
Flash address	0x2007	0x2006	0x2005	0x2004
Write data 1	FF	77 (10)	66 (10)	FF
Flash address	0x2003	0x2002	0x2001	0x2000
Write data 0	FF	33 (10)	22 (10)	FF

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.

Figure 2. 8-Word Write Burst Waveform Example

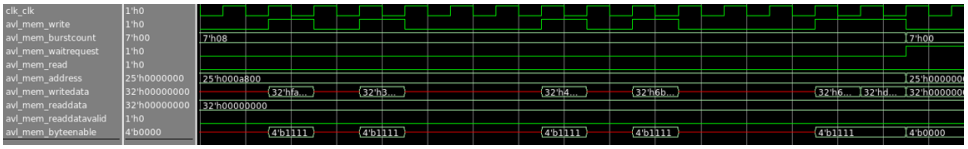
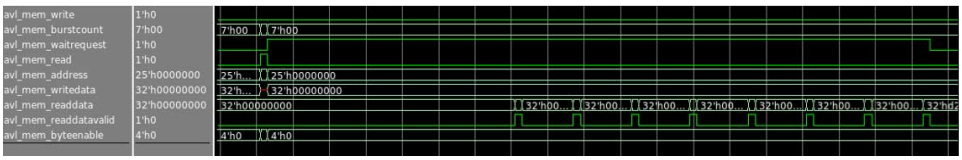
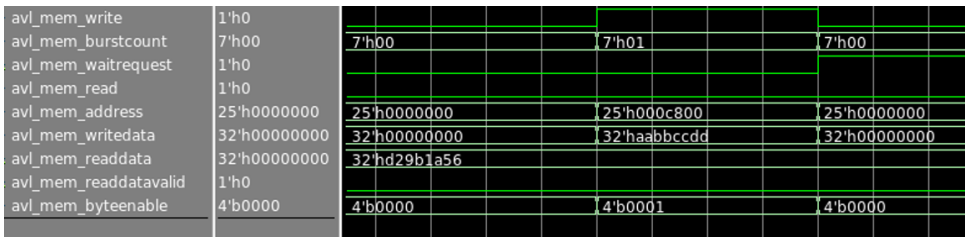


Figure 3. 8-Word Reading Burst Waveform Example



(10) New data written into the flash.

Figure 4. 1-Byte Write `byteenable = 4'b0001` Waveform Example



Note: There are two internal unconstrained clocks in the Generic Flash Serial Interface Intel FPGA IP core when you compile your design in the Quartus Prime Pro Edition software. Intel recommends that you constraint the path by using the following command:

```
create_generated_clock -name <name_of_generated_clock> -source [get_ports
<input_clock_name>] -divide_by 2 [get_registers <path_of_the_unconstrained_path>]
```

1.6.3. Byte Enabling

1.6.3.1. Byte Enabling Supported Patterns

Table 7. Byte Enabling Supported Patterns

Byte Enable Pattern	Support
4'b0000	Supported (when burst is more than 1)
4'b0001	Supported
4'b0010	Supported
4'b0100	Supported
4'b1000	Supported
4'b0011	Supported
4'b0110	Supported
4'b1100	Supported
4'b0111	Supported
4'b1110	Supported
4'b1111	Supported

All write bursts greater than 1 is set to byte enable of 4'b1111, in which all byte enables 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

1.6.4.1. General Purpose I/O

The 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
#refer to the GSFI UG, and you find that the maximum SPI clock baud-rate
divisor is 2.
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}]
create_generated_clock -name {dclk} -source [get_pins {u0|
intel_generic_serial_flash_interface_top_0|
intel_generic_serial_flash_interface_top_0|qspi_inf_inst|flash_clk_reg|q}]
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_dclk}]
```

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

  Note: The multicycle path is different in Quartus Prime Pro Edition and Quartus Prime Standard Edition.

  — In Quartus Prime Pro Edition:

```
#**************************************************************
# Set Multicycle Path
#**************************************************************
# For a divide by 2 DCLK
# SPI lactched 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
failing 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
```

— Quartus Prime Standard Edition:

```
#**************************************************************
# Set Multicycle Path
#**************************************************************
# For a divide by 2 DCLK
# SPI lactched 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}]  1
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:

```
#**************************************************************
# 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_max
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[0]}]
set_input_delay -clock { dclk } -min -clock_fall -add_delay $input_delay_min
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[0]}]
set_input_delay -clock { dclk } -max -clock_fall -add_delay $input_delay_max
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[1]}]
set_input_delay -clock { dclk } -min -clock_fall -add_delay $input_delay_min
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[1]}]
set_input_delay -clock { dclk } -max -clock_fall -add_delay $input_delay_max
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[2]}]
set_input_delay -clock { dclk } -min -clock_fall -add_delay $input_delay_min
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[2]}]
set_input_delay -clock { dclk } -max -clock_fall -add_delay $input_delay_max
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[3]}]
set_input_delay -clock { dclk } -min -clock_fall -add_delay $input_delay_min
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[3]}]
```

```
#**************************************************************
# 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_max
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[0]}]
set_output_delay -clock { dclk } -min -add_delay $output_delay_min
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[0]}]
set_output_delay -clock { dclk } -max -add_delay $output_delay_max
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[1]}]
set_output_delay -clock { dclk } -min -add_delay $output_delay_min
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[1]}]
set_output_delay -clock { dclk } -max -add_delay $output_delay_max
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[2]}]
set_output_delay -clock { dclk } -min -add_delay $output_delay_min
```

```
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[2]}]
set_output_delay -clock { dclk } -max -add_delay $output_delay_max
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[3]}]
set_output_delay -clock { dclk } -min -add_delay $output_delay_min
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_data[3]}]
set_output_delay -clock { dclk } -max -add_delay $output_delay_max
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_ncs}]
set_output_delay -clock { dclk } -min -add_delay $output_delay_min
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_ncs}]
```

1.6.4.2. Dedicated Active Serial Interface

In control-block based devices, the Generic Serial Flash Interface Intel FPGA IP can access the Active Serial configuration flash through the dedicated Active Serial interface.

The Quartus Prime software does not automatically generate the I/O timing constraints for the Generic Serial Flash Interface Intel FPGA IP file. When using the dedicated Active Serial Interface pins, you may enter the timing constraints to meet timing requirements with better effectiveness. Follow the timing guidelines and examples to ensure that the Timing Analyzer analyzes the I/O timing correctly.

To render the dedicated Active Serial Interface visible, the Intel Quartus INI variable (`fiomgr_enable_spi_timing=on`) is required.

- 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
#refer to the GSFI UG, and you find that the maximum SPI clock baud-rate
divisor is 2.
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}]
create_generated_clock -name {dclk} -source [get_pins {u0|
intel_generic_serial_flash_interface_top_0|
intel_generic_serial_flash_interface_top_0|qspi_inf_inst|flash_clk_reg|q}]
[get_ports {intel_generic_serial_flash_interface_top_0_qspi_pins_dclk}]
```

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

  Note: The multicycle path is different in Quartus Prime Pro Edition and Quartus Prime Standard Edition.

  — In Quartus Prime Pro Edition:

```
#**************************************************************
# Set Multicycle Path
#**************************************************************
# For a divide by 2 DCLK
# SPI lactched 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
failing 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
```

— Quartus Prime Standard Edition:

```
#**************************************************************
# Set Multicycle Path
#**************************************************************
# For a divide by 2 DCLK
# SPI lactched 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}]  1
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:

```
#**************************************************************
# 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_max
[get_ports {AS_DATA*}]
set_input_delay -clock { dclk } -min -clock_fall -add_delay $input_delay_min
[get_ports {AS_DATA*}]
```

```
#**************************************************************
# 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_max
[get_ports {AS_DATA*}]
set_output_delay -clock { dclk } -min -add_delay $output_delay_min
[get_ports {AS_DATA*}]
set_output_delay -clock { dclk } -max -add_delay $output_delay_max
[get_ports {NCSO0*}]
set_output_delay -clock { dclk } -min -add_delay $output_delay_min
[get_ports { NCSO0*}]

# Specify the maximum allowable skew between AS_DATA registers
#$max_skew is recommended at 10ns, and tighten it as needed by your design.
set_max_skew -to [get_ports {AS_DATA*}] $max_skew
```

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
- 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

Figure 5. Reference Design Block Diagram

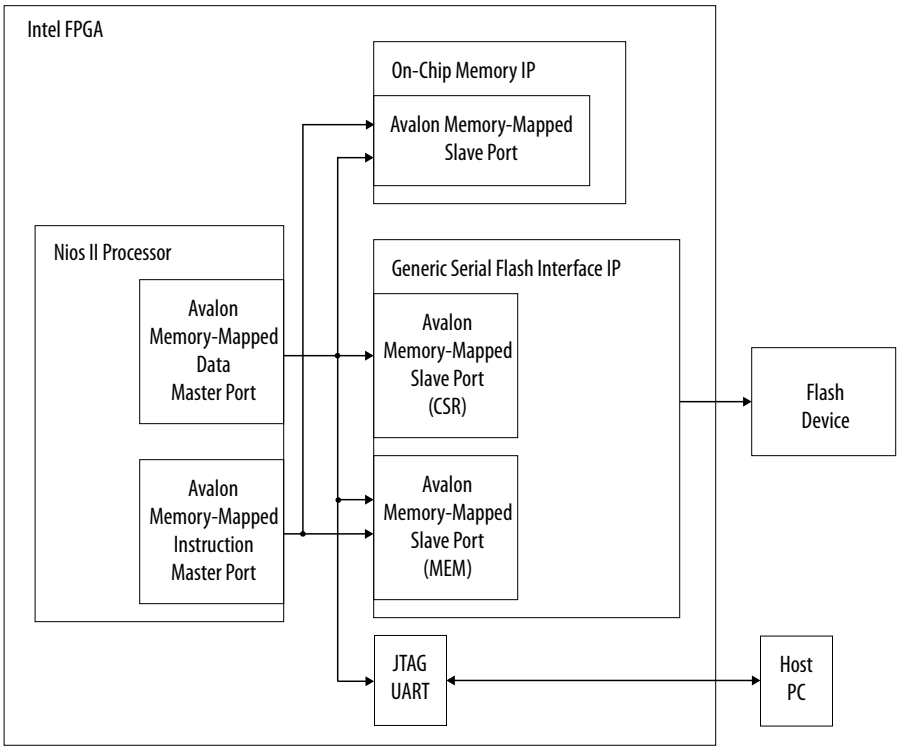
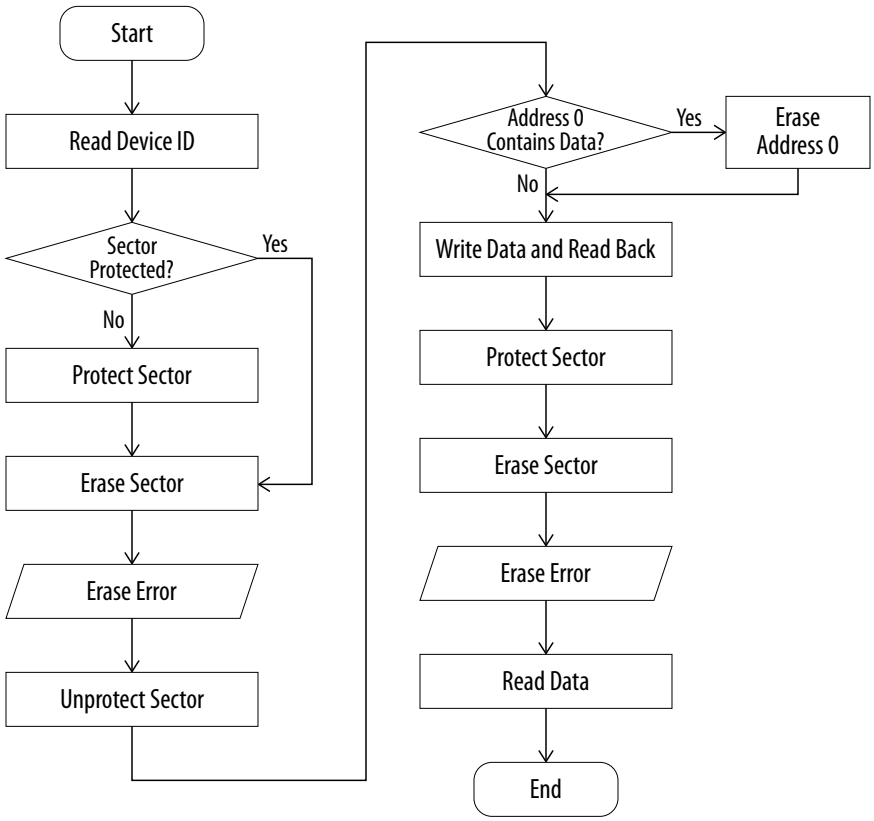


Table 8. Reference Design Components Descriptions

Component	Description
JTAG UART Intel FPGA IP	Enables communication between the Nios II processor and the host computer.
Nios II Processor	Runs application program by executing data and instruction.
On-Chip Memory Intel FPGA IP	• Stores code and data. • Connects the Nios II instruction master to the on-chip memory block.
Generic Serial Flash Interface Intel FPGA IP	Controls vendor-independent flash device to perform flash interaction.

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 write in progress (WIP) bit (bit 0 of status register) until it return 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 Quartus Prime software, go to **File ➤ New Project Wizard**.
2. Create a new 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:

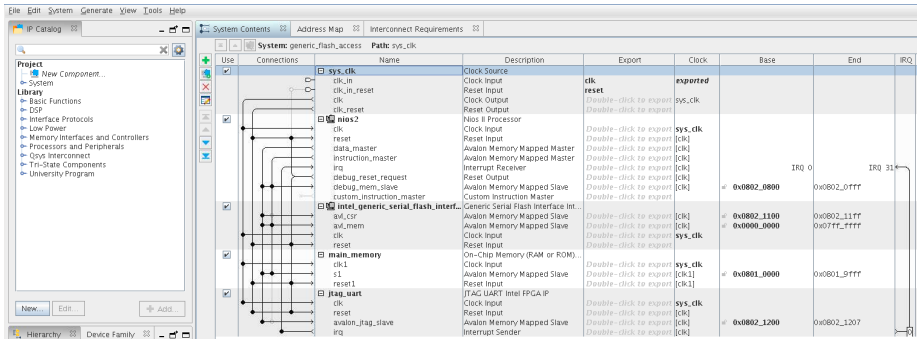
a. Double click the Nios II processor `nios2`.

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.

Figure 7. Completed Platform Designer Connection



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 Quartus Prime Project

1. In the 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 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 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...
Erase Error as erase is not allow during sector is protected!
Now perform sector unprotect...
```

```
Sector unprotect successfully! :)
Reading data at address 0...
Memory content at address 0: abcd1234
Address 0 containing data, it is not empty.
Trying to erase sector 0...
Sector erase successfully. Sector 0 is now empty.
Memory not containing data...
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 MT25Q512:

```
Flash Device: Micron flash MT25Q512
Device ID: 1020ba20
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 erase is not allow during sector is protected!
Now perform sector unprotect...
Sector unprotect successfully! :)
Reading data at address 0...
Memory content at address 0: abcd1234
Address 0 containing data, it is not empty.
Trying to erase sector 0...
Sector erase successfully. Sector 0 is now empty.
Memory not containing data...
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.
```

EPCQ256:

```
Flash Device: EPCQ256
Device ID: 1019ba20
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 erase is not allow during sector is protected!
Now perform sector unprotect...
Sector unprotect successfully! :)
Reading data at address 0...
Memory content at address 0: abcd1234
Address 0 containing data, it is not empty.
Trying to erase sector 0...
Sector erase successfully. Sector 0 is now empty.
Memory not containing data...
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.
```

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

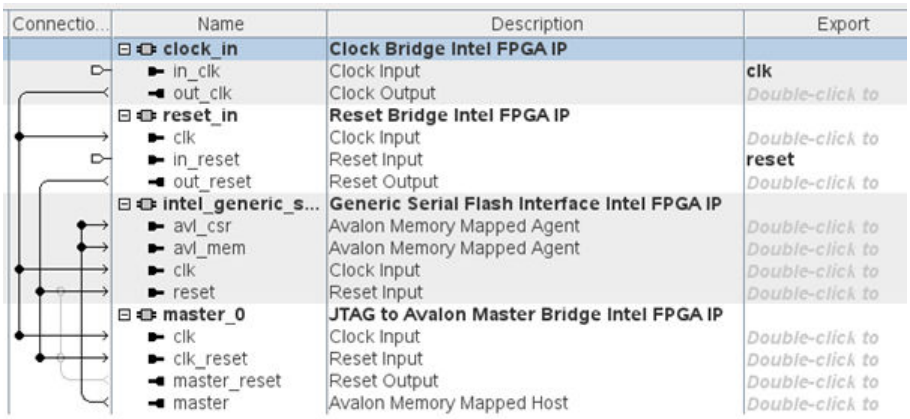
This section provides information to use the Generic Serial Flash Interface (GSFI) Intel FPGA IP registers (Table 4. Register Map) to perform flash access. This example design performs flash access to the Active Serial (AS) configuration flash in control block-based devices. To begin, you must build the Platform Designer system as shown in Figure 8. Example of Creating Flash Access Using the Generic Serial Flash Interface Intel FPGA IP.

Key Components:

- Clock and Reset Bridge Intel FPGA IP
- Generic Serial Flash Interface Intel FPGA IP
- JTAG to Avalon Master Bridge Intel FPGA IP

For more information about other components, refer to the *Embedded Peripherals IP User Guide*.

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



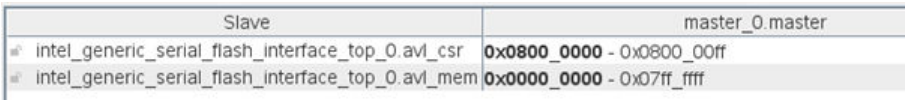
Note: To access the AS configuration flash, set the MSEL pins of the FPGA devices to the AS configuration mode. To access the general purpose QSPI flash, enable the **Disable Dedicated Active Serial Interface** and **Enable SPI Pins Interface** parameter of this IP.

For SDM-based devices, use the *Mailbox Client Intel FPGA IP* to access the AS configuration flash.

For MAX 10 devices, enable the **Disable Dedicated Active Serial Interface** and **Enable SPI Pins Interface** parameter of this IP to access the general purpose QSPI flash.

As per the Address Map shown in the figure below, the GSFI Intel FPGA IP register map spans within the address 0x8000000 to 0x80000ff, whereas the flash memory (1024 Mb) spans from 0x0 to 0x7ffffff. These addresses are crucial to access the correct register or flash memory space.

Figure 9. Address Map



Slave	master_0.master
intel_generic_serial_flash_interface_top_0.avl_csr	**0x0800_0000** - 0x0800_00ff
intel_generic_serial_flash_interface_top_0.avl_mem	**0x0000_0000** - 0x07ff_ffff

In this example, System Console is used to access the GSFI Intel FPGA IP and flash memory. The System Console utilizes the JTAG to Avalon Master Bridge Intel FPGA IP along with a TCL script.

Example : Sample `.tcl` script

```
#set GSFI CSR base address and register map according to Platform Designer
#system
set base 0x8000000
set control_register [expr {$base + 0x0}]
set spi_clock_baud_rate_register [expr {$base + 0x4}]
set cs_delay_setting_register [expr {$base + 0x8}]
set read_capturing_register [expr {$base + 0xc}]
set operating_protocols_setting [expr {$base + 0x10}]
set read_instr [expr {$base + 0x14}]
set write_instr [expr {$base + 0x18}]
set flash_cmd_setting [expr {$base + 0x1c}]
set flash_cmd_ctrl [expr {$base + 0x20}]
set flash_cmd_addr_register [expr {$base + 0x24}]
set flash_cmd_write_data_0 [expr {$base + 0x28}]
set flash_cmd_write_data_1 [expr {$base + 0x2c}]
set flash_cmd_read_data_0 [expr {$base + 0x30}]
set flash_cmd_read_data_1 [expr {$base + 0x34}]

#claims JTAG to Avalon Master Bridge service
set mp [claim_service master [lindex [get_service_paths master] 0] top]

#print the value of Control Register
set reg [master_read_32 $mp $control_register 0x1]
puts "Control Register : $reg"

#you may perform the flash operation here

#close JTAG to Avalon Master Bridge service
close_service master $mp
```

For more information on System Console services and commands, refer to the *Analyzing and Debugging Designs with System Console*.

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

Related Information

- Embedded Peripheral IP User Guide
  Provides information about other components.
- Mailbox Client IP User Guide
  Provides information about access to the AS configuration flash for SDM-based devices.
- Analyzing and Debugging Designs with System Console
  Provides information about system console services and commands.

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

```
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 06h 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

```
proc read_device_id { } {
global mp flash_cmd_setting flash_cmd_ctrl flash_cmd_read_data_0
master_write_32 $mp $flash_cmd_setting 0x000048 9F
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` so that you can read 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 0x0000007c
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 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

```
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 is 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)

```
proc read { } {
global mp operating_protocols_setting control_register read_instr
master_write_32 $mp $operating_protocols_setting 0x000000000
master_write_32 $mp $control_register 0x00000001
master_write_32 $mp $read_instr 0x00000003
master_read_32 $mp 0x0100000 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.
   a. This example is using the 3-byte addressing mode. Set bit 8 to 0.
4. Write to the read instruction register to customize the read memory operation.
   a. Set the read operation code [7:0] to 03 as 03h is the operation code for read.
   b. Set the dummy cycles [12:8] to 0 as the read operation does not contain any dummy cycles.
5. After setting the registers, you can perform read memory content in the address.
   a. In this example, 1 word of data is read from address 0x01000000.

Example 6. Perform the Dual-Output Fast Read (Dual-SPI Mode)

```
proc dual_output_fast_read { } {
global mp operating_protocols_setting control_register read_instr
master_write_32 $mp $operating_protocols_setting 0x000110001
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)

```
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 0, write address transfer mode [5:4] to 0, and write data in transfer mode [9:8] to 0.

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 that does not 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)

```
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).
   a. Set the instruction transfer mode [1:0] to 2, write address transfer mode [5:4] to 2, and write data in transfer mode [9:8] to 2.
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 that does not 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. Intel HAL Driver

The HAL API is available for this controller in the following software files:

- `intel_generic_serial_flash_interface_top.h`
- `intel_generic_serial_flash_interface_top.c`

These files implement the Generic Serial Flash Interface core device driver for the HAL system library.

Intel HAL supports a number of generic device model classes including one for device flashes. Developing against these generic classes gives a consistent interface for driver functions so that the HAL can access the driver functions uniformly.

A HAL API application begins by calling `alt_flash_open_dev()` to open the flash device, which returns a file handle to a flash device. You can obtain the IP name from `system.h`. All offset-related variables are based on absolute addressing to the flash memory.

HAL API supports the following QSPI flash device families:

- Micron (MT25QL and MT25QU variants)
- Cypress (S25FL-S and S25FS-S variants)
- Macronix (MX25L and MX66U variants)
- Winbond (W25Q variant)
- GigaDevice (GD55LB variant)
- ISSI (I25LP and I25WP variants)

Note: The Intel HAL Driver is supported in Quartus Prime Pro Edition software only.

Note: Enable `altera_safelib` in **BSP Software Package** from **BSP Editor** to use the HAL API for Quartus Prime Pro Edition software version prior to 21.4.

Note: `Flash_Timeout` is configurable in the Board Support Package Editor with the default value of 700000 microseconds (0.7 seconds).

Related Information

- Nios II Software Developer Handbook
  For more information, refer to Using Flash Devices.
- Nios II Configuration and Booting Solutions
  For more information about booting Nios II processor from Flash.
- Example of HAL API Application
  A C source code that implements all the Generic Serial Flash Interface Intel HAL driver.
- Nios V Configuration and Booting Solutions
  For more information about booting Nios V processor from Flash

1.9.1. Driver API

Table 9. intel_gsfi_get_info

Prototype:	`intel_gsfi_get_info (alt_flash_fd *fd, flash_region **info, int *number_of_regions`
Include:	`<intel_generic_serial_flash_interface_top.h>`
Parameter:	• fd – pointer to the flash device structure • info – pointer to the flash region • number_of_regions - pointer to the number of regions
Return:	Return 0 or otherwise return • EINVAL for Invalid argument • EIO for possible hardware problem
Description:	Provide information about the flash devices. Returns the flash memory offset, flash memory size, number of flash device, number of sectors, and sector size values.

Table 10. intel_gsfi_read

Prototype:	`intel_gsfi_read (alt_flash_dev *flash_info, int offset, void *dest_addr, int length)`
Include:	`<intel_generic_serial_flash_interface_top.h>`
Parameter:	• flash_info – pointer to flash device structure • offset – flash address • dest_addr – destination buffer • length – size of read data
Return:	Return 0 for success and otherwise return: • EINVAL for invalid argument • ENODEV for no device detected
Description:	Read data from selected address. This function copies data in the flash and passes it to the destination buffer.

Table 11. intel_gsfi_erase_block

Prototype:	`intel_gsfi_erase_block (alt_flash_dev *flash_info, int block_offset)`
Include:	`<intel_generic_serial_flash_interface_top.h>`
Parameter:	• flash_info – pointer to the flash device structure • block_offset – byte-address offset, from the start of flash of the sector to be erased
Return:	Return 0 for success and otherwise return: • EINVAD for invalid argument • ENODEV for no device detected • EIO for erase failed and sector might be protected • ETIME for timeout after `Flash_Timeout`
Description:	Erase a single flash sector.

Table 12. intel_gsfi_write_block

Prototype:	`intel_gsfi_write_block (alt_flash_dev *flash_info, int block_offset, int data_offset, const void *data, int length)`
Include:	`<intel_generic_serial_flash_interface_top.h>`

continued...

Parameter:	• flash_info – pointer to the flash device structure • block_offset – byte-address offset, from the start of flash of the sector to be written • data_offset – byte offset (unaligned access) of write into memory • data – data to be written • length – size of writing
Return:	Return 0 for success and otherwise: • EINVAL for invalid argument • ENODEV for no device detected • EIO for write failed and sector might be protected • ETIME for timeout after `Flash_Timeout`
Description:	Write one block/sector of data to the flash. The length of the write cannot spill into the adjacent sector, the function assumes the address is empty, otherwise you should erase it first.

Table 13. intel_gsfi_write

Prototype:	`intel_gsfi_write (alt_flash_dev *flash_info, int offset, const void *src_addr, int length)`
Include:	`<intel_generic_serial_flash_interface_top.h>`
Parameter:	• flash_info – pointer to the flash device structure • offset – flash address (unaligned access) of write to flash memory • src_addr – source buffer • length – size of writing
Return:	Return 0 for success and otherwise: • EINVAL for invalid argument • ENODEV for no device detected • EIO for write failed and sector might be protected • ETIME for timeout after `Flash_Timeout`
Description:	Program the data into the flash at the selected address. This function automatically erases the block as needed.

Table 14. intel_gsfi_lock

Prototype:	`intel_gsfi_lock (alt_flash_dev *flash_info, alt_u32 sectors_to_lock)`
Include:	`<intel_generic_serial_flash_interface_top.h>`

Parameter:	• flash_info – pointer to the flash device structure
	• sectors_to_lock – block protection settings
	  — Bit 4 : Top/Bottom bit (TB) or Block Protection bit 4 (BP4)
	  — Bit 3 : BP3
	  — Bit 2 : BP2
	  — Bit 1 : BP1
	  — Bit 0 : BP0

Return:	Return 0 for success and otherwise: • EINVAL for invalid argument • ENODEV for no device detected • ENOLCK for One-Time Programmable (OTP) TB bit write failed • EIO for sector protection failed • ETIME for timeout after `Flash_Timeout`
Description:	Perform block protection on a range of the flash sectors. The protected flash sectors cannot be written or erased. Refer to the flash device datasheet for the protected area definitions.

Table 15. intel_gsfi_chip_select

Prototype:	`intel_gsfi_chip_select (intel_gsfi_dev *flash_info, int flash_device)`
Include:	`<intel_generic_serial_flash_interface_top.h>`
Parameter:	• flash_info – pointer to the flash device structure • flash_device - flash device selection — 0x0 : First device — 0x1 : Second device — 0x2 : Third device
Return:	Return 0 for success and otherwise: • EINVAL for invalid argument • ENODEV for no device detected
Description:	Select the flash device.

Table 16. intel_gsfi_config

Prototype:	`intel_gsfi_config (intel_gsfi_dev *flash_info, gsfi_config *setting)`
Include:	`<intel_generic_serial_flash_interface_top.h>`
Parameter:	• flash_info – pointer to the flash device structure • setting - pointer to the configuration structure
Return:	Return 0 for success and otherwise: • EINVAL for invalid argument • ENODEV for no device detected
Description:	Configure: • tSHSL (CS high time) value(11) • CS de-assert (CS active hold time) value(11) • CS assert (CS active setup time) value(11) • Baud rate divisor value(11) • Flash to support Quad I/O mode(12)

(11) Refer to the respective registers in 1.5 Register Map for the input range.

(12) Enable this feature by asserting to 1, otherwise disable with 0.

1.10. Generic Serial Flash Interface Intel FPGA IP User Guide Archives

For the latest and previous versions of this user guide, refer to Generic Serial Flash Interface Intel® FPGA IP User Guide. If an IP or software version is not listed, the user guide for the previous IP or software version applies.

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

1.11. Document Revision History for the Generic Serial Flash Interface Intel FPGA IP User Guide

Document Version	Quartus Prime Version	IP Version	Changes
2024.04.01	24.1	20.2.5	• Updated table *Generic Serial Flash Interface Intel FPGA IP Release Information*. • Added Agilex 5 to the *Device Family Support*.
2023.11.09	23.1	20.2.3	• Added the following subtopics under *Constraining the I/O Pins* section: — *General Purpose I/O* — *Dedicated Active Serial Interface*
2023.10.11	23.1	20.2.3	• Edited table *Ports Description*: — Removed Interrupt. — Updated Conduit Interface. — Added ASMI Block Interface.
2023.04.10	23.1	20.2.3	• Updated product family name to "Intel Agilex® 7". • Edited *Device Family Support* to replace Intel Agilex with "Intel Agilex 7" devices and added clarity to the footnotes. • Edited the footnotes in the table *Ports Description* to improve clarity.
2022.12.22	22.2	20.2.1	• Added a note in Table: *Ports Description* to state that only one port (`avl_csr` or `avl_mem`) can be accessed at a time. • Made minor editorial edits.
2022.07.25	22.2	20.2.1	• Updated *Intel HAL Driver* topic: — Added a note for `Flash_Timeout`. — Added a note to state support for Quartus Prime Pro Edition software only. • Added ETIME error code in intel_gsfi_erase_block, intel_gsfi_write_block, intel_gsfi_write, intel_gsfi_lock in *Driver API* topic.
2022.06.23	22.2	20.2.1	• Updated *Parameter Settings - General tab* in *Parameters* section. • Added *Parameter Settings - Simulation tab* table in *Parameters* section. • Updated title *Nios II HAL Driver* to *Intel HAL Driver*. • Updated instances of Nios II to Intel in *Intel HAL Driver* section. • Added support for QSPI flash device in *Intel HAL Driver* section.

continued...

Document Version	Quartus Prime Version	IP Version	Changes
			• Added note in *Intel HAL Driver* section. • Added related information for Nios V Configuration and Booting Solutions in *Intel HAL Driver* section. • Updated Prototype, Return, Description in intel_gsfi_get_info, intel_gsfi_read, intel_gsfi_erase_block, intel_gsfi_write_block, intel_gsfi_write in *Intel HAL Driver* section. • Added intel_gsfi_lock, intel_gsfi_chip_select, and intel_gsfi_config tables in *Intel HAL Driver* section.
2022.04.20	22.1	20.1.1	Corrected the Reserved Bit value for the following registers in Table: *Register Map*: • From 31:8 to 31:9 for *Control Register*. • From 31:14 to 31:13 for *Read Instruction Register*.
2022.04.07	22.1	20.1.1	Updated description for *Flash Access Using the Generic Serial Flash Interface Intel FPGA IP* section with additional information.
2021.11.09	21.2	20.1.1	Updated *CS Delay Setting Register* in *Register Map* topic.
2021.06.21	21.2	20.1.1	• Updated *Device Family Support*. • Updated *Nios II HAL Driver*: — Updated the `intel_gsfi_get_info` description in Table: *intel_gsfi_get_info*. — Updated the `intel_gsfi_read` description in Table: *intel_gsfi_read*. — *Nios II HAL Driver* • Removed the *Driver API Application* topic. • Updated Figure: *Reference Design Block Diagram*.
2021.03.29	21.1	20.1.1	• Added the following sections: — *Nios II HAL Drivers* — *Driver API* — *Driver API Application* • Corrected the default value for dummy cycles from 0xA to 0x0 in Table: *Register Map*. • Updated the steps in the Perform Page Program (Extended Mode) and Perform 4-byte Quad Input Fast Program (Quad SPI Mode) examples in *Program Flash*.
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.

continued...

Document Version	Quartus Prime Version	IP Version	Changes
			• 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 through out 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.
2020.04.13	19.4	19.1.1	• Renamed document title as Generic Serial Flash Interface Intel FPGA IP User Guide • 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.
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.