Gisselquist
Technology, LLC

QUAD SPI FLASH CONTROLLER SPECIFICATION

Dan Gisselquist, Ph.D.
zipcpu (at) gmail.com

January 23, 2021

Revision History

Rev.	Date	Author	Description
1.0	11/22/2018	Gisselquist	Completely updated into a more universal set of controllers
0.3	8/11/2016	Gisselquist	Added information on the Extended Quad SPI controller
0.2	5/26/2015	Gisselquist	Minor spelling changes
0.1	5/13/2015	Gisselquist	First Draft

Contents

Figures

Tables

Preface

The genesis of this project was a desire to communicate with and program an FPGA board without the need for any proprietary tools. This includes Xilinx JTAG cables, or other proprietary loading capabilities such as Digilent's Adept program. As a result, all interactions with the board need to take place using open source tools, and the board must be able to reprogram itself.

After several iterations of this controller, it has finally been rebuilt into a generic controller that will provide a Wishbone interface for any flash chip.

Dan Gisselquist, Ph.D.

1. Introduction

This repository of serial flash controllers has now gone through several iterations of development.

It was initially built to support a fairly generic Spansion flash part with QSPI mode. This initial interface supported four control registers, for erasing the flash, configuring the flash, reading from the status register, and reading the manufacturer's ID. This original core worked wonderfully well beyond the flash part it was initially targetted for, so the core got wide usage.

A second core was rebuilt and added to the repository to support a Micron flash, with the purpose of running at twice the clock rate (200MHz system clock, 100MHz SPI clock). Instead of four registers, this core supported roughly 16 registers. This core was only ever used on the Digilent Arty board, since it had way too many features for general purpose use and since it had a horrible latency.

A third core was built for the S6SoC project. This third core was designed for the absolute minimum amount of logic, and was very successful at it. However, since this core was read-only, it has seen been eclipsed by the current controllers.

When it came time to build yet another flash controller, I started getting tired of building controllers that were specific to one flash or another. This is the genesis of version one of this project.

This project contains three basic flash controllers that are designed to be completely general purpose. There's a basic SPI controller, a Dual SPI controller, and a Quad SPI controller. All three controllers are designed for high speed operation using an ODDR clock. All three controllers may optionally contain a configuration register to issue other-than-read commands. Likewise, they may optionally allow pipelined accesses whereby multiple values may be read without re-issuing either command or address. Further, the DSPI and QSPI controllers will also use an XIP mode by default.

Where the *universal* part of the flash controller designs breaks down is in the initial startup. A configuration option can be used to remove this logic. Alternatively, this startup logic is maintained within single generic block, allowing for easy adjustment.

For a description of how the internals of each core work, feel free to browse through the Architecture chapter.

The registers that control the cores are discussed in the Registers chapter.

As required, you can find a wishbone datasheet in Chapt. 5.

The final pertinent information for implementing the cores is found in the I/O Ports chapter, Chapt. 7.

As always, write me if you have any questions or problems.

2. Architecture

2.1 OPT_PIPE

2.2 OPT_CFG

2.3 OPT_STARTUP

2.4 LGFLASHSZ

3. Operation

Each of these flash controller cores is designed to be as simple as possible to implement. This means that reading from the flash is by design easy, but sending or receiving other commands is more involved.

3.1 Reading

Reading from the flash is designed to be as simple as reading from the Wishbone bus.

SPI Upon a read request, the SPI controller will issue a read request, `8'h03`, followed by the request address. Once the data returns from the flash, the value is returned on the bus.

DSPI The DUAL spi controller is built around the understanding that some startup code has already issued a Dual I/O read command `8'hbb` with the mode byte set for XIP mode. Ever after, a read request results in the address (not the command) sent to the flash, followed by the mode byte and then 8-dummy bits. Values are then returned on completion.

QSPI The Quad SPI controller does the same as the Dual SPI controller, save that the command issued is the Quad I/O read command, `8'heb`.

3.2 Pipelined Reading

If the `OPT_PIPE` option parameter is set, then consecutive reads may be made without sending the address or the read command multiple times. Using this mode, the various controllers can return a 32-bit value in 32 clocks (SPI mode), 16 clocks (Dual SPI mode), or 8 clocks (Quad SPI mode).

3.3 Startup

Some flash chips and implementations have a special startup sequences that need to be followed. This is the purpose of the `OPT_STARTUP` parameter and the startup section it enables.

The Dual and Quad SPI controllers have an additional startup need: they need the initial Dual I/O read or Quad I/O read command to be issued with a proper mode byte. At the end of this command, the device will be placed into an eXecute In Place (XIP) mode. From the XIP mode, read requests may start immediately with the address in Dual or Quad I/O mode for extra speed.

3.4 Low-level Control

Low level control can be accomplished through the use of the `R_FLASHCFG` register. This register will allow a software program to issue any command to the flash and receive any result. Configuration commands are all 8-bit commands. In SPI mode, the command writes the eight given bits to the `MOSI` port, while reading the bits back from the `MISO` port. A write to this port will initiate the transaction. A later read will return the 8-bits read during that operation.

For the two high speed controllers, the `R_FLASHCFG` register can also be used to place the controller into a high speed mode where either two I/O lines or all four I/O lines are used together to send or read a message. When a command is given for high speed operation, the command also includes the direction of that operation.

While the `R_FLASHCFG` port is controlling the SPI controller, any reads from the FPGA memory will silently fail.

Finally, when you are ready to return the controller to normal read opeartions, the Dual and Quad SPI controllers need to be placed back into Dual or Quad I/O Modes.

3.5 Low Level Examples

To use the low-level facilities, the higher speed flashes need to begin by taking the flash controller out of normal operating read mode. This can be done by first taking control of the interface.

fpga-¿writeio(R_FLASHCFG, F_END);

This writes a `13'h1100` to the interface, placing it into configuration control.

After this, writing 16-bits of zeros to the device in normal SPI mode will take the device out of any higher speed read mode it may have been in. This equates to sending a dummy address (only every other or every fourth bit will be read), followed by a one on the `MOSI` channel when the mode byte would have taken place. Then, when the CS line is later deactivated, the device will be out of any XIP mode and ready for commands.

`fpga->writeio(R_FLASHCFG, F_RESET); fpga->writeio(R_FLASHCFG, F_RESET); fpga->writeio(R_FLASHCFG, F_END);`

The device is now ready to accept any command the underlying SPI device supports.

For example, if we wish to issue a READID command (8'h9f), we could simply write

`m_fpga->writeio(R_FLASHCFG, CFG_USERMODE | 0x9f); // a 16'h109f`

This will send the READID command to the device. To read the command back, we'll need to clock the device and read one byte at a time. The following therefore will read eight bits: one write request to clock the interface, followed by a second read request to get the data returned from the interface.

`m_fpga->writeio(R_FLASHCFG, CFG_USERMODE | 0); // a 16'h1000 r = m_fpga->readio(R_FLASHCFG);`

The software flash driver will read and pack three more bytes before returning.

Finally, to return the device to normal operation, we'll neeed to issue a highspeed read command. The following shows a Quad I/O read command.

`fpga->writeio(R_FLASHCFG, QUAD_IO_READ);`

This is followed by three (dummy) address bytes.

`// 3 address bytes fpga->writeio(R_FLASHCFG, CFG_USERMODE | CFG_QSPEED | CFG_WEDIR); fpga->writeio(R_FLASHCFG, CFG_USERMODE | CFG_QSPEED | CFG_WEDIR); fpga->writeio(R_FLASHCFG, CFG_USERMODE | CFG_QSPEED | CFG_WEDIR);`

A mode byte, 8'ha0, is required to place the device into XIP mode upon completion.

// Mode byte `fpga->writeio(R_FLASHCFG, CFG_USERMODE | CFG_QSPEED | CFG_WEDIR | 0xa0);`

At this point, we need to change direction from write to read. Most interfaces require a dummy byte before the read command actually begins.

`// Read a dummy byte  fpga->writeio(R_FLASHCFG, CFG_USERMODE | CFG_QSPEED );`

Were we to continue, we'd be reading from the flash. Instead, let's end our command.

`// Close the interface  fpga->writeio(R_FLASHCFG, 0);`

3.6 External Software Control

To make this easier, a universal flash driver software is provided with this core. This driver requires the use of a class implementing the DEVID interface. From this interface, the controller uses the `readio` and `writeio` methods to access the controller. Reading from the controller is done using the `readi` (read with increment) vector method.

This controller has methods `take_offline()`, to remove the Dual or Quad SPI controllers from their high speed modes, as well as `place_online()` to return them to their high speed modes upon completion. The software uses the existence of either the `DSPI_FLASH` or the `QSPI_FLASH` macro to know whether to place the flash in Dual or Quad I/O modes upon completion.

The primary entry point for this controller is the `write()` method. The `write()` method takes as an argument an address within the flash memory space, a length, and a buffer. It then tries to write that buffer to that memory space. To do this, it will first check to see if the flash configuration is set properly by calling `verify_config()`. If not, it will try to `set_config()` and exit if a subsequent `verify_config()` fails. It then checks through the memory region given to determine if the sector needs to be erased, or just programmed.

If any bits within the region need to switch from zero to one, the `write()` method will call the `erase_sector()` method with an address of the sector that needs to be erased. (It doesn't support sub-sector erase calls.) A `verify` argument can also be given to double check that the erase was successful if desired.

Following the (optional) `erase_sector()`, the `write()` method will call the `page_program()` method. This method accepts an address, length, and data pointer. It will program the flash (turning ones to zeros), starting at the given address, using length bytes starting at the data pointer. As with the `erase_sector()` command, a `verify` flag may be passed to check that the page program command was successful.

As the `write()` method completes, it will issue a call to `place_online()` to put the flash controller back into a mode where it can read from the flash again.

4. Registers

The three flash controller support two address spaces: a single configuration and control register, and the flash memory itself. Only one can be read or written to at a time. Writes to the flash memory will quietly fail with no error. Reads from the flash memory while the configuration port are active will also quietly fail with no error. Reads from the configuration port will return undefined values unless the port is in configuration mode.

Table. 4.1 shows this one register.

Name	Address	Width	Access	Description
FLASHCFG	0	13	R/W	Configuration register, for low-level control

Table 4.1: One register for all three controllers

The bits within this configuration register are shown in Table 4.2.

Taking the device off line in order to issue a command is slightly different between the SPI controller and the higher speed controllers. With the SPI controller, all that needs to be done is to write to the configuration register with bit eight low. This will activate the chip-select and clock the eight data bits out of the port. A subsequent read may read the bits read during this time.

With the higher speed controllers, you also need to set the user override mode bit. Hence, if bit 12 is high and bit 8 is low, a message will be sent or received from the device. If the speed flag is left low, the message will be sent in normal SPI mode.

If bit 12 is high and bit 8 is high, a high speed device will be removed from normal operation, but nothing will be sent to it. Setting thedual or quad speed flag at thte same time will allow you to read back which of the two implementations was built into the design. If the Dual Speed bit is set, then Dual controller is implemented within the design. If the Quad speed bit is set, then the quad controller is implemented within the design. Sadly, this method will not work with the SPI controller simply because commands with bit 8 high will be ignored, and commands with bit 8 low will send a message across the SPI channel. For this reason, the software controller uses the `DUAL_SPI` and `QUAD_SPI` macros to determine speed instead.

Bit #	Access	Description
13-31		Reserved. These bits are ignored on read and write
12	R/W	User mode. True if the port being controlled by the FLASHCFG register.
11	R/W	QUAD speed. True if the controller supports QUAD I/O mode, and if the port has been set to high speed QUAD mode.
10	R/W	Dual speed. True if the controller supports DUAL I/O mode, and if the port has been set to high speed DUAL mode.
9	R/W	High speed direction. True if the pins of the bus are all set for writing, false if they are all high impedence for reading. This pin is read-only for the SPI flash controller.
8	R/W	CS_n. If zero, the CS pin is active (low).
0–7	R/W	These are data lines. A write to the SPI interface with bit 8 low will write these bits to the interface using SPI mode. Likewise, for the DUAL and QUAD interfaces, a write to the register with bit 8 low, bit 12 high, and the speed and direction bits set will write these bits at high speed across the port. If the speed bit isn't set, the direction bit will be ignored and the bits will be written using SPI mode. Once the operation is completed, any resulting bits from SPI mode, or from a read operation where the direction bit was clear, may be read from these bits of this register.

Table 4.2: FLASHCFG bit definitions

5.

Wishbone Datasheet

Tbl. 5.1 is required by the wishbone specification, and so it is included here.

Description	Specification	
Revision level of wishbone	WB B4	
Type of interface	Slave, (Pipelind) Read/Write	
Port size	32–bit	
Port granularity	32–bit	
Maximum Operand Size	32–bit	
Data transfer ordering	Big Endian	
Clock constraints	Most serial flash chips require 100MHz or slower	
Signal Names	Signal Name	Wishbone Equivalent
	`i_clk`	`CLK_I`
	`i_reset`	`CLK_I`
	`i_wb_cyc`	`CYC_I`
	`i_wb_stb`	`STB_I`
	`i_cfg_stb`	`STB_I`
	`i_wb_we`	`WE_I`
	`i_wb_addr`	`ADR_I`
	`i_wb_data`	`DAT_I`
	`o_wb_ack`	`ACK_O`
	`o_wb_stall`	`STALL_O`
	`o_wb_data`	`DAT_O`

Table 5.1: Wishbone Datasheet for the basic Flash controller

6. Clocks

These cores are based upon the existence of an ODDR controller for the clock. Using this ODDR part, it is possible to run the controller SCK at the system clock rate. Most flash parts, however, are limited to roughly 108MHz or so. A safe clock speed limit should therefore be about 100MHz. See Table. 6.1.

Name	Source	Rates (MHz)		Description
		Max	Min	
i_clk	External	100		System clock.

Table 6.1: Clock speeds

7. I/O Ports

There are two primary interfaces that these cores support: a wishbone interface, and the interface to the top level I/O's. Both of these have their own section in the I/O port list. For the purpose of this table, the wishbone interface is listed in Tbl. 7.1, and the flash interface is listed in Tbl. ??. The two lines that don't really fit this classification are found in Tbl. 7.2.

Port	Width	Direction	Description
i_wb_cyc	1	Input	Wishbone bus cycle wire.
i_wb_data_stb	1	Input	Wishbone strobe, when the access is to the data memory.
i_wb_cfg_stb	1	Input	Wishbone strobe, for when the access is to the configuration register
i_wb_we	1	Input	Wishbone write enable, indicating a write interaction to the bus. Writes tot he memory interface will be quietly ignored
i_wb_addr	21	Input	Wishbone address. When accessing the configuration, register, these address bits are ignored.
i_wb_data	32	Input	Wishbone bus data register. Used by the configuration register only.
o_wb_ack	1	Output	Return value acknowledging a wishbone write, or signifying valid data in the case of a wishbone read request.
o_wb_stall	1	Output	Indicates the device is not yet ready for another wishbone access, effectively stalling the bus.
o_wb_data	32	Output	Wishbone data bus, returning data values read from the interface.

Table 7.1: Wishbone I/O Ports

While this core is wishbone compatible, there was one necessary change to the wishbone interface to make this possible. That was the split of the strobe line into two separate lines. The first strobe line, the data strobe, is used when the access is to data memory–such as a read or write (program) access. The second strobe line, the control strobe, is for reads and writes to one of the four control registers. By splitting these strobe lines, the wishbone interconnect designer may place the control registers in a separate location of wishbone address space from the flash memory. It is an error for both strobe lines to be on at the same time.

With respect to the higher speed interfaces, one piece of glue logic is necessary to tie the flash I/O to the in/out port at the top level of the device. Specifically, the following two lines must be added at the top level. First, for dual I/O,

```
assign io_dspi_dat = (~qspi_mod[1])?({1'bz,dspi_dat[0]}) // Serial mode
                     :((dspi_bmod[0])?(2'bzz):(dspi_dat[1:0])); // Dual mode
```

and then for quad I/O,

```
assign io_qspi_dat = (~qspi_mod[1])?({2'b11,1'bz,qspi_dat[0]}) // Serial mode
                     :((qspi_bmod[0])?(4'bzzzz):(qspi_dat[3:0])); // Quad mode
```

These provide the transition between the input and output ports used by this core, and the bi–directional inout ports used by the actual part. Further, because the two additional lines are defined to be ones during serial I/O mode, the hold and write protect lines are effectively eliminated in this design in favor of faster speed I/O (i.e., Quad I/O).

Finally, the clock line is not specific to the wishbone bus, and the interrupt line is not specific to any of the above. These have been separated out here.

Port	Width	Direction	Description
i_clk	1	Input	The 100 MHz clock driving all SPI interactions.

Table 7.2: Other I/O Ports