Gisselquist
Technology, LLC

QUAD SPI FLASH CONTROLLER SPECIFICATION

Dan Gisselquist, Ph.D.
dgisselq (at) opencores.org

August 11, 2016

Revision History

Rev.	Date	Author	Description
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

Figure Page

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.

That was the beginning of the QSPI flash controller.

The EQSPI flash controller started from a similar need for a board that had an EQSPI flash. That particular board was an Arty, and so the EQSPI flash controller has been designed around the Arty platform.

Dan Gisselquist, Ph.D.

1.

Introduction

This document discusses the design and usage of two cores: a Quad SPI flash controller, and a newer Extended Quad SPI flash controller. In general, the two are *very* similar. However, their construction and register usage are subtly different, so the user will need to pay attention to these differences.

Both Flash controllers handle all of the necessary queries and accesses to and from a SPI Flash device that has been augmented with an additional two data lines and enabled with a mode allowing all four data lines to work together in the same direction at the same time. Since the interface was derived from a SPI interface, most of the interaction takes place using normal SPI protocols and only some commands work at the higher four bits at a time speed. This remains true, even though the newer Extended SPI flash controller allows control accesses and Dual I/O and Quad I/O speeds: control interactions remain at SPI speeds, and only data reads and writes take place at the Quad I/O speed.

Both controllers attempt to mask the underlying operation of the Flash device behind a wishbone interface, to make it so that reads and writes are as simple as using the wishbone interface. However, the difference between erasing (turning bits from '0' to '1') and programming (turning bits from '1' to '0') breaks this model somewhat. Therefore, reads from the device act like normal wishbone reads, writes program the device (if the write protect is properly removed) and sort of work with the wishbone, while erase commands require another register to control. Please read the Operations chapter for a detailed description of how to perform these relevant operations.

This QSPI controller implements the interface for the Quad SPI flash found on the Basys-3 board built by Digilent, Inc, as well as their CMod-S6 board. A similar controller has been modified for the flash on the XuLA2-LX25 SoC. It is possible that some portions of the interface may be specific to the Spansion S25FL032P chip used on the Basys-3 board, and the 100 MHz system clock found on the board, although there is no reason the controller needs to be limited to this architecture. It just happens to be the one the QSPI controller was designed to and for.

The Extended QSPI controller, or EQSPI controller, was designed to control the Micron Serial NOR Flash Memory, N25Q128A, found on Digilent's Arty board. As with the Spansion chip, it is possible that parts of the interface are specific to this board and this chip.

For a description of how the internal 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

The internal architecture of each of these two cores is different, reflecting their different chips and goals. The QSPI controller is designed to run using a 50 MHz SPI clock, generated from a 100 MHz controller clock. The EQSPI controller, however, was designed to prove that a 100 MHz SPI clock could be used to drive a flash controller from a 200 MHz controller clock. As a result of these clocking differences, the architectures of each are quite different.

2.1 QSPI Flash Architecture

As built, the core consists of only two components: the wishbone quad SPI flash controller, `wbqspiflash`, and the lower level quad SPI driver, `llqspi`. The controller issues high level read/write commands to the lower level driver, which actually implements the Quad SPI protocol.

Pictorally, this looks something like Fig. 2.1. This is also what you will find if you browse through the code.

While it isn't relevant for operating the device, a quick description of these internal wires may be educational. The lower level device is commanded by asserting a `spi_wr` signal when the device is not busy (i.e. `spi_busy` is low). The actual command given depends upon the other signals. `spi_len` is a two bit value indicating whether this is an 8 bit (2'b00), 16 bit (2'b01), 24 bit (2'b10), or 32 bit (2'b11) transaction. The data to be sent out the port is placed into `spi_in`.

Further, to support Quad I/O, `spi_spd` can be set to one to use all four bits. In this case, `spi_dir` must also be set to either 1'b0 for writing, or 1'b1 to read from the four bits.

When data is valid from the lower level driver, the `spi_valid` line will go high and `spi_out` will contain the data with the most recently read bits in the lower bits. Further, when the device is idle, `spi_busy` will go low, where it may then read another command.

Sadly, this simple interface as originally designed doesn't work on a device where transactions can be longer than 32 bits. To support these longer transactions, the lower level driver checks the `spi_wr` line before it finishes any transaction. If the line is high, the lower level driver will deassert `spi_busy` for one cycle while reading the command from the controller on the previous cycle. Further, the controller can also assert the `spi_hold` line which will stop the clock to the device and force everything to wait for further instructions.

This hold line interface was necessary to deal with a slow wishbone bus that was writing to the device, but that didn't have it's next data line ready. Thus, by holding the `i_wb_cyc` line high, a write could take many clocks and the flash would simply wait for it. (I was commanding the device via a serial port, so writes could take *many* clock cycles for each word to come through, i.e. 1,500 clocks or so per word and that's at high speed.)

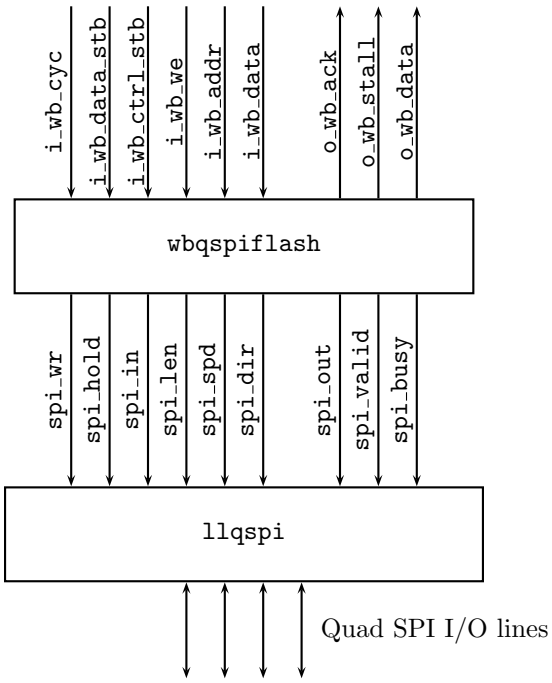

Figure 2.1: QSPI Architecture Diagram

The upper level component, the controller `wbqspiflash`, is little more than a glorified state machine that interacts with the wishbone bus. From it's idle state, it can handle any command, whether data or control, and issue appropriate commands to the lower level driver. From any other state, it will stall the bus until it comes back to idle–with a few exceptions. Subsequent data reads, while reading data, will keep the device reading. Subsequent data writes, while in program mode, will keep filling the devices buffer before starting the write. In other respects, the device will just stall the bus until it comes back to idle.

While they aren't used in this design, the wishbone error and retry signals would've made a lot of sense here. Specifically, it should be an error to read from the device while it is in the middle of an erase or program command. Instead, this core stalls the bus–trying to do good for everyone. Perhaps a later, updated, implementation will make better use of these signals instead of stalling. For now, this core just stalls the bus.

Perhaps the best takeaway from this architecture section is that the varying pieces of complexity have each been separated from each other. There's a lower level driver that handles actually toggling the lines to the port, while the higher level driver maintains the state machine controlling which commands need to be issued and when.

2.2 EQSPI Flash Architecture

The EQSPI flash architecture was an entire redesign. The reason for the redesign is quite simple: the QSPI flash controller was just way to complex to run at a 200 MHz clock. This new and modified architecture is shown in Fig. 2.2. All of the various modules of this architecture, save the `lleqspi` and `xioddr` modules, are found in the `eqspiflash.v` file.

The goal of this architecture was to reduce the amount of logic necessary to process the many various requests this controller allows.

At the top, all requests to the controller come from the bus straight into the `qspibus` module. The purpose of this module is to parse the various commands to their respective modules. One command, however, never gets parsed: the request to read from the erase register. This register returns the status of the controller, and particularly whether or not it is still busy with the last erase or write command.

The top level controller has the ability to latch a bus request. Such requests are then issued to the lower level controllers. However, they remain latched in the top level controller until the lower controller acknowledges them, at which point the bus may advance to its next request. Depending on the lower level controller, this may not occurr until the lower level transaction is complete, or nearly so.

The lower level controllers also communicate with the command multiplexer beneath them. Each controller has a request line, whereby it requests access to the lowerest level controller. Once granted, the controller maintains control of that lowest level until it is released. In this fashion, for example, the `readqspi` controller implements the execute in place functionality: it reads from the interface, then maintains the interface. If another driver requests the interface, the read controller reactivates itself and returns the interface to a non–XIP mode.

Now, of these four controllers, the `readqspi` controller handles reads from the device. Reads are always done in Quad SPI mode, if so enabled, and the device is left in XIP mode until another controller requests the interface. XIP mode is left by reading a 32'bit value from the device at address zero.

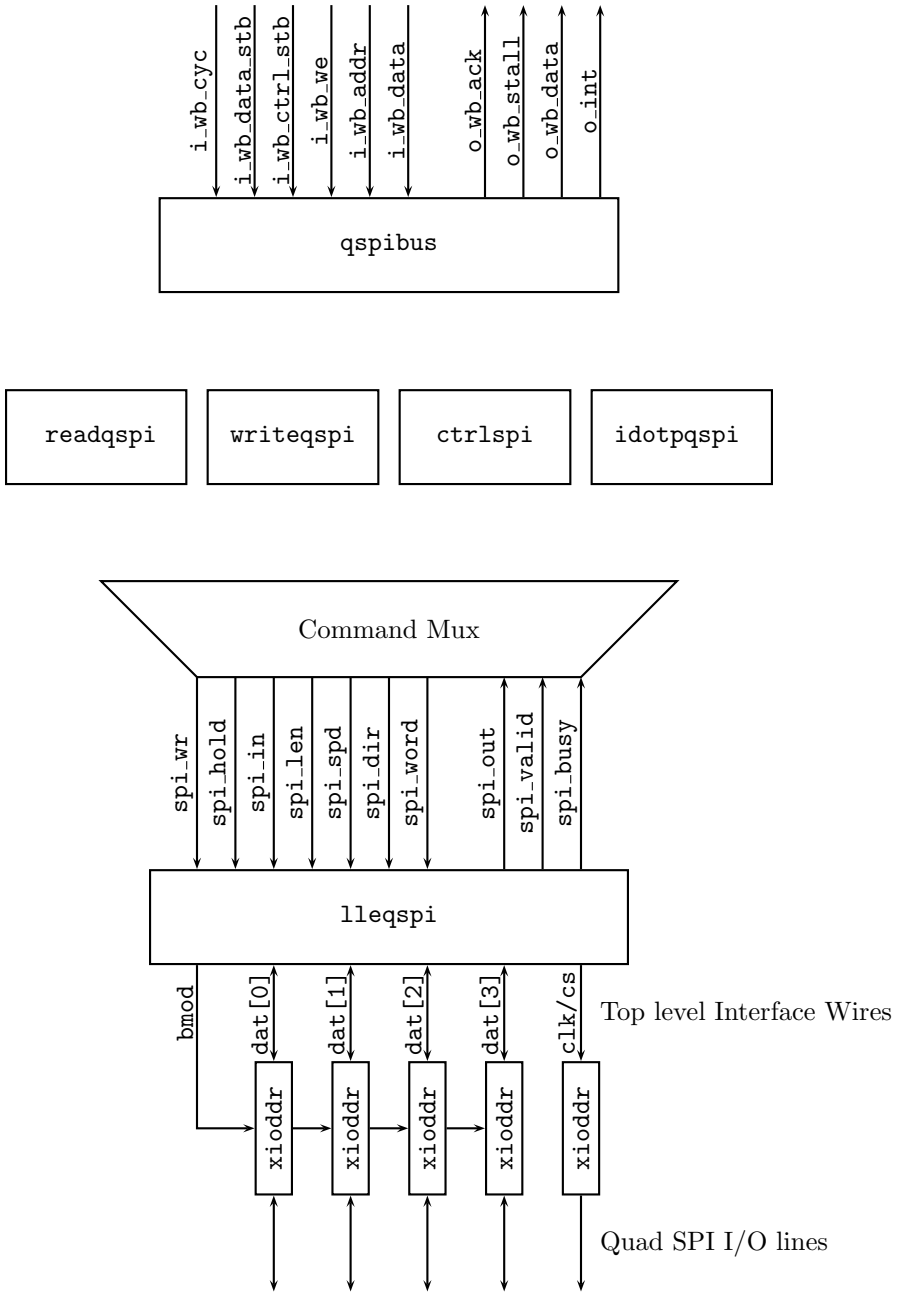

Figure 2.2: EQSPI Architecture Diagram

The `writeqspi` controller handles both program and erase requests. Upon completion of either request, the `writeqspi` controller holds on to the interface perpetually reading from the status register until the device is no longer busy.

The `ctrlspi` controller handles requests to read and write the various control registers internal to the device. These include the status register, the non–volatile configuration register, the volatile configuration register, the extended volatile configuration register, the flags register, and the lock registers associated with the most recently selected sector (set in the erase register). Writes to these registers, though, aren't quite so simple: One must first disable the write protect in the erase control register, thus setting the Write Enable Latch of the device, before the device will accept write requests.

Finally, the `idotpqspi` controller handles the logic associated with the ID memory internal to the controller, as well as both reading and writing the One Time Programmable (OTP) registers, and eventually locking the OTP registers so that they can no longer be read or written. As with the `writeqspi` module, write requests do not release the port until the write has completed. Reading the erase register will provide the status on this operation.

Moving down in the architecture to the command multiplexer, this portion is really not that remarkable. It takes one clock for requests to go through the command multiplexer and get to the lower level controller, and it grants and releases control to the various controllers.

The `lleqspi` controller is the lower level controller for this device. It's operation mirrors that of the `llqspi` lower–level controller from the QSPI flash. The biggest difference is that the Micron chip requires a particular recovery time following any command other than a read command leaving the chip in the XIP mode.

What is new in this controller is the requirement for the output ports to be connected to the I/O banks via `ODDR` and `IDDR` modules. These are contained within the `xioddr` modules. Since the wires can change direction, the `bmod` pair of wires provides an indication of which direction the various bits of the port are moving–either as inputs or outputs.

3.

Operation

This implementation attempts to encapsulate (hide) the details of the chip from the user, so that the user does not need to know about the various subcommands going to and from the chip. The original goal was to make the chip act like any other read/write memory, however the difference between erasing and programming a flash chip made this impossible. Therefore a separate register is provided to control the erase any given sector, while reads and writes may proceed (almost) as normal.

The wishbone bus that this controller works with, however, is a 32–bit bus. Address one on the bus addresses a completely different 32–bit word from address zero or address two. Bus select lines are not implemented, all operations are 32–bit. Further, the device is little–endian, meaning that the low order byte is the first byte that will be or is stored on the flash.

3.1 High Level

From a high level perspective, this core provides read/write access to the device either via the wishbone (read and program), or through a control register found on the wishbone (the EREG). Programming the device consists of first clearing the write protect, and then erasing the region of interest. This will set all the bits to '1' in that region. After erasing the region, the write protect may again be cleared, and then the region can then be programmed, setting some of the '1' bits to '0's. When neither erase nor program operation is going on, the device may be read. The section will describe each of those operations in detail.

To erase a sector of the device, two writes are required to the EREG register. The first write turns off the write protect bit, whereas the second write commands the erase itself. The first write should equal 0x1000_0000 for the QSPI controller and 0x4000_0000 for the EQSPI controller. After this write, the EQSPI controller will issue a Write Enable command to the device. For the QSPI controller, the second write should be any address within the sector to be erased together with setting the high bit of the register. This is equivalent to setting it to 0x8000_0000 plus the address. The EQSPI flash driver is subtly different in that it requires a *key* to erase. Hence, for the EQSPI flash driver, one must write 0xc000_01be plus the first address in the sector to accomplish the same result. Further, the EQSPI flash controller allows the erasing of 4 kB subsegments. To do this, the second write must also set the subsector bit, so it looks like writing 0xd000_01be plus the first address in the subsector. After this second write, the QSPI controller will issue a write–enable command to the device (the EQSPI controller will have already issued the write–enable), followed by a sector erase command. In summary, for the QSPI flash:

1. Disable write protect by writing `0x1000_0000` to the EREG register

2. Command the erase by writing `0x8000_0000` plus the device address to the EREG register. (Remember, this is the *word address* of interest, not the *byte address*.)

and for the EQSPI flash:

1. Disable write protect by writing `0x4000_0000` to the EREG register
2. Command the sector (64 kB) erase by writing `0xc000_01be` plus the first address in the segment to the EREG register.

   In the case of a subsegment (4 kB) erase command, write `0xd000_01be` plus the first address in the subsegment to the EREG register.

While the device is erasing, the controller will idle while checking the status register over and over again. Should you wish to read from the EREG during this time, the high order bit of the EREG register will be set indicating that a write is in progress (WIP). Once the erase is complete, this bit will clear, the interrupt line will be strobed high, and other operations may take then place on the part. Any attempt to perform another operation on the part prior to that time will stall the bus until the erase is complete.

Once an area has been erased, it may then be programmed. To program the device, first disable the write protect by writing a `0x1000_0000` to the EREG register for the QSPI controller, or `0x4000_0000` for the EQSPI controller. After that, you may then write to the area in question whatever values you wish to program. One 256 byte (64 bus word) page may be programmed at a time. Pages start on even boundaries, such as addresses `0x040`, `0x080`, `0x0100`, etc. To program a whole page at a time, write the 64 words of the page to the controller without dropping the `i_wb_cyc` line. Attempts to write more than 64 words will stall the bus, as will attempts to write more than one page. Writes of less than a page work as well. In summary,

1. Disable the write protect by writing a `0x1000_0000` to the EREG register when using the QSPI flash controller, or `0x4000_0000` for the EQSPI flash controller.
2. Write the page of interest to the data memory of the device.

   The first address should start at the beginning of a page (bottom six bits zero), and end at the end of the page (bottom six bits one, top bits identical). Writes of less than a page are okay. Writes crossing page boundaries will stall the bus, while waiting for the first write to complete before attempting to start the second write.

While the device is programming a page, the controller will idle while checking the status register as it did during an erase. During this idle, both the EREG register and the device status register may be queried. Once the status register drops the write in progress line, the top level bit of the EREG register will be cleared and the interrupt line strobed. Prior to this time, any other bus operation will stall the bus until the write completes.

Reads are simple for the QSPI flash controller, you just read from the device and the device does everything you expect. Reads may be pipelined. To use the QSPI mode of transferring 4–bits at a time, when using the QSPI controller, you must first either read (or set) the quad mode bit in the configuration register. This will enable Quad–I/O mode reads. Once enabled, reads will take place four bits at a time from the bus.

Using the EQSPI flash controller, reads are almost as simple, but with a couple of caveats. The first caveat is that the controller defaults to Quad I/O mode, and will not leave it. The problem

is that this mode depends upon a variable number of dummy cycles set to 8. Hence, before issuing reads from the data section of the device, the number of dummy cycles will need to be set in either the volatile or non–volatile configuration register.

Both controllers provide for a special mode following a read, where the next read may start immediately in Quad I/O mode following a 12 clock setup for the QSPI controller, or 16 clocks for the EQSPI controller. Both controllers leaves the device in this mode following any initial read. Therefore, back to back reads as part of separate bus cycles will only take 20 clocks (24 for EQSPI) to read the first word, and 8 clocks per word thereafter. Other commands, however, such as erasing, writing, reading from the status, configuration, or ID registers, will require a 32 device clock operation before entering.

3.2 Low Level

At a lower level, the QSPI core implements the following Quad SPI commands:

1. FAST_READ, when a read is requested and Quad mode has not been enabled.
2. QIOR, or quad I/O high performance read mode. This is the default read command when Quad mode has been enabled, and it leaves the device in the Quad I/O High Performance Read mode, ready for a faster second read command.
3. RDID, or Read identification
4. WREN, or Write Enable, is issued prior to any erase, program, or write register (i.e. configuration or status) command. This detail is hidden from the user.
5. RDSR, or read status register, is issued any time the user attempts to read from the status register. Further, following an erase or a write command, the device is left reading this register over and over again until the write completes.
6. RCR, or read configuration, is issued any time a request is made to read from the configuration register. Following such a read, the quad I/O may be enabled for the device, if it is enabled in this register.
7. WRR, or write registers, is issued upon any write to the status or configuration registers. To separate the two, the last value read from the status register is written to the status register when writing the configuration register.
8. PP, or page program, is issued to program the device in serial mode whenever programming is desired and the quad I/O has not been enabled.
9. QPP, or quad page program, is used to program the device whenever a write is requested and quad I/O mode has been enabled.
10. SE, or sector erase, is the only type of erase this core supports.
11. CLSR, or Clear Status Register, is issued any time the last status register had the bits `P_ERR` or `E_ERR` set and the write to the status register attempts to clear one of these. This command is then issued following the WRR command.

4.

Registers

4.1 QSPI Controller

The QSPI controller supports four control registers. These are the EREG register, the configuration register, the status register, and the device ID, as shown and listed in Table. 4.1.

Name	Address	Width	Access	Description
EREG	0	32	R/W	An overall control register, providing instant status from the device and controlling erase commands.
Config	1	8	R/W	The devices configuration register.
Status	2	8	R/W	The devices status register.
ID	3	16	R	Reads the 16-bit ID from the device.

Table 4.1: List of QSPI Registers

4.1.1 EREG Register

The EREG register was designed to be a replacement for all of the device registers, leaving all the other registers a part of a lower level access used only in debugging the device. This would've been the case, save that one may need to set bit one of the configuration register to enter high speed mode.

The bits associated with this register are listed in Tbl. 4.2.

In general, only three bits and an address are of interest here.

The first bit of interest is bit 27, which will tell you if you are in Quad–I/O mode. The device will automatically start up in SPI serial mode. Upon reading the configuration register, it will transition to Quad–I/O mode if the QUAD bit is set. Likewise, if the bit is written to the configuration register it will transition to Quad–I/O mode.

While this may seem kind of strange, I have found this setup useful. It allows me to debug commands that might work in serial mode but not quad I/O mode, and it allows me to explicitly switch to Quad I/O mode. Further, writes to the configuration register are non–volatile and in some cases permanent. Therefore, it doesn't make sense that a controller should perform such a write without first being told to do so. Therefore, this bit is set upon noticing that the QUAD bit is set in the configuration register.

Bit #	Access	Description
31	R/W	Write in Progress/Erase. On a read, this bit will be high if any write or erase operation is in progress, zero otherwise. To erase a sector, set this bit to a one. Otherwise, writes should keep this register at zero.
30	R	Dirty bit. The sector referenced has been written to since it was erased. This bit is meaningless between startup and the first erase, but valid afterwards.
29	R	Busy bit. This bit returns a one any time the lower level Quad SPI core is active. However, to read this register, the lower level core must be inactive, so this register should always read zero.
28	R/W	Disable write protect. Set this to a one to disable the write protect mode, or to a zero to re–enable write protect on this chip. Note that this register is not self–clearing. Therefore, write protection may still be disabled following an erase or a write. Clear this manually when you wish to re–enable write protection.
27	R	Returns a one if the device is in high speed (4-bit I/O) mode. To set the device into high speed mode, set bit 1 of the configuration register.
20–26	R	Always return zero.
14–19	R/W	The sector address bits of the last sector erased. If the erase line bit is set while writing this register, these bits will be set as well with the sector being erased.
0–13	R	Always return zero.

Table 4.2: EREG bit definitions

The second bit of interest is the write protect disable bit. Write a '1' to this bit before any erase or program operation, and a '0' to this bit otherwise. This allows you to make sure that accidental bus writes to the wrong address won't reprogram your flash (which they would do otherwise).

The final bit of interest is the write in progress slash erase bit. On read, this bit mirrors the WIP bit in the status register. It will be a one during any ongoing erase or programming operation, and clear otherwise. Further, to erase a sector, disable the write protect and then set this bit to a one while simultaneously writing the sector of interest to the device.

The last item of interest in this register is the sector address of interest. This was placed in bits 14–19 so that any address within the sector would work. Thus, to erase a sector, write the sector address, together with an erase bit, to this register.

4.1.2 Config Register

The Quad Flash device also has a non–volatile configuration register, as shown in Tbl. 4.3. Writes to this register are program events, which will stall subsequent bus operations until the write in progress bit of either the status or EREG registers clears. Note that some bits, once written, cannot be cleared such as the BPNV bit.

Writes to this register are not truly independent of the status register, as the Write Registers (WRR) command writes the status register before the configuration register. Therefore, the core implements this by writing the status register with the last value that was read by the core, or zero if the status register has yet to be read by the core. Following the status register write, the new value for the configuration register is written.

Bit #	Access	Description
8–31	R	Always return zero.
6–7	R	Not used.
5	R/W	TBPROT. Configures the start of block protection. See device documentation for more information. (Default 0)
4	R/W	Do not use. (Default 0)
3	R/W	BPNV, configures BP2–0 bits in the status register. If this bit is set to 1, these bits are volatile, if set to '0' (default) the bits are non–volatile. *Note that once this bit has been set, it cannot be cleared!*
2	R/W	TBPARM. Configures the parameter sector location. See device documentation for more detailed information. (Default 0)
1	R/W	QUAD. Set to '1' to place the device into Quad I/O (4–bit) mode, '0' to leave in dual or serial I/O mode. (This core does not support dual I/O mode.) (Most programmers will set this to '1'.)
0	R/W	FREEZE. Set to '1' to lock bits BP2–0 in the status register, zero otherwise. (Default 0).

Table 4.3: Configuration bit definitions

Further information on this register is available in the device data sheet.

4.1.3 Status Register

The definitions of the bits in the status register are shown in Tbl. 4.4. For operating this core, only the write in progress bit is relevant. All other bits should be set to zero.

Bit #	Access	Description
8–31	R	Always return zero.
7	R/W	Status register write disable. This setting is irrelevant in the current core configuration, since the W#/ACC line is always kept high.
6	R/W	P_ERR. The device will set this to a one if a programming error has occurred. Writes with either P_ERR or E_ERR cleared will clear this bit.
5	R/W	E_ERR. The device will set this to a one if an erase error has occurred, zero otherwise. Writes clearing either P_ERR or E_ERR will clear this bit.
2–4	R/W	Block protect bits. This core assumes these bits are zero. See device documentation for other possible settings.
1	R	Write Enable Latch. This bit is handled internally by the core, being set before any program or erase operation and cleared by the operation itself. Therefore, reads should always read this line as low.
0	R	Write in Progress. This bit, when one, indicates that an erase or program operation is in progress. It will be cleared upon completion.

Table 4.4: Status bit definitions

4.1.4 Device ID

Reading from the Device ID register causes the core controller to issue a RDID `0x9f` command. The bytes returned are first the manufacture ID of the part (`0x01` for this part), followed by the device ID (`0x0215` for this part), followed by the number of extended bytes that may be read (`0x4D` for this part). This controller provides no means of reading these extended bytes. (See Tab. 4.5)

Bit #	Access	Description
0–31	R	Always reads `0x0102154d`.

Table 4.5: Read ID 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 spec
Type of interface	Slave, (Block) Read/Write
Port size	32–bit
Port granularity	32–bit
Maximum Operand Size	32–bit
Data transfer ordering	Little Endian
Clock constraints	Must be 100 MHz or slower (QSPI)
	Must be 200 MHz or slower (EQSPI)
Signal Names	Signal Name / Wishbone Equivalent i_clk_100mhz — CLK_I (QSPI) i_clk_200mhz — CLK_I (EQSPI) i_wb_cyc — CYC_I i_wb_ctrl_stb — STB_I i_wb_data_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 (E)QSPI Flash controller

The EQSPI flash controller has a further simplified wishbone usage: the strobe lin, `i_wb_ctrl_stb` or `i_wb_data_stb`, must be guaranteed low any time `i_wb_cyc` is low. This simplifies transaction processing internal to the controller, and is part of the method of getting the controller running at 200 MHz.

5.1 EQSPI Controller

5.1.1 EREG Register

5.1.2 Status Register

5.1.3 Non–Volatile Configuration Register

5.1.4 Volatile Configuration Register

5.1.5 Extended Volatile Configuration Register

5.1.6 Sector Lock Register

5.1.7 Flag Status Register

5.1.8 Identification memory

5.1.9 One–Time Programmable Memory

6.

Clocks

The QSPI core is based upon the Basys–3 design. The Basys–3 development board contains one external 100 MHz clock. This clock is divided by two to create the 50 MHz clock used to drive the device. According to the data sheet, it should be possible to run this core at up to 160 MHz, however I have not tested it at such speeds. See Table. 6.1.

Name	Source	Rates (MHz)		Description
		Max	Min	
i_clk_100mhz	External	160		System clock.

Table 6.1: List of QSPI Controller Clocks

The EQSPI core is based upon a very similar Arty design, but one that instead uses a 200 MHz core clock frequency. In a fashion similar to the QSPI controller, this clock is divided down to create a 100 MHz clock to command the device. Hence, the EQSPI clock is much faster, as shown in Table. 6.2.

Name	Source	Rates (MHz)		Description
		Max	Min	
i_clk_200mhz	External	200		System clock.

Table 6.2: List of EQSPI Controller Clocks

7.

I/O Ports

There are two interfaces that this device supports: a wishbone interface, and the interface to the Quad–SPI flash itself. 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 Quad SPI flash interface is listed in Tbl. 7.2. The two lines that don't really fit this classification are found in Tbl. 7.3.

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_ctrl_stb	1	Input	Wishbone strobe, for when the access is to one of control registers.
i_wb_we	1	Input	Wishbone write enable, indicating a write interaction to the bus.
i_wb_addr	19	Input	Wishbone address. When accessing control registers, only the bottom two bits are relevant all other bits are ignored.
i_wb_data	32	Input	Wishbone bus data register.
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 Quad SPI interface itself, one piece of glue logic is necessary to tie the QSPI flash I/O to the in/out port at the top level of the device. Specifically, these two lines must be added somewhere:

```
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).

The EQSPI controller is similar, but the glue logic is more involved.

Port	Width	Direction	Description
o_qspi_sck	1	Output	Serial clock output to the device. This pin will be either inactive, or it will toggle at 50 MHz.
o_qpsi_cs_n	1	Output	Chip enable, active low. This will be set low at the beginning of any interaction with the chip, and will be held low throughout the interaction.
o_qspi_mod	2	Output	Two mode lines for the top level to control how the output data lines interact with the device. See the text for how to use these lines.
o_qspi_dat	4	Output	Four output lines, the least of which is the old SPI MOSI line. When selected by the o_qspi_mod, this output becomes the command for all 4 QSPI I/O lines.
i_qspi_dat	4	Input	The four input lines from the device, of which line one, `i_qspi_dat[1]`, is the old MISO line.

Table 7.2: List of Quad–SPI Flash I/O ports

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_100mhz	1	Input	The 100 MHz clock driving all QSPI interactions.
o_interrupt	1	Output	An strobed interrupt line indicating the end of any erase or write transaction. This line will be high for exactly one clock cycle, indicating that the core is again available for commanding.

Table 7.3: Other I/O Ports