SD/eMMC/MMC card host

SD Host Pack : VHDL synthesizable cores Technical Reference Manual

Written by:

John Clayton
Klugwhallah FPGA design team

June 26, 2017

1 List of Acronymns

FPGA	Field Programmable Gate Array
JTAG	Joint Test Access Group
MMC	Multi-Media Card
SD	Secure Digital
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

2 Introduction

This document provides a description of the interface signals, internal structure and registers present in the synthesizable cores within the VHDL package file named “sd_host_pack.vhd” Some of the cores are intended for use at lower levels of a design, in hierarchical fashion. All cores in the VHDL package are synthesizable and have been tested via simulation and in hardware using a Xilinx “ARTY” Artix 7 FPGA development board.

3 Description of Cores

3.1 Background

The “sd_host_pack.vhd” VHDL package includes three entities, two of which are used inside the third. Specifically, there is a command host (“sd_cmd_host”) and a data host (“sd_data_8bit_host”) which are each instantiated inside the higher level entity “sd_controller_8bit_bram,” which is an SD/MMC controller core. The use of the term SD, and the absence of the term MMC, in the naming of signals within this set of cores is not meant to imply that MMC cards are not supported. It is merely a result of the origins of this set of VHDL entities. These entities have their roots in some modules written in Verilog, by Marek Czerski, and posted at www.opencores.org. After contacting Marek via email, John Clayton was added as a project maintainer of the sd_card_controller project, and it was concluded that John Clayton would work to proofread and correct the project documentation, as well as complete a translation of the original Verilog code into VHDL. This work of translation was done prior to the beginning of work on these cores for a commercial project, but no testing had been done. Since the original project was focused on SD cards only, effect of the original naming conventions is still manifest. Nevertheless, the cores in this VHDL package file were expanded to include support for 8-bit wide data bus, thereby enabling MMC support as well. And they have now been simulated and tested. The other two related VHDL packages, “sd_card_pack” and “mmc_test_pack” were created “from scratch” entirely by John Clayton, and no Verilog versions of those entities has yet been created. The main specification document used during development was the JEDEC document JESD84-A44 titled “Embedded MultiMediaCard(e•MMC)” (MMCA, 4.4) version.

3.2 Summary of Cores In Package

The cores present in the package file “sd_host_pack.vhd” are shown in Table 1. The controller core was written specifically to enable hardware testing of an MMC emulator core from “sd_card_pack.” The controller uses one instance of each of the other two cores.

Name	Description
sd_cmd_host	SD/MMC Card command transceiver
sd_data_8bit_host	SD/MMC Card data packet transceiver
sd_controller_8bit_bram	SD/MMC Card host with port for BRAM storage

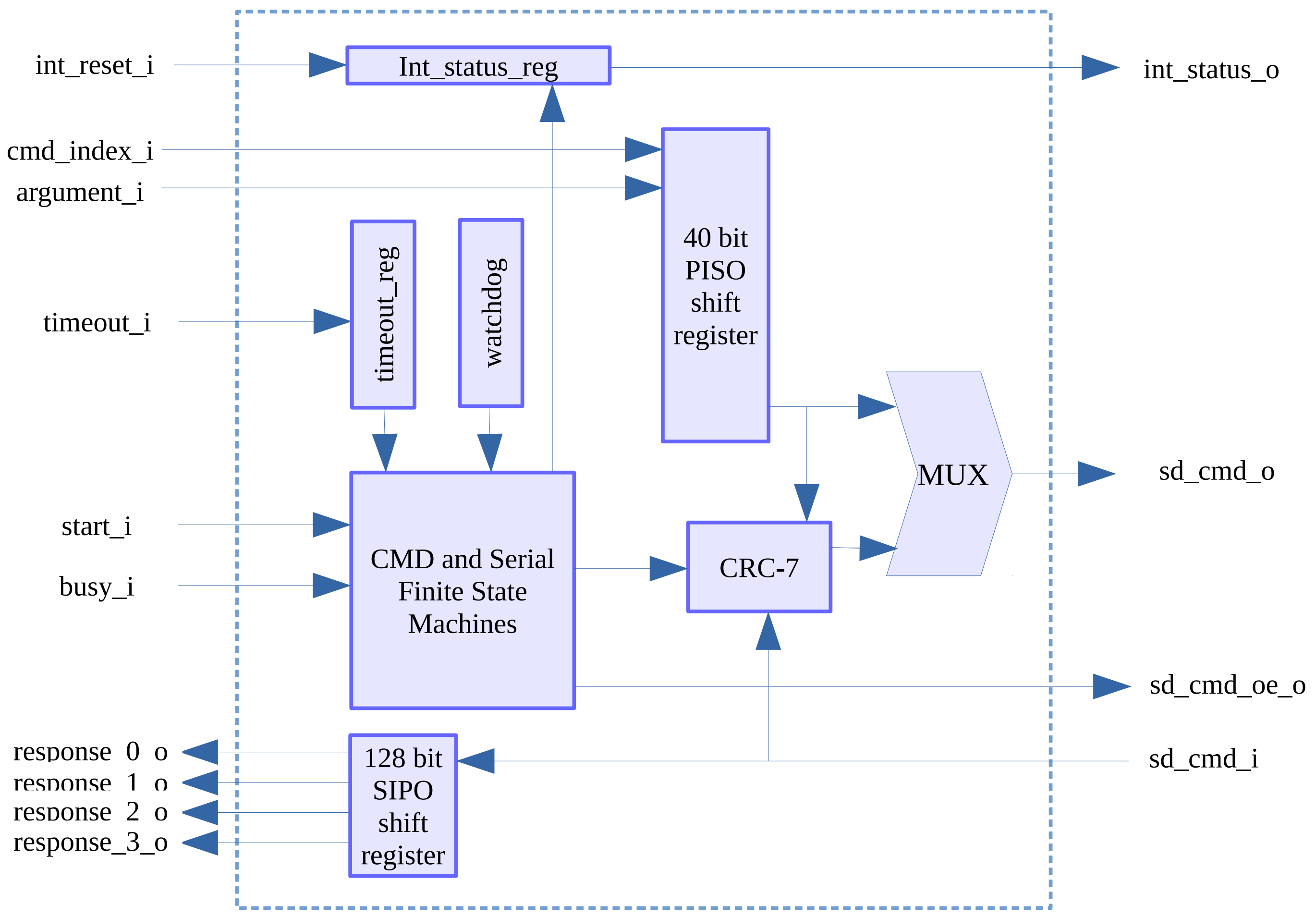
Table 1: SD Host Pack Cores

In addition to the cores listed in Table 1, the “sd_host_pack.vhd” file also defines some constants at the VHDL package level. These constants determine the bit-width of the counters used for keeping track of data sectors, and also there are constants which define different interrupts. The interrupts are useful when interfacing the sd_controller_8bit_bram to a processor, or when writing a software driver for that purpose.

3.3 sd_cmd_host

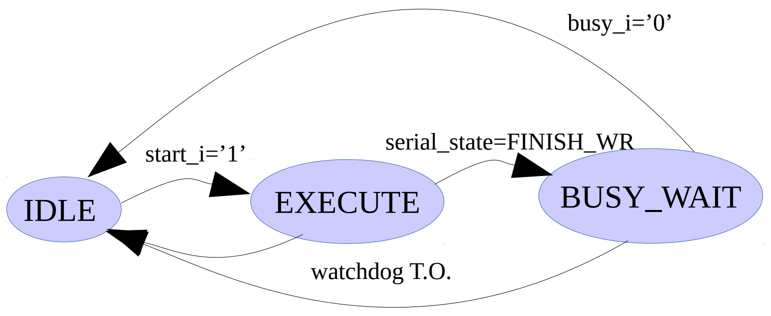
This core combines what was originally the “sd_cmd_master” and the “sd_cmd_serial_host” from the opencores.org VHDL code, in order to simplify by reducing the total number of different cores. The reason for combining them is that eight interface signals were shared only between those two modules when they were instantiated, and the constituent modules are not needed individually. This core runs entirely within the sd_clk_i clock domain.

The sd_cmd_host internal structure is shown in this block diagram:

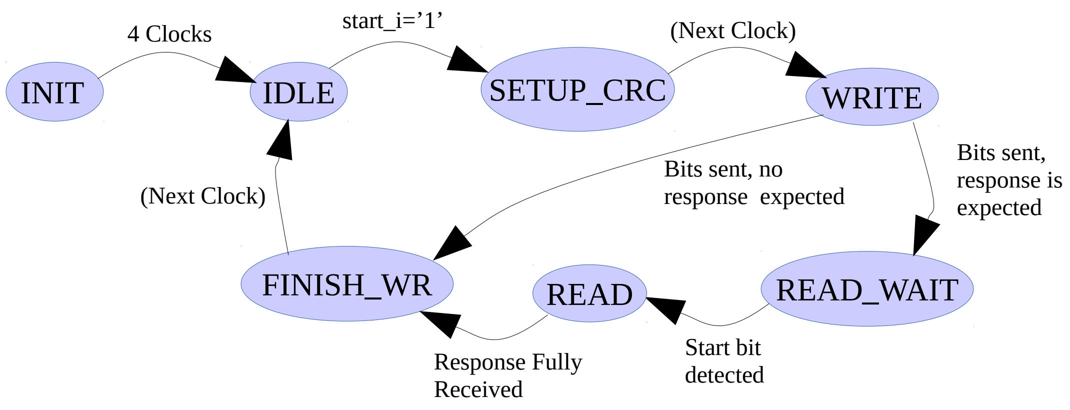


When start_i is pulsed high, the 16 bit timeout_i value is latched. Also, the 6 bit interrupt_status_o, the 128 bit response register, the 16 bit watchdog and the CRC-7 are all cleared. The Finite State Machines begin operating, sending out the requested command, and then looking for the expected reply. For cmd_index_i of 2, 9 or 10, the expected response is set to 128 bits. All others set the expected response length to 40 bits. The upper two bits of the 40 bit command register are set to "10", meaning a command from the host to a card. The next six bits are the cmd_index_i value, and the 32 least significant bits are set to the argument_i value. Once all 40 bits of the cmd PISO shift register are sent out, the mux switches so that the CRC-7 value is sent out, followed by a '1' stop bit. Then, receive operations begin.

There are two separate state machines, the “CMD” FSM and the “Serial” FSM. The state transition diagram for the CMD FSM is very simple:



During the CMD FSM’s EXECUTE state is when all of the Serial FSM action occurs:



During the CMD FSM state “EXECUTE” or “BUSY_WAIT” states, various interrupt flags can be set, as determined by the exit conditions. The bit positions are shown in Illustration 1, and the conditions are listed in Table 2.

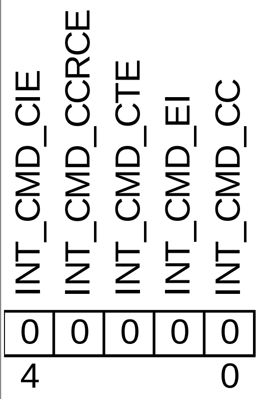


Illustration 1: sd_cmd_host interrupt status bits

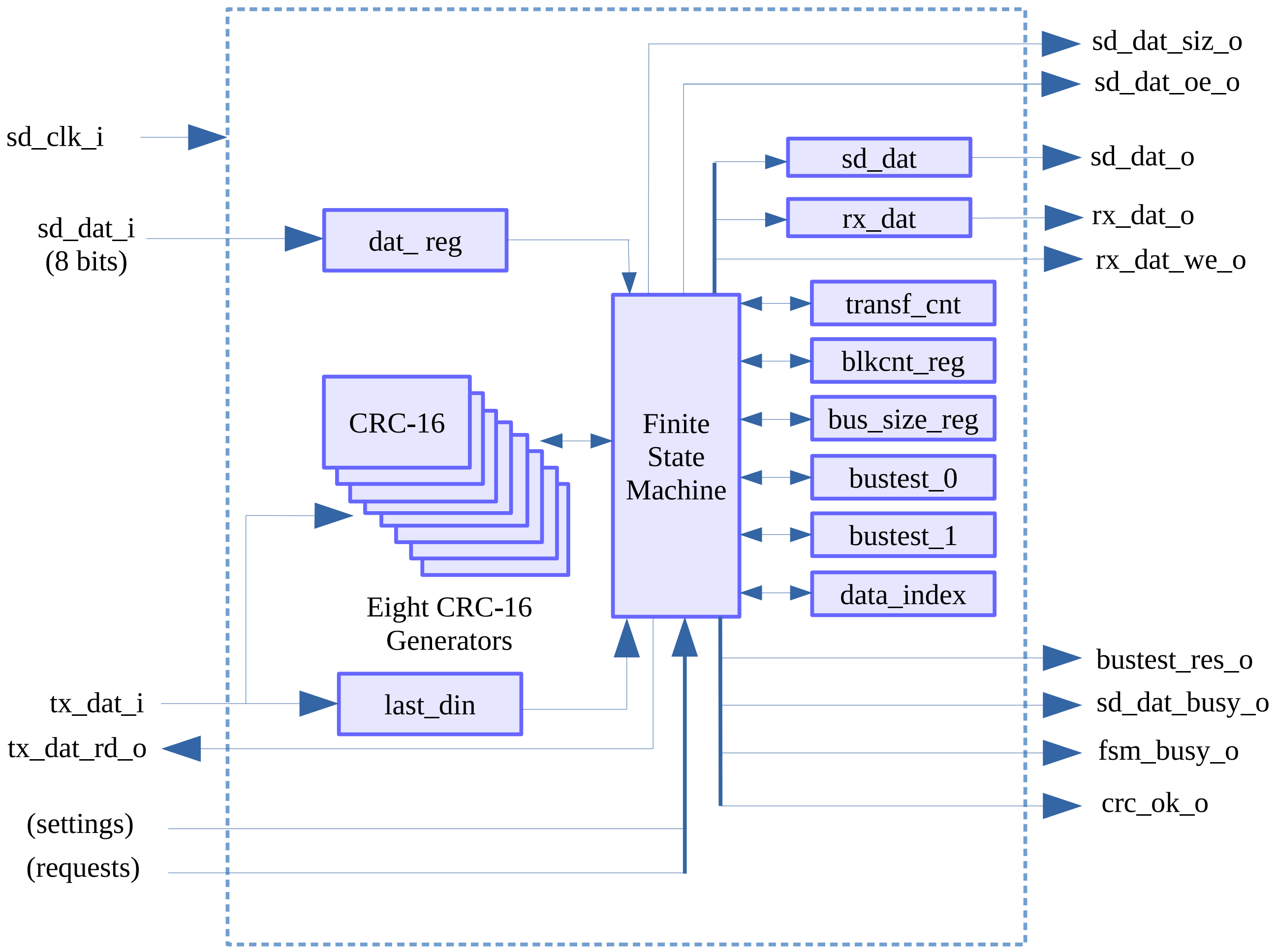
Condition	Value	Result
start_i='1'	`00000b`	All bits cleared
Command Timeout	`00110b`	Error Indicator + CMD timeout Error bit, response set to 0x55555555555555555555555555555555
Response has CRC error	`01011b`	Error Indicator + CMD CRC Error bit + CMD Complete
Response index does not match command index	`10011b`	Error Indicator + CMD Index Error bit + CMD Complete
No Error	`00001b`	CMD Complete

Table 2: CMD host interrupt status conditions

3.4 sd_data_8bit_host

This core handles data transfers from the host perspective. That is to say, it executes commands to read or write blocks of data on the SD/MMC data bus, checking and populating the CRC-16 field for each lane or data line used on the data bus, as needed. The size of the block, in bytes, is configurable by the input blksize_i. As a point of reference, the customary block size for SD/MMC is 512 bytes, although the blksize_i input is 12 bits wide, to accommodate blocks of up to 4096 bytes. The number of blocks to be transferred in sequence is specified by the blkcnt_i input, which is 16 bits wide, allowing for up to 65535 blocks to be treated in one single command. Setting blkcnt_i greater than 1 will cause multiple blocks to be transferred. However, setting it to zero will not prevent the transfer of a block, and is exactly the same as setting it to one.

The internal structure of sd_data_8bit_host is shown in the following block diagram.



The settings and request types for this core are listed in Table 3 and Table 5, respectively.

Signal Name	Width	Purpose
blksize_i	`12 bits`	Sets size of data block (512 bytes is common)
bus_size_i	`2 bits`	Sets width of SD/MMC data transfers
blkcnt_o	`16 bits`	Sets number of blocks to transfer per read/write request

Table 3: sd_data_8bit_host setting inputs

This core works as part of a larger core, called “sd_controller_8bit_bram,” which handles all the handshaking and assertions of request inputs to this core. Because the higher level entity performs all the handshaking, and this core is not intended to be instantiated without that higher level entity, the details of how requests are made and how transfers are stopped are best seen by inspecting the state machine of the sd_controller_8bit_bram core that instantiates this one.

One important task handled by the sd_data_8bit_host, is sizing the data to fit the currently selected width of SD/MMC data bus being used. For SD cards, the bus can only be one bit wide, or four bits wide. For MMC, a full eight bit width is added into the mix. The number of cycles needed to transfer a given block over the bus is also inversely related to the size of the bus, and the calculations for it are handled automatically by this core. The bus size setting is selected via the two bit bus_size_i input, with settings as shown in Table 4. The setting of "11" is technically reserved, but it produces behavior just the same as "00"

bus_size_i	Bus Size Selection	Transfers for 512 byte block
00b	One bit per transfer	2048
01b	Four bits per transfer	1024
10b	Eight bits per transfer	512
11b	One bit per transfer	2048

Table 4: Data transfer bus size selections

The sd_data_8bit_host core has five different discrete command request inputs, which are handled whenever the unit is in the IDLE state. When already processing a command, the command request inputs are ignored, and no internal latching, buffering or queueing of requests is done. The command request inputs are summarized in Table 5.

Name	Function
d_stop_i	Aborts read or write
d_read_i	Read data
d_write_i	Write data
bustest_w_i	Write Bustest Pattern
bustest_r_i	Read Bustest Reply

Table 5: Command Request Signals

Note that the d_stop_i only aborts an active read or write transaction. If the d_stop_i input is asserted at the same time as d_read_i, for example, the unit will begin a read sequence. If the d_stop_i input continues to be asserted, then the read sequence will be aborted instantly. The d_stop_i input has no effect on bustest operations in progress, only on reads or writes.

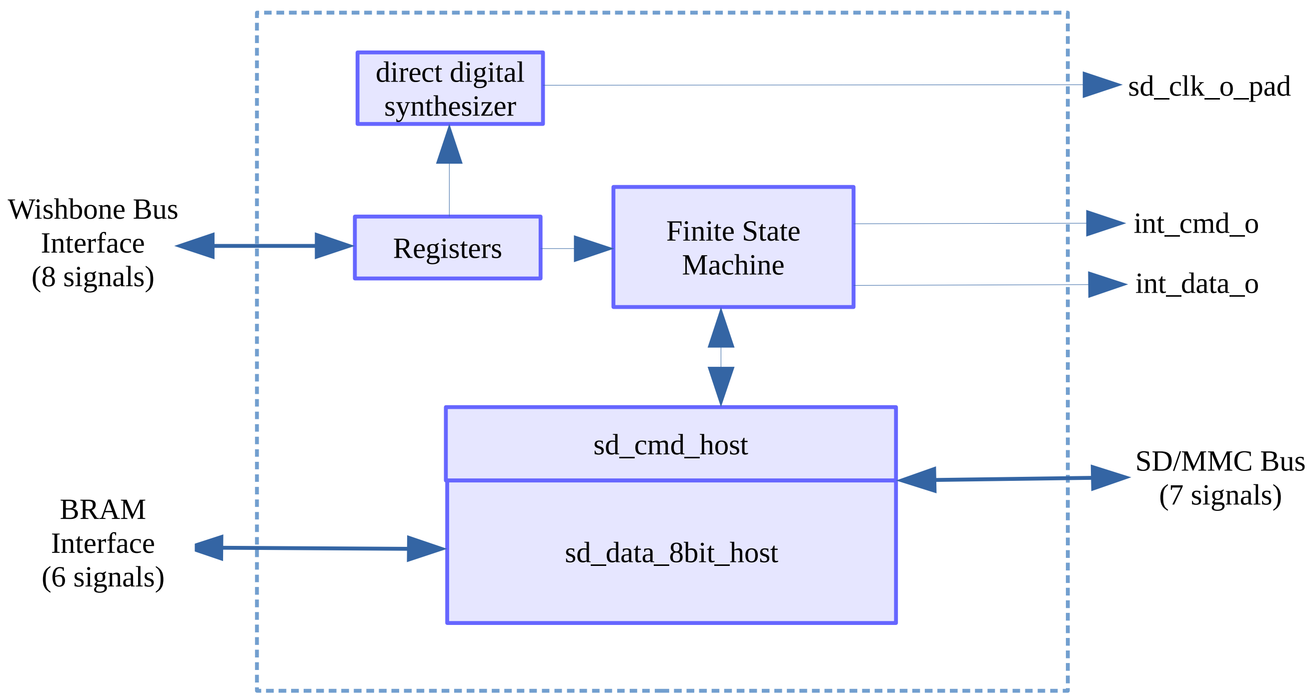
It is instructive to note that this unit is purely a “slave” to the controller core. Only one transfer can occur at any given time over the SD/MMC data bus. This data host core handles splitting apart bytes being written to an SD/MMC bus which is less than eight bits, and joining together bit-slices from a bus less than eight bits. Incidental to that task, the sd_data_8bit_host keeps track of the number of cycles left, the CRC to be checked or appended, the number of bytes remaining in the current block, and the number of blocks to be transferred before the task is finished. There are also bus test patterns that can be issued and/or checked by this unit. The purpose of the bus test is to determine how many data bits are in use on the SD/MMC bus. The maximum number of data bits for SD is four, but for MMC it is eight.

The logic within this core runs entirely inside the sd_clk clock domain. The controller implements measures to allow the data to cross between the sd_clk domain and a different clock domain.

3.5 sd_controller_8bit_bram

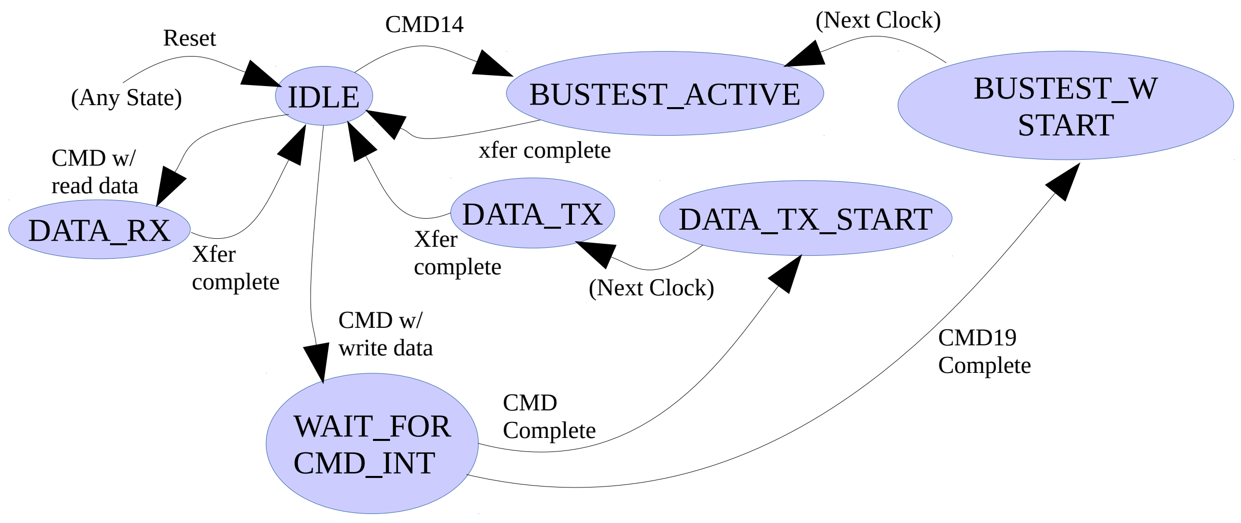
This core is memory mapped onto a parallel “wishbone” type bus, and provides the capability to send SD/MMC commands and receive the responses. The unit coordinates the transfer of data to and from an SD/MMC bus as part of the commanded operations. It also provides a direct interface to an area of storage RAM used for holding send/receive data. In addition to the register based interface, two interrupt outputs are provided, one for command and one for data. This core can be used by a software driver to implement an SD/MMC interface, or it can be part of a hardware based tester, such as the “mmc_tester” core in the mmc_test_pack.vhd package.

The structure of this core is shown in the following block diagram:



For simplicity, the register connections, and the edge detectors used for generating interrupts are not shown in this block diagram. There are two main clock domains in this core. The first is the SD/MMC cardbus clock domain, which is used by the "data master" finite state machine, the sd_cmd_host and the sd_data_8bit_host. The second clock domain is the Wishbone bus clock domain, used for the DDS and for register writes.

A state transition diagram of the "data master" finite state machine helps as an aid to understanding how it works:



The sd_controller_8bit_bram registers are all summarized in Table 6. These registers are further explained using diagrams inside this document.

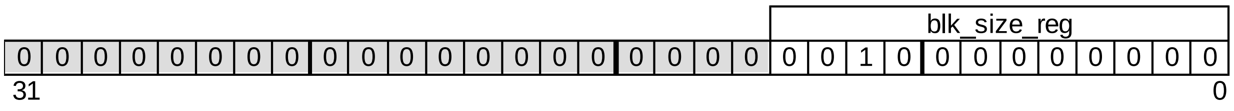
4-bit address	Name	Function
0x0	blk_size_reg	Set the size of data blocks, in bytes
0x1	blk_count_reg	Set the number of data blocks to transfer
0x2	cmd_index_reg	Set which command to use
0x3	argument_reg	Set the payload contents of the command
0x4	resp_0_reg	Response bytes [3..0]
0x5	resp_1_reg	Response bytes [7..4]
0x6	resp_2_reg	Response bytes [11..8]
0x7	resp_3_reg	Response bytes [15..12]
0x8	settings	Set bus size, data reset, stop, timeout
0x9	sd_freq_reg	Set the SD/MMC clock frequency
0xA	cmd_int_status	Status of command interrupt bit
0xB	cmd_int_enable	Enable command interrupt
0xC	data_int_status	Status of data interrupt bit
0xD	data_int_enable	Enable data interrupt
0xE	dma_adr_reg	Starting address for BRAM read/write
0xF	(RESERVED)	(RESERVED)

Table 6: mmc_tester memory map full register list

For each of the registers implemented, further explanation and a register diagram are given. Note that in this core, the parallel system bus is a 4-bit address bus selecting 32-bit registers on a 32-bit data bus, with a select line being used to activate the block of 16 register addresses. Addresses in the system are 32-bit "double word" addresses, meaning that each address selects a specific 32-bit data word location, and there are no byte-enables or other byte addressing constructs. the address given for each register is the relative offset from the base address used to generate the 16-register block select signal.

Register 0x0 : Block Size Register

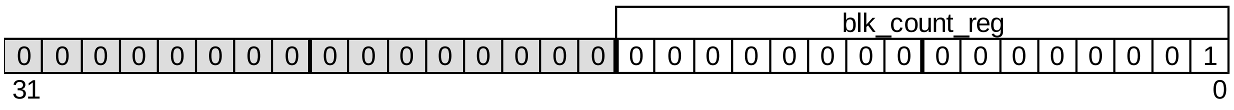
Addr: 0x0 Access: Read/Write



This register allows setting the size, in bytes, of a block of data to be transferred. Sometimes this quantity is also referred to as the “sector size.” In most SD/MMC situations, the block size is 512 bytes, so that is the value in this register by default.

Register 0x1 : Block Count Register

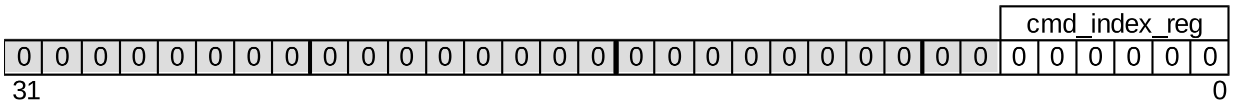
Addr: 0x1 Access: Read/Write



Address 0x1 is the block count register. This register sets the number of blocks of data, each containing the block size bytes (Register 0x0), which are to be transferred in a given transaction. This register has been sized so that up to 65535 sectors or “blocks” of data can be included in a single transfer operation.

Register 0x2 : Command Index Register

Addr: 0x2 Access: Read/Write



This register contains the command number to be sent out when writing register 0x3 to request a command start. See the SD/MMC specification for the complete listing of commands. Some more commonly used commands are: CMD0, CMD1, CMD3, CMD6, CMD7, CMD12, CMD13, CMD17, CMD18, CMD25.

Register 0x3 : Argument Register

Addr: 0x3 Access: Read/Write

argument_reg																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31																															0

Each command sent out on the SD/MMC command line has an associated 32-bit “argument” that, for many commands, further defines what the command is intended to do. This register holds the argument for the next command to be issued. Morevoer, writing the argument into this register has the effect of actually requesting the hardware to start sending the command. This was chosen to make it easy to send a given command. All that needs to be done is to program the command index and the argument, and “voila” the command goes out.

Register 0x4 – 0x7 : Response Registers

These registers are read only, containing the most recently received response from the addressed SD/MMC device. There are five types of responses, known as R1, R2, R3, R4 and R5. Except for R2, all responses are 48 bits long, including a 32-bit “payload” after all the headers and CRC are stripped away. Therefore all reponses, except R2, are presented in the first register. For the R2 responses, the payload part of the message is 128 bits long, so that four complete 32-bit registers are occupied by it.

The first, and most commonly used, response register:

Addr: 0x4 Access: Read Only

resp_0_reg																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31																															0

The second reponse register:

Addr: 0x5 Access: Read Only

resp_1_reg																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31																															0

The third response register:

Addr: 0x6 Access: Read Only

resp_2_reg																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31																															0

The fourth response register:

Addr: 0x7 Access: Read Only

resp_3_reg																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31																															0

Register 0x8 : Control and Status Register

Addr: 0x8 Access: Read/Write (*Fields are read only)

timeout_reg																					bustest_res*						sw_rst_reg			bus_siz_reg	
0	0	0	0	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31																															0

This register contains several fields, described here.

Bits [1..0] = bus_siz_reg This field selects which setting to use for the width of the SD/MMC data bus. Using the 8-bit width is an MMC function, and is not allowed for SD cards, since SD cards only have four data bits present. The settings for this field are described in Table 7.

bus_siz_reg	Meaning
`0x0`	One bit data bus used.
`0x1`	Four bit data bus used.
`0x2`	Eight bit data bus used.
`0x3`	One bit data bus used.

Table 7: Control register - bus size field

Bits [3..2] = “00” (Reserved)

Bit [4] = sw_rst_reg. Setting this bit causes all internal logic and state machines, outside of the register block itself, to be held in reset. The bit must be explicitly cleared for the SD controller to emerge from reset.

Bits [7..5] = "000" (Reserved)

Bits [10..8] = Bus test result. This field is read-only, and it provides indication of the most recent bus test read operation. The interpretation of this register field is given in Table 8.

bustest_res	Meaning
`0x0`	Bus test has not been run.
`0x1`	One bit data bus detected.
`0x2`	Four bit data bus detected.
`0x3`	Eight bit data bus detected.
`0x4`	Unexpected result. (Should not happen.)

Table 8: Control register - bus test result field

Bits [15..11] = “00000” (Reserved)

Bits [31..16] = Command timeout field. The number in this register field is the number of clock cycles allowed for sending a command, and receiving back the complete response, including waiting for the card to come out of “busy” indication. The default value for this field is 1000 clock cycles of the SD/MMC clock. If a timeout occurs while waiting for a response from the SD/MMC device, the response registers are all set to contain 0x55555555, and the “command timeout error” (INT_CMD_CTE) and “error indicator” (INT_CMD_EI) interrupt bits are set. If, on the other hand, a timeout occurs after the response is fully received, such as when an SD/MMC device enters an extended “busy” state, then the response registers are left alone with their response contents intact, and the “command timeout error” (INT_CMD_CTE) and “error indicator” (INT_CMD_EI) interrupt bits are set.

Register 0x9 : SD/MMC clock frequency register

Addr: 0x9 Access: Read/Write

sd_freq_reg																															
0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	1	0	0	1	0	0	1	1	0	1	1	1	0	1
31																															0

This register provides a way to set the SD/MMC clock frequency being generated by this core. A direct digital squarewave generator is used to create the clock signal. The generated clock can be set to any desired frequency, at any time, according to the following formula:

Frequency = (sd_freq_reg/Fsys_clk) * 2^32

The default value of sd_freq_reg, 0x010624DE is set for an SD/MMC clock frequency of 400 kHz, using a system clock frequency of 100 MHz.

Register 0xA : Command interrupt status register

Addr: 0xA Access: Readable, Write clears all

																											INT_CMD_CIE	INT_CMD_CCRCE	INT_CMD_CTE	INT_CMD_EI	INT_CMD_CC
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31																															0

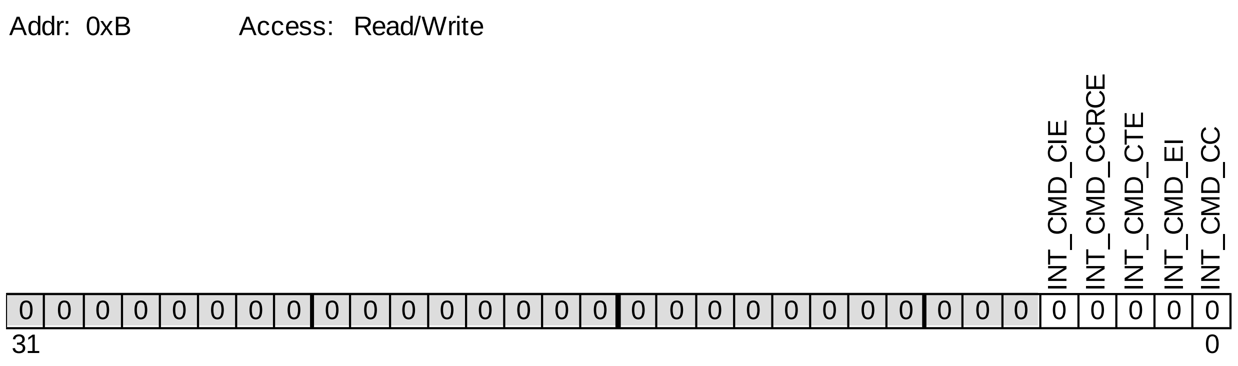
This register shows the current status of interrupts related to the last SD/MMC command operation. The presence of a condition is indicated by the associated bit being set.

Bit	Name	Meaning
0	`INT_CMD_CC`	The command is completed.
1	`INT_CMD_EI`	An Exception is Indicated.
2	`INT_CMD_CTE`	Command Timeout Exception.
3	`INT_CMD_CCRCE`	Command CRC Exception.
4	`INT_CMD_CIE`	Command Index Exception.

Table 9: Command interrupt status bits

In each case, if an exception occurs, two bits are set: One is the INT_CMD_EI, and the other is the particular event or exception that occurred. Writing this register clears all of the bits.

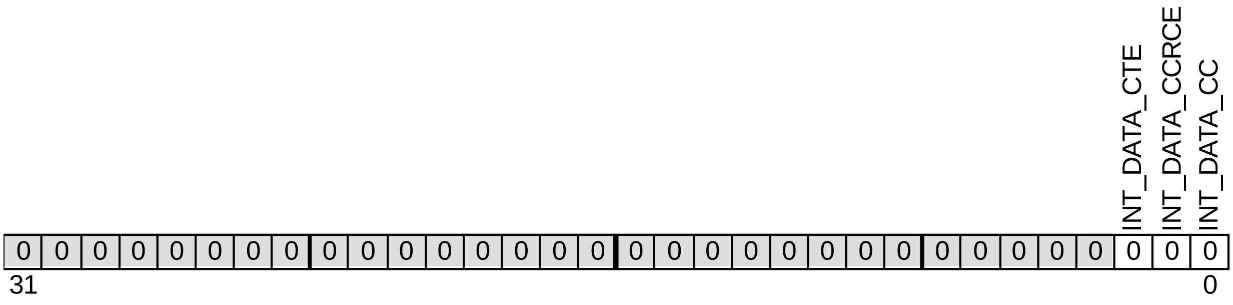
Register 0xB : Command interrupt enable register



The bits in this register match exactly with the bits in register 0xA. The purpose of this register is to provide a capability to enable interrupts to be observed. In other words, setting this register to 11111b enables all the interrupts to be reported, setting it to 00000b disables all interrupts from being reported. Note: the operation of this interrupt enable register does not prevent the interrupt occurrences from being latched internally, it merely acts as an “AND” function with the internally latched interrupt bits. This means that it is possible for an interrupt to have occurred in the past, but to have been disabled, and to suddenly appear if the enable bit is set without first clearing the interrupts by writing to register 0xA.

Register 0xC : Data interrupt status register

Addr: 0xC Access: Readable, Write clears all



This register shows the current status of interrupts related to the last SD/MMC data operation. The presence of a condition is indicated by the associated bit being set.

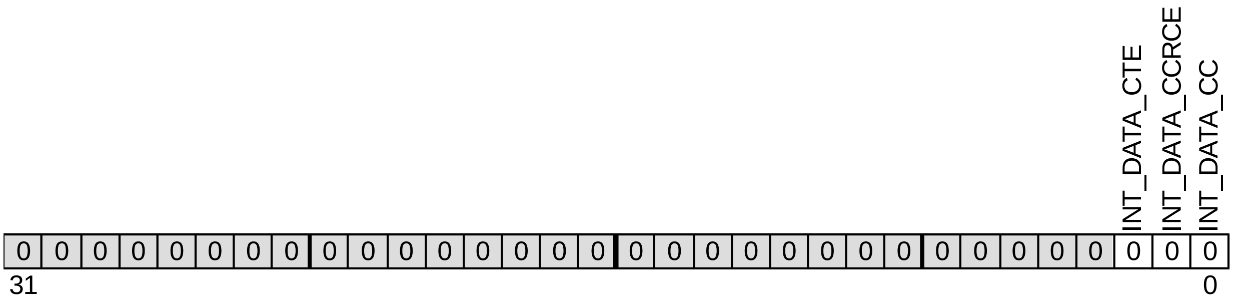
Bit	Name	Meaning
0	`INT_DATA_CC`	The data transfer is completed.
1	`INT_DATA_CCRCE`	Data CRC exception.
2	`INT_DATA_CFE`	Vestigial data exception (not implemented).

Table 10: Data interrupt status bits

The bit labeled INT_DATA_CFE is not implemented, and its meaning is not clear. It is vestigial in the sense that it was carried over into this VHDL code from the work of the original authors. Writing this register clears all of the bits.

Register 0xD : Data interrupt enable register

Addr: 0xD Access: Read/Write



The bits in this register match exactly with the bits in register 0xC. The purpose of this register is to provide a capability to enable interrupts to be observed. In other words, setting this register to 111b enables all the interrupts to be reported, setting it to 000b disables all interrupts from being reported. Note: the operation of this interrupt enable register does not prevent the interrupt occurrences from being latched internally, it merely acts as an "AND" function with the internally latched interrupt bits. This means that it is possible for an interrupt to have occurred in the past, but to have been disabled, and to suddenly appear if the enable bit is set without first clearing the interrupts by writing to register 0xC.

Register 0xD : DMA address register

Addr: 0xE Access: Read/Write

dma_addr_reg																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31																															0

The presence of the acronym "DMA" in the name of this register merely indicates a "Direct Memory Access" in the simplest sense: It is an address used for reading data from and writing data to memory, directly. In this sense it can be thought of as a read/write pointer or index register. Any attached RAM is accessed starting at the address in this register. The address contained in this register is added to an offset counter value, so that this register's contents do not increment as the RAM access progresses. The dma_addr_reg value is therefore a "base address" at which data transfers originate. The data from an earlier transfer will be overwritten by later transfers if the dma_addr_reg value is not changed in between transfers.