Gisselquist
Technology, LLC

SDSPI CONTROLLER SPECIFICATION

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

June 18, 2016

Copyright (C) 2016, Gisselquist Technology, LLC

This project is free software (firmware): you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTIBILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see `http://www.gnu.org/licenses/` for a copy.

Revision History

Rev.	Date	Author	Description
0.1	6/18/2016	Gisselquist	First Draft

Contents

Page

1 Introduction 1

2 Architecture 2

3 Operation 4
3.1 Constants 4
3.2 SD–Card Setup 5
3.3 Reading Card Registers 7
3.4 Reading and Writing 8

4 Registers 10
4.1 CMD Register 10
4.2 DATA Register 11
4.3 FIFO Registers 11
4.4 CONFIG Register 11

5 Wishbone Datasheet 13

6 Clocks 14

7 I/O Ports 15

Figures

Figure Page

4.1. CMD Register fields 10
4.2. CONFIG Register fields 12

Tables

Table Page

4.1. I/O Peripheral Registers 10

5.1. Wishbone Slave Datasheet 13

7.1. List of IO ports 15

Preface

When I started this project, I was informed that other projects similar to this one existed. The OpenRISC project has used an SD–Card controller, for example, as has the Google project vault. Of these two, the first uses the full SD–Card interface which is unavailable on the XuLA2 board, and I could never find the code for the second.

Still, had I found such interfaces, I would've still had another reason for building my own: controlling the license. By rolling my own interface, I can offer it to anyone interested in it under the GPL license, such as you have here. Further, by not using code belonging to others, I am not restricted or encumbered by any of their licenses–whether it be the GPL or otherwise. This code, and specification document, are therefore completely the product of Gisselquist Technology, LLC.

That said, I am indebted to Mr. Tambe for sharing his interface. While the current work does not use his approach and represents a complete rewrite, his approach was valuable in helping me understand what the SD–Card specification meant at several points along the way.

Dan Gisselquist, Ph.D.

1. Introduction

This Verilog core exports an SD card controller interface from internal to an FPGA to the rest of the FPGA core, while taking care of the lower level details internal to the interface. Unlike the other OpenCores SD Card controller[1] which offers a full SD–interface, this controller focuses on the SPI interface of the SD Card. While this is a slower interface, the SPI interface is necessary to access the card when using a XuLA2 board[2], or in general any time the full 9–bit, bi–directional interface to the SD card has not been implemented. Further, for those who are die–hard Verilog authors, this core is written in Verilog as opposed to the XESS provided demonstration SD Card controller found on GitHub[3], which was written in VHDL. For those who are not such die–hard Verilog authors, this controller provides a lower level interface to the card than these other controllers. Whereas the XESS controller will automatically start up the card and interact with it, this controller requires external software to be used when interacting with the card. This makes the SDSPI controller both more versatile, in the face of potential changes to the card interface, but also less–turn key.

While this core was written for the purpose of being used with the ZipCPU, as enhanced by the Wishbone DMA controller used by the ZipCPU, nothing in this core prevents it from being used with any other architecture that supports the 32–bit Wishbone interface of this core.

This core has been written as a wishbone slave, not a master. Using the core together with a separate master, such as a CPU or a DMA controller, only makes sense. This design, however, also restricts the core from being able to use the multiple block write or multiple block read commands, restricting us to single block read and write commands alone.

[1] See `http://www.opencores.org/project,sdcard_mass_storage_controller`.

[2] See `http://www.xess.com/shop/product/xula2-lx25/`

[3] See `https://github.com/xesscorp/VHDL_Lib/SDCard.vhd`

2. Architecture

This SD Card interface is designed to provide a means of commanding an SD Card, via the SPI port, and returning its results.

The first thing to know about the SDSPI interface is that it is designed to work over a shared SPI port. Hence, when the SDSPI controller lowers the CS line to select the SD–Card, the lowered line is only a request to an external controller. Once that external controller grants access to the SPI port to this device, it will tell this controller and processing can continue. If your application does not have any contention for the SPI interface, you may simply wire this grant line high. The controller will still work, although it will wait one clock more than necessary in that case before starting any transaction.

Second, the SDSPI core is completely controlled via the wishbone bus. In particular a command register is used to initiate interaction across the bus. A separate data register is used to provide an argument to the command, and two FIFO registers are used when transferring larger amounts of data to the card. We'll examine each of these interactions in turn.

Writes to the command register (CMD) will initiate actions across the port, whether they be reads from or writes to the card. These writes take the form of sending a 48–bit command to the card. The command sent to the card is taken from the lower 8–bits of the command register, and the argument to the command is taken from the DATA register. The last 8–bits of the command sent to the card are formed from a command CRC byte which the core generates internally. From the perspective of the Wishbone bus, writes to the register will complete immediately, even though the action they initiate will may take much longer to complete. Further, writes made to the CMD register will be silently ignored if the device is already busy.

Reads from the CMD register will always return immediately. In particular, the `busy` bit, as returned by the CMD register, can be used to determine if the interface is busy.

There is one exception to the rule that writes take many clocks to complete, and that is writes which configure the SDSPI port. Internal to the SDSPI port is a configuration register, which determines the speed of the port clock as well as the length of the FIFO (up to 128 samples, or 512–bytes). To read the current speed and FIFO configuration, write an `0x00bf` value to the command register. This will cause the DATA register to be filled with the internal configuration register. Likewise writing a `0x0ff` to the CMD register will cause the current DATA value, or specifically those non-zero parts, to be transferred to the internal configuration register possibly setting the clock divider and/or the FIFO length.

As part of each write to the CMD register, the controller must also be told which type of response to expect from the SDSPI card. Responses can be either R1 (single byte), R1b (single byte, followed by a variable delay), or R1 followed by up to four bytes, such as the R2, R3, or R7 responses. (Expected responses for particular commands may be found in the SD Specifications documents.[1]

Finally, individual commands may or may not use the FIFO. Commands that need use of the FIFO will be specified by the `use_fifo` bit of the CMD register. Commands writing to the device will also set the `fifo_wr` bit of the CMD register, whereas commands simply reading from the fifo will set the `use_fifo` bit alone.

The DATA register is used during these transactions to first provide the argument to the CMD interaction, and second to provide a place to put the R2, R3, or R7 response after the transaction has completed. The register will be set to `0xffffffff` if not set by the response.

Finally, this core supports two separate FIFO's. This allows a program to fill (or read) one FIFO while the second FIFO can be read/written from the SD card. The FIFO internal address will be cleared and reset to the beginning upon any write to the CMD register. After clearing, the FIFO may be written (read) one value at a time. Reading both FIFO's in any interleaved fashion, however, is not allowed as they share a common internal address.

Currently, the core will detect two types of errors in the interface. The first is a CRC error, and the second a timeout error. Either error will set an error bit in the CMD register. Once set, only writing the error bit back to the CMD register will clear it.

Now, if this discussion isn't thoroughly confusing, let's move on to the Operation chapter to see some examples of how this might be used.

[1]This particular interface, and the examples using it, were built using the SD Specifications, Part 1: Physical Layer Simplified Specification, Version 4.10, dated 22 January, 2013.

3. Operation

This chapter will walk through some constants that can be used to simplify interaction with the controller, the logic necessary to start up the card, to read its registers, and then examples of how to read and write sectors from the SD Card using this interface.

3.1 Constants

Since so much of the interface is controlled by the CMD register, it helps to define several constants which can be used when issueing commands to the SD Card. Lets discuss some of these constants.

First, as discussed in the last chapter, the SDSPI core maintains an auxiliary register to handle FIFO length and clock speed. To set this register, we define `SD_SETAUX` to `0x0ff`. Thus, when `SD_SETAUX` is written to the CMD register, the value of the DATA register is transferred to the internal configuration. Likewise, we also define `SD_READAUX` to `0x0bf`. When this value is written to the SD–Card, the internal configuration registers value will be copied to the DATA regiseter.

```
#define  SD_SETAUX   0x0ff
#define  SD_READAUX  0x0bf
```

Second, every command to the SD–Card starts with a single byte. Of that byte, bit-7 must be clear and bit 6 set. For this purpose, we define `SD_CMD` to be `0x040`. Thus, `SD_CMD+0` can be used to send an SD command `CMD0`, and `SD_CMD+1` can be used to send an SD command `CMD1`.

```
#define  SD_CMD  0x040
```

Third, for those commands that will read an SD–Card register, such as those expecting an R2, R3, or R7 response from the card, we define `SD_READREG` to be `0x0200`. Thus, we can send a CMD8 by writing `SD_CMD|SD_READREG` to the port. `#define  SD_READREG  0x0200`

The next thing we'll want to be able to do is use the FIFO. There are two types of commands that use the FIFO, those that read from the card and those that write to the card. Both need the FIFO bit set, so we'll set `SD_FIFO_OP` to `0x0800` to be a read operation from the card, and the same but with the write bit set `SD_WRITEOP` will be set to `0x0c00` to write to the card.

```
#define  SD_FIFO_OP  0x800
#define  SD_WRITEOP  0xc00
```

Finally, we want to be able to choose which FIFO we are using. For this purpose, we define `SD_ALTFIFO` to be `0x01000`. When this bitmask is included in a command, FIFO number one will be used for the command data, otherwise FIFO zero. (Note that this is separate from the DATA register, which is still used for any command argument.)

```
#define  SD_ALTFIFO  0x1000
```

Two other constants are necessary: `SD_BUSY`, set to `0x04000`, which can be used to test when the SD interface is still busy, and `SD_ERROR`, set to `0x08000` which can be used to tell if an error has occurred. Clearing an error may be done by writing SD_ERROR back to the card, but to make things simpler we also create SD_CLEARERR for the same purpose.

```
#define  SD_BUSY       0x4000
#define  SD_ERROR      0x8000
#define  SD_CLEARERR   0x8000
```

The two most important commands, though, are probably going to be those that read and write a sector. For these, we shall define SD_READ_SECTOR and SD_WRITE_SECTOR. As the first is a CMD17 to the card and the second a CMD24, these can be defined as:

```
#define  SD_READ_SECTOR   ((SD_CMD|SD_CLEARERR|SD_FIFO_OP)+17)
#define  SD_WRITE_SECTOR  ((SD_CMD|SD_CLEARERR|SD_WRITEOP)+24)
```

'Or'ing the SD_ALTFIFO mask to either of these commands will cause the interface to read from or write to the alternate FIFO.

As a very last #define, we can define the macro SD_WAIT_WHILE_BUSY to wait until the SD operation completes:

```
#define  SD_WAIT_WHILE_BUSY  while(CMD & SD_BUSY)
```

Alternatively, we could wait for an interrupt instead since the SDSPI core will create an interrupt upon completion. For now, and for this example, we'll ignore interrupts.

3.2 SD–Card Setup

Setting up an SD–Card takes a bit of work. There's a series of commands and interactions that need to take place with the card before the card can be used. You can read about how to do this within the SD–Specification, so we won't repeat the how's or why's here. Instead, let's focus for now on how this interaction can be made to take place using this controller.

The first step in any start up sequence is to clear the card from any prior condition. Hence we wait for the card to be no longer busy (it shouldn't be busy anyway), and we then clear any errors:

```
SD_WAIT_WHILE_BUSY;
CMD SD_CLEARERR;
```

Now that the controller is idle (which it should've been from startup anyway), we can now set up our interface. For this, we'll set our clock rate to 400 KHz. The clock division register, sometimes erroneously called the speed, is found in the lower eight bits of the soft-core configuration register. The actual SPI clock frequency, given this value, will be:

$$f_{\text{SDSPI}} = \frac{f_{\text{CLK}}}{2\left(\texttt{fDIV}+1\right)} \qquad (3.1)$$

Hence, since the XuLA2-LX25 SoC runs at an 80 MHz clock, setting this value to 0x63 sets the SPI clock to 400 kHz.

```
DATA = 0x063;
CMD  = SD_SETAUX;
```

Note that we could have also set the higher order configuration bits to set the size of the FIFO. In particular, the next four bits, bits 8–11, set the FIFO length. Setting these to zero will cause the controller to ignore the change, whereas setting the value to three will set the FIFO length to $4 \cdot 2^3$ bytes, and setting it to seven will set the FIFO length to the nominal $4 \cdot 2^7$ or 512 bytes.

The controller is now ready to send commands to the SD card. The first command to the card is always a command zero, with zero data. This is sometimes called the `GO_IDLE_STATE` command. We then wait for the command to complete:

```
DATA = 0;
CMD  = SD_CMD+0;
SD_WAIT_WHILE_BUSY;
```

The card should now be in its idle state.

The next part of the negotiation tells the card whether or not we are able to handle high capacity cards. For this, we send a command one, `SEND_OP_COND`, with an argument of `0x40000000` to tell it that we are able to support high capacity cards. (An argument of zero would mean that we could not.)

```
DATA = 0x40000000;
CMD  = SD_CMD+1;
SD_WAIT_WHILE_BUSY;
```

Then the card needs to know what voltage it will be run at. We communicate this via a `SEND_IF_COND` command, or CMD8. Since the XuLA2 only operates the card at 3.3V, we tell the card we wish to run at 3.3V in the argument. The last eight bits of the argument, however, are simply to determine whether communication has taken place. We set these bits to `0x0a5`, although they could be anything. The card will echo this value back in the response:

```
DATA = 0x1a5;
CMD  = SD_CMD+8;
SD_WAIT_WHILE_BUSY;
// assert((DATA&0x0ff)==0x01a5);
```

The card will also echo back the voltage range, if it accepts it. Thus, we should receive `0x01a5` as a response.

The card will now try to start up its own internal state machines. This could take a while. We therefore poll the device, and wait for its startup sequence to complete:

```
bool dev_busy = false;
do {
    // CMD55 gives us access to SD specific commands
    DATA = 0;
    CMD = SD_CMD+55;
    SD_WAIT_WHILE_BUSY;

    // Now we can issue the ACMD41, to get the idle
    // status
    DATA = 0x40000000;
    CMD = SD_CMD+41;
    DATA = 0x1a5;
    CMD  = SD_CMD+8;
    SD_WAIT_WHILE_BUSY;
    // The R1 response can be found in the lower 8 bits
    // of the CMD register after the command is complete.
    // Bit 1 of R1 indicates the card hasn't finished its
    // startup
    dev_busy = CMD&1;
} while(dev_busy);
```

3.3 Reading Card Registers

Once the card has started, we can request its operating conditions register, or OCR register as it is called. For this, we issue a `READ_OCR` command, or CMD58 by number. Since this command returns a 32–bit value, we use the `SD_READREG` macro as well:

```
int OCR;

DATA = 0;
CMD  = (SD_READREG|SD_CMD)+58;
SD_WAIT_WHILE_BUSY;
OCR = DATA;
```

When I issue this command on my card, I get a `0xc0ff8000` response telling me that my card can handle between 2.7 and 3.6 Volts, that it is a higher capacity card, and that it has completed its startup sequence.

Now let's switch up to a higher speed, and read the 16–byte Card Specific Data (CSD) register field from the card. First, the switch to a 20 MHz clock and a 16–byte fifo,

```
DATA = 0x0201;
CMD  = SD_SETAUX;
```

Remember that the `0x0200` switches to a FIFO of length 2^2 words, or equivalently $4 \cdot 2^2 = 16$ bytes. Likewise the `0x01` component switches our frequency to 20 MHz. Now we can issue the `SEND_CSD_COND`, or CMD9, command itself. Note that we didn't need to wait for the `SD_SETAUX` command to complete. Further, since this command is going to read from our FIFO, we need to include the `SD_FIFO_OP` part of the command:

```
int CSD[4];
DATA = 0;
CMD  = (SD_FIFO_OP|SD_CMD)+9; SD_WAIT_WHILE_BUSY;
for(int i=0; i<4; i++)
     CSD[i] = FIFO[0];
```

Once the command is complete, we can read the four 32–bit words of the CSD register from the FIFO, as shown above. Alternatively, we could have issued another command first, before reading that FIFO result.

We could read the Card Identification (CID) register as well, but this would've been the same sequence as above, save only that we would've written a `(SD_READREG|SD_CMD)+10)` to CMD.

Reading the STATUS is similar, only the response to the `SEND_STATUS` command is an 8–bit value from an R2 response, not the 32–bit values of the R3 (OCR) or R7 responses. Still, the R2 response is provided in the DATA register, so we only need to send `(SD_READREG|SD_CMD)+13` to the CMD register in order to read its result from the DATA register. The status register will be returned in the top eight bits of the DATA register (the interface still reads 32–bits, even though the other 24 can be ignored), so:

```
int card_status;
DATA = 0;
CMD  = (SD_READREG|SD_CMD)+13; SD_WAIT_WHILE_BUSY;
card_status = DATA>>24;
```

As a final register example, let's read the SD Card Configuration Register (SCR). This register is read in a fashion very simmilar to the CSD register, except that because of its width the FIFO needs to be set for a shorter register width:

```
int SCR[2];
// Set the FIFO length to two words, 2^1.
DATA = 0x0100;
CMD  = SD_SETAUX;
// Issue an ALT command, to get the other command set.
DATA = 0;
CMD  = (SD_CMD)+55;
SD_WAIT_WHILE_BUSY;
// Now get the SCR register.
DATA = 0;
CMD  = (SD_FIFO_OP|SD_CMD)+51;
SD_WAIT_WHILE_BUSY;
for(int i=0; i<2; i++)
     SCR[i] = FIFO[0];
```

3.4 Reading and Writing

For our first example, let's read the boot sector from our card. For this, we set our FIFO back to 128 words (512 bytes), and then issue a read sector command:

```
void read(int sector_num, int *buf) {
     // Set the FIFO length to 128 words, 2^7.
     DATA = 0x0700;
     CMD  = SD_SETAUX;
     // Read from the reqeusted sector
     DATA = sector_num;
     CMD  = SD_READ_SECTOR;
     SD_WAIT_WHILE_BUSY;
     for(int i=0; i<128; i++)
          buf[i] = FIFO[0];
}
```

We could also write to any sector on the card in a very similar fashion:

```
void write(int sector_num, int *buf) {
    // Set the FIFO length to 128 words, 2^7.
    DATA = 0x0700;
    CMD  = SD_SETAUX;
    // Fill the FIFO with our data
    for(int i=0; i<128; i++)
        FIFO[0] = buf[i];
    // Issue the write command
    DATA = sector_num;
    CMD  = SD_WRITE_SECTOR;
    SD_WAIT_WHILE_BUSY;
}
```

As mentioned in the introductory chapter, this interface does not support reading or writing multiple blocks at once. Hence, I expect all interaction using this card controller to be accomplished through these two commands: reading a single sector, and writing to a single sector.

4. Registers

As mentioned in the last two chapters, the SDSPI core has only four registers, and one internal register. These are shown in Tbl. 4.1. The most powerful of these is the command register, CMD,

Name	Address	Width	Access	Description
CMD	`0x00`	32	R/W	SDSPI Command register
DAT	`0x01`	32	R/W	SDSPI return data/argument register
FIFO[0]	`0x02`	32	R/W	FIFO[0] data
FIFO[1]	`0x03`	32	R/W	FIFO[1] data
CONFIG		12	R/W	Internal configuration register

Table 4.1: I/O Peripheral Registers

so we'll spend most of our time discussing that one.

4.1 CMD Register

Writes to the CMD register will cause the device to act, or if the device is already busy then any writes will be ignored. The CMD register itself is composed of several packed bit fields, as shown in Fig. 4.1. Perhaps the most important of these is the R1/CMD field. If bits 7–6 are the two bits 2'b01, then the write is a command and the bottom six bits specify the rest of the 8–bit command identifier. Once the command is complete, these 8–bits represent the R1 response from the device. R1 should be one while the device is still starting, or zero in the case of no error. Further interpretation of this value may be found in the SD–Card Specification.

Of next importance is the *R* field. This specifies the response the controller should expect from the card given the command that was issued to the card. There are three possible values for this field: $2'b00$, meaning the controller should expect an R1 response, $2'b01$, meaning the controller should expect an R1b response, and $2'b10$ meaning the controller should expect an R2/R3/R7 32–bit response.

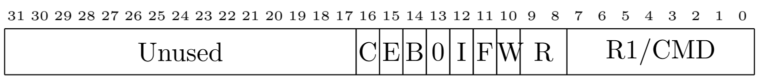

Figure 4.1: CMD Register fields

The *F*, or FIFO, field should be set if the command being given requires the use of the FIFO. *W* should be set at the same time if the controller will be writing to the card from the FIFO, and cleared if the controller will be reading from the card into the FIFO. Finally, *I* specifies which FIFO will be used: 0 for the primary, or 1 for the alternate.

While the command is running, the BUSY or *B* bit will be set.

Once the command has completed, the *E* bit may be set if either the command timed out, or a CRC error was noted while reading from the card. (To detect a CRC error when writing to the card, check the R1 response according to the SD Card specification.) Errors may be cleared by writing a 1 to this bit. Until such time as an error is cleared, the error condition will simply persist.

4.2 DATA Register

Compared to the CMD register, the DATA register is quite simple. Like the CMD register, the DATA register may only be written when the interface is idle. When issuing a command to the device, the 32–bit argument for the command is taken from the DATA register. When reading the results of a device command, the DATA register will contain the R2 response in the upper 8–bits, or an R3 or R7 response in the full 32–bits.

4.3 FIFO Registers

The SDSPI controller maintains two 128 word (512 byte) FIFOs. Reads from the card will read data into one of the two FIFO's, whereas writes to the card will write data out from one of the FIFO's. Which FIFO the card uses is determined by the *I* bit in the CMD register (above).

Further, upon any write to the CMD register, the FIFO address will be set to point to the beginning of the FIFO.

The purpose of the FIFO's is to allow one to issue a command to read into one FIFO, then when that command is complete to read into a second FIFO. While the second command is ongoing, a CPU or DMA may read the data out of the first FIFO and place it wherever into memory. Then, when the second read is complete, a third read may be issued into the first buffer while the data is read out of the second and so forth.

This interleaving approach, sometimes called ping-pong buffering, can also be used for writing: Write into one FIFO, issue a write command, write into the second FIFO, wait for the first write command to complete, issue a second write command, and so forth.

One item to note before closing: there is only one internal address register when accessing the FIFO from the wishbone bus. Attempts to read from or write to either FIFO from the wishbone bus will increment this address register. Interleaved read, or write attempts, such as reading one item from FIFO[0] and writing another item to FIFO[1], will each increment the internal address pointer so that the result is likely to be undesirable. For this reason, it is recommended that only one FIFO be read from or written by the wishbone bus at a time.

4.4 CONFIG Register

The CONFIG register controls the SPI clock rate and the FIFO size. Specifically, with regards to the FIFO size, it controls how many bytes will be written into the FIFO (which is really of a fixed

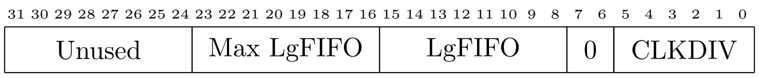

Figure 4.2: CONFIG Register fields

size) before the expecting a CRC, or equivalently how many bytes to read out of the FIFO before adding a CRC. The fields of this register are shown in Fig. 4.2.

The CLKDIV field sets a divisor from the current clock to create a SPI clock. The minimum value of this field is '1', corresponding to dividing the input clock by '4'. As discussed earlier, the input clock will be divided by twice this field plus one. Hence, setting this field to one will cause the original clock to be divided by $2(1 + \texttt{CLKDIV})$ or 4. Thus an 80 MHz input clock will become a 20 MHz SPI clock. Setting this value to zero will cause the set to be ignored.

The LgFIFO field sets the log, base two, of the FIFO size. The actual FIFO size used will be 2^{LGFIFO} words. The maximum size the device will support is returned by the `Max LgFIFO` field, which is currently set to 7 for a 128 word (512 byte) FIFO.

To set the CONFIG register, first set the DATA register to the new config value (or zero for the fields that will not change), and then write `0x0ff` to the CMD register. Likewise, to read the CONFIG register write a `0x0bf` to the CMD register and read the CONFIG register from the DATA register.

5. Wishbone Datasheet

Tbl. 5.1 is required by the wishbone specification, and so it is included here. Note that all wishbone operations may be pipelined, to include FIFO operations, for speed.

Description	Specification
Revision level of wishbone	WB B4 spec
Type of interface	Slave, (Block/pipelined) Read/Write
Port size	32–bit
Port granularity	32–bit
Maximum Operand Size	32–bit
Data transfer ordering	Big Endian
Clock constraints	(See below)
Signal Names	

Signal Name	Wishbone Equivalent
`i_clk`	`CLK_I`
`i_wb_cyc`	`CYC_I`
`i_wb_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 Slave Datasheet

The particular constraint on the clock is not really a wishbone constraint, but rather an SD–Card constraint. Not all cards can handle clocks faster than 25 MHz. For this reason, the wishbone clock, which forms the master clock for this entire controller, must be divided down so that the SPI clock is within the limits the card can handle.

6. Clocks

This core is based upon the XuLA2-LX25 SoC design. The XuLA2 development board contains one external 12 MHz clock, which is internally boosted within the SoC to 80 MHz. This clock is divided by a programmable divider, often set to four, to create the 20 MHz clock used to drive the device. (Maximum device speed is usually 25MHz, although some devices can run at 50 MHz.) This controller does not support setting the SPI clock frequency any faster than one quarter of the input clock rate.

7. I/O Ports

Table. 7.1 lists all of the input and output ports to this core. You may notice these inputs and

Port	Width	Direction	Description
i_clk	1	Input	Clock
i_wb_cyc	1	Output	Wishbone bus cycle active
i_wb_stb	1	Output	Wishbone Strobe, true one clock only for each interaction
i_wb_we	1	Output	Wishbone Write-Enable line
i_wb_addr	2	Output	Selects our I/O register
i_wb_data	32	Output	Incoming wishbone bus data
o_wb_ack	1	Output	Acknowledge a WB request, always true one clock after the request
o_wb_stall	1	Output	Always zero
o_wb_data	32	Output	32–bit wishbone data response
o_cs_n	1	Output	Chip–select and SPI request line
o_sck	1	Output	SD Card clock
o_mosi	1	Output	Output data wire to the SD Card
i_miso	1	Input	Input data wire from the SD Card
o_int	1	Output	An interrupt line to the CPU controller
i_bus_grant	1	Input	True if the SDSPI controller is controlling the bus
o_debug	32	Output	See Verilog for details

Table 7.1: List of IO ports

outputs are divided into sections: the master clock, the wishbone bus, the SPI interface to the card, and three other wires. Of these, the last two chapters discussed the wishbone bus interface and the clock. The SPI interface should be fairly straightforward, so we'll move on and discuss the other three wires.

This controller supports an interrupt line, `o_int`. Upon completion of any operation, when the SPI chip select line is deactivated (raised high), `o_int` will be strobed for one cycle. It is up to the logic using this chip to catch and use that interrupt line or ignore it. In particular, it is possible to use that interrupt line to trigger a DMA service to move data in or out of the FIFO, although the details of that are beyond this discussion here.

The second wire, i_bus_grant, exists because this controller is expected to operate together with other SPI controllers that may also wish to drive the same three wires (o_sck and o_mosi to the device, and i_miso from the device). When the controller therefore lowers the o_cs_n line to make it active, it then waits for i_bus_grant to go high. This is its signal, from the outside world, that its chip has been selected and that it is now driving the o_sck and o_mosi pins. Should you not need this in your environment, you can simply leave this line wired high.

The final bus of 32–wires, o_debug, is defined internally and used when/if necessary to debug the core and watch what is going on within it. These wires may be left unconnected in most implementations, as they are not necessary for using the actually controller.