SD Card Controller IP Specification

Marek Czerski

Saturday 25th June, 2022

List of Figures

List of Tables

1 Introduction

This document describes the multimedia card (MMC) / secure digital (SD) card controller ip core - *Wishbone SD Card Controller IP Core*.

1.1 Purpose of the IP core

The *Wishbone SD Card Controller IP Core* is MMC/SD communication controller designed to be used in System-on-Chip (img. 1). IP core provides simple interface for any MCU with Wishbone bus. The communication between the MMC/SD card controller and MMC/SD card is performed according to the MMC/SD protocol.

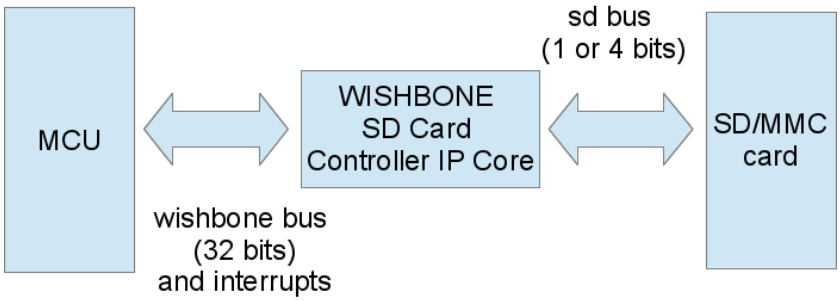


Figure 1: SoC with SD Card IP core

1.2 Features

The MMC/SD card controller provides following features:

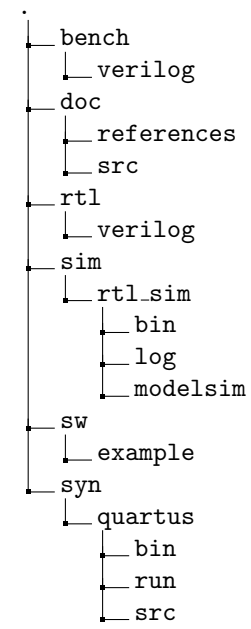
- 1- or 4-bit MMC/SD mode (does not support SPI mode),
- 32-bit Wishbone interface,
- DMA engine for data transfers,
- Interrupt generation on completion of data and command transactions,
- Configurable data transfer block size,
- Support for any command code (including multiple data block transfer),
- Support for R1, R1b, R2(136-bit), R3, R6 and R7 responses.

2 Usage

This chapter describes usage of the IP core.

2.1 Directory structure

Wishbone SD Card Controller IP Core comes with following directory structure:



`bench/verilog` - verilog testbench sources,

`doc` - documentation files,

`doc/src` - documentation LaTeXsources,

`rtl/verilog` - ip core verilog sources,

`sim/rtl_sim/bin` - binaries required for simulation,

`sim/rtl_sim/log` - log files created during simulation,

`sim/rtl_sim/modelsim` - modelsim simulation specific files and makefile,

`sw/example` - bare metal example application for or1k,

`syn/quartus/bin` - synthesis makefile and scripts for quartus example project,

`syn/quartus/run` - synthesis execution directory,

`syn/quartus/src` - example project sources.

2.2 Simulation

To start simulation just enter to `sim/rtl_sim/modelsim` directory and type `make`:

```
#> cd sim/rtl_sim/run
#> make
```

Every testbench is written in SystemVerilog (mostly due to use of `assert` keyword). Every testbench is self checking. Test error are represented by assert failures. Every testbench starts by displaying:

```
# testbench_name start ...
```

and ends by displaying:

```
# testbench_name finish ...
```

If no asserts are displayed between these lines, the test passes. Below is an example of passing test:

```
...
some compilation output
...
# sd_cmd_master_tb start ...
# sd_cmd_master_tb finish ...
# ** Note: $finish    : ../../../bench/verilog/sd_cmd_master_tb.sv(385)
#    Time: 3620 ps  Iteration: 0  Instance: /sd_cmd_master_tb
```

Below is an example of failing test:

```
...
some compilation output
...
# sd_cmd_master_tb start ...
# ** Error: Assertion error.
#    Time: 3280 ps  Scope: sd_cmd_master_tb File: ../../../bench/verilog/
                                                    sd_cmd_master_tb.sv Line: 376
# sd_cmd_master_tb finish ...
# ** Note: $finish    : ../../../bench/verilog/sd_cmd_master_tb.sv(385)
#    Time: 3620 ps  Iteration: 0  Instance: /sd_cmd_master_tb
```

2.2.1 Simulation makefile targets

The default simulation target is to run all testbenches from `bench/verilog` directory that ends with `_.sv`. Other simulation targets are:

`clean` - remove all simulation output files,

`print_testbenches` - lists all available testbenches,

`modelsim` - compiles all sources and launches modelsim (see 2.2.2),

`*_tb` - compiles and executes given testbench. All items listed by the `print_testbenches` target can be executed this way,

`*_tb_gui` - same as `*_tb` target, only instead of executing simulation in command-line, launches modelsim.

2.2.2 Simulation makefile environment variables

Simulation makefile uses couple of environment variables to setup simulation:

`MODELSIM_DIR` - modelsim installation directory (`\$(MODELSIM_DIR)/bin/vsim` should be a valid path),

`VCD` - when set to 1 - all waveforms are dumped to `sim/rtl_sim/out/*.vcd` files; when set to 0 - no waveforms are dumped (0 is default),

`V` - when set to 1 - enables verbose output; when set to 0 - normal simulation output (0 is default).

2.3 Synthesis

For the purpose of synthesis verification there is an example FPGA project made for Altera Quartus. To start synthesis just enter to `syn/quartus/run` directory and type `make`:

```
#> cd syn/quartus/run
#> make
```

Example project consist of all verilog sources from `rtl/verilog` directory and `syn/quartus/src/sdc_controller_top.v` source file. The purpose of the additional verilog file is to instantiate the *Wishbone SD Card Controller IP Core* and register all inputs/outputs to/from the core. This makes timing verification more accurate.

2.3.1 Synthesis makefile targets

The default synthesis target is to synthesize the project and create .sof file in `syn/quartus/run` directory. Other synthesis targets are:

`clean` - remove all synthesis output files,

`print_config` - prints projects configuration of FPGA device,

`project` - creates quartus project files (.qpf and .qsf),

`quartus` - creates quartus project files and launches quartus IDE.

2.3.2 Synthesis makefile environment variables

Synthesis makefile uses couple of environment variables to setup synthesis:

`QUARTUS_DIR` - quartus installation directory (`\$(QUARTUS_DIR)/bin/quartus` should be a valid path),

`FPGA_FAMILY` - name of the FPGA device family,

`FPGA_PART` - name of the FPGA device,

`V` - when set to 1 - enables verbose output; when set to 0 - normal simulation output (0 is default).

3 HDL interface

IP core has very simple interface:

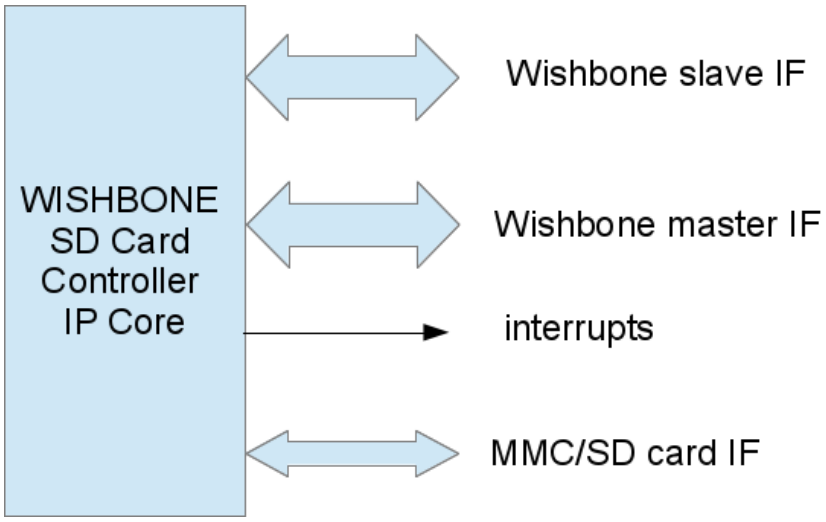


Figure 2: Wishbone SD Card Controller IP Core interface

Wishbone slave interface provides access from CPU to all IP core registers (see 4.1). It must be connected to CPU data master. Wishbone master interface provides access for DMA engine to RAM (see 3.2). It must be connected to RAM memory slave. Interrupts signals provides mechanism to notify the CPU about finished transactions (data and command transfers). They are not necessary for proper operation (if You don't want to use interrupts). MMC/SD card interface provides communication with external MMC/SD cards. It must be connected to external pins of the FPGA which are connected to MMC/SD card connector. Because those external pins are bidirectional, IP core provides inputs, outputs and output enables for these signals. Table 1 presents all IP core signals with descriptions.

3.1 Clock consideration

IP core needs two clock sources. First one is for Wishbone bus operation (`wb_clk_i`). There are no constraints for this clock. Second one is for MMC/SD interface operation (`sd_clk_i_pad`). `sd_clk_i_pad` is used to drive `sd_clk_o_pad` output, which is the external MMC/SD card clock source, through internal clock divider. This clock divider is able to divide `sd_clk_i_pad` clock by 2, 4, 6, 8, ... etc. (2*n where n = [1..256]). `sd_clk_o_pad` clock frequency depends on MMC/SD specification. To fully utilize the transmission bandwidth `sd_clk_o_pad` should be able to perform at 25MHz frequency which imposes constraint of minimum 50MHz on `sd_clk_i_pad` clock. Clock inputs `wb_clk_i` and `sd_clk_i_pad` can be sourced by the same signal.

3.2 DMA engine

DMA engine is used to lower the CPU usage during data transactions[1]. DMA starts its operation immediately after successful end of any read or write command transactions[2] [3]. During write transactions, data is fetched from RAM automatically, starting from known address. This address has to be configured by the CPU before sending any write command. Similarly, during read

[1]Data transaction refers to any traffic on the data lines of MMC/SD card interface.
[2]Command transaction refers to any traffic on the command line.
[3]Read or write command refer to command with data payload such as *block read*(CMD17) or *block write*(CMD24).

Table 1: Signals description

name	direction	width	description
Wishbone common signals			
wb_clk_i	input	1	clock for both master and slave wishbone transactions
wb_rst_i	input	1	reset for whole IP core
Wishbone slave signals			
wb_dat_i	input	32	data input
wb_dat_o	output	32	data output
wb_adr_i	input	32	address
wb_sel_i	input	4	byte select
wb_we_i	input	1	write enable
wb_cyc_i	input	1	cycle flag
wb_stb_i	input	1	strobe
wb_ack_o	output	1	acknowledge flag
Wishbone master signals			
m_wb_dat_o	output	32	data output
m_wb_dat_i	input	32	data input
m_wb_adr_o	output	32	address
m_wb_sel_o	output	4	byte select
m_wb_we_o	output	1	write enable
m_wb_cyc_o	output	1	cycle flag
m_wb_stb_o	output	1	strobe
m_wb_ack_i	input	1	acknowledge flag
m_wb_cti_o	output	3	cycle type identifier (always 000)
m_wb_bte_o	output	2	burst type (always 00)
MMC/SD signals			
sd_cmd_dat_i	input	1	command line input
sd_cmd_out_o	output	1	command line output
sd_cmd_oe_o	output	1	command line output enable
sd_dat_dat_i	input	4	data line inputs
sd_dat_out_o	output	4	data line outputs
sd_dat_oe_o	output	1	data line outputs enable
sd_clk_o_pad	output	1	clock for external MMC/SD card
sd_clk_i_pad	input	1	clock for MMC/SD interface
Interrupts			
int_cmd	output	1	command transaction finished interrupt
int_data	output	1	data transaction finished interrupt

transactions, data is written to RAM automatically, starting from known address. This address also has to be configured by the CPU before sending any read command. Because data transmission is half-duplex, read and write addresses are placed in the same configuration register. Function of this register depends on the command to be sent.

3.3 Interrupt generation

Interrupts are useful when polling technique is not an option. There are two interrupt sources. One to notify the end of the command transaction (`int_cmd` signal) and one to notify the end of the data transaction (`int_data` signal). Both interrupts has active high logic. All events that trigger each interrupts can be masked(see 4.1) and therefore, do not participate in interrupt generation(see 3).

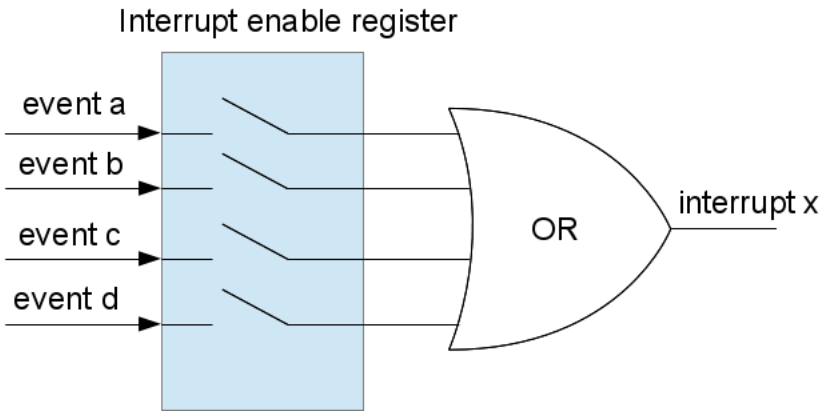


Figure 3: Interrupt generation scheme

3.3.1 Command transaction events

Command transaction end interrupt is driven by the command transaction events. The events are:

completion - transaction completed successfully,

error - transaction completed with error (one or more of the following events occurred),

timeout - timeout error (the card did not respond in a timely fashion),

wrong crc - crc check error (crc calculated from received response data did not match to the crc field of the response),

wrong index - index check error (response consists of wrong index field value).

3.3.2 Data transaction events

Data transaction end interrupt is driven by the data transaction events. The events are:

completion - transaction completed successfully,

wrong crc - crc check error (in case of write transaction, crc received in response to write transaction was different than one calculated by the core; in case of read transaction, crc calculated from received data did not match to the crc field of received data),

fifo error - internal fifo error (in case of write transaction, tx fifo became empty before all data was send; in case of read transaction, rx fifo became full; both cases are caused by to slow wishbone bus or wishbone bus been busy for to long)).

4 Software interface

Access to IP core registers is provided through Wishbone slave interface.

4.1 IP Core registers

Table 2: List of registers

name	address	access	description
argument	0x00	RW	command argument
command	0x04	RW	command transaction configuration
response0	0x08	R	bits 31-0 of the response
response1	0x0C	R	bits 63-32 of the response
response2	0x10	R	bits 95-64 of the response
response3	0x14	R	bits 119-96 of the response
data_timeout	0x18	RW	data xfer timeout configuration
control	0x1C	RW	IP core control settings
cmd_timeout	0x20	RW	command xfer timeout configuration
clock_divider	0x24	RW	MMC/SD interface clock divider
reset	0x28	RW	software reset
voltage	0x2C	R	power control information
capabilities	0x30	R	capabilities information
cmd_event_status	0x34	RW	command transaction events status / clear
cmd_event_enable	0x38	RW	command transaction events enable
data_event_status	0x3C	RW	data transaction events status / clear
data_event_enable	0x38	RW	data transaction events enable
blkock_size	0x44	RW	read / write block transfer size
blkock_count	0x48	RW	read / write block count
dst_src_address	0x60	RW	DMA destination / source address

4.1.1 Argument register

Write operation to this register triggers command transaction (command register has to be configured before writing to this register).

Table 3: Argument register

bit #	reset value	access	description
[31:0]	0x00000000	RW	command argument value.

4.1.2 Command register

This register configures all aspects of command to be sent.

Table 4: Command register

bit #	reset value	access	description
[31:14]			reserved
[13:8]	0x00	RW	command index
[7]			reserved
[6:5]	0x0	RW	data transfer specification. 0x0 - no data transfer; 0x1 - triggers read data transaction after command transaction; 0x2 - triggers write data transaction after command transaction
[4]	0x0	RW	check response for correct command index
[3]	0x0	RW	check response crc
[2]	0x0	RW	check for busy signal after command transaction (if busy signal will be asserted after command transaction, core will wait for as long as busy signal remains)
[1:0]	0x0	RW	response check configuration. 0x0 - don't wait for response; 0x1 - wait for short response (48-bits); 0x2 - wait for long response (136-bits)

4.1.3 Response register 0-3

Response registers 0-3 contains response data bits after end of successful command transaction (if bits 1-0 of command register were configured to wait for response).

Table 5: Response register 0-3

bit #	reset value	access	description
[31:0]	0x00000000	R	response data bits

4.1.4 Data xfer timeout register

Data timeout register configures data transaction watchdog counter. If any data transaction will last longer than configured timeout, interrupt will be generated. Value in timeout register represents the number of `sd_clk_o_pad` clock cycles. Register value is calculated by following formula:

$$REG = \frac{timeout[s] * frequency_{\texttt{sd\_clk\_i\_pad}}[Hz]}{(2 * (\texttt{clock\_devider} + 1))} \quad (1)$$

Table 6: Data xfer timeout register

bit #	reset value	access	description
[31:24]			reserved
[23:0]	0x0	RW	timeout value (when 0 - timeout is disabled)

4.1.5 Control register

Table 7: Control register

bit #	reset value	access	description
[31:1]			reserved
[0]	0x0	RW	MMC/SD bus width; 0x0 - 1-bit operation; 0x1 - 4-bit operation

4.1.6 Command xfer timeout register

Command timeout register configures command transaction watchdog counter. If any command transaction will last longer than configured timeout, interrupt will be generated. Value in timeout register represents the number of `sd_clk_o_pad` clock cycles. Register value is calculated by following formula:

$$REG = \frac{timeout[s] * frequency_{\texttt{sd\_clk\_i\_pad}}[Hz]}{(2 * (\texttt{clock\_devider} + 1))} \tag{2}$$

Table 8: Command xfer timeout register

bit #	reset value	access	description
[31:16]			reserved
[15:0]	0x0	RW	timeout value (when 0 - timeout is disabled)

4.1.7 Clock divider register

Clock divider register control division of `sd_clk_i_pad` signal frequency. Output of this divider is routed to MMC/SD interface clock domain. Register value is calculated by following formula:

$$REG = \frac{frequency_{\texttt{sd\_clk\_i\_pad}}[Hz]}{2 * frequency_{\texttt{sd\_clk\_i\_pad}}[Hz]} - 1 \tag{3}$$

Table 9: Clock divider register

bit #	reset value	access	description
[31:8]			reserved
[7:0]	0x0	RW	divider ratio

4.1.8 Software reset register

Table 10: Software reset register

bit #	reset value	access	description
[31:1]			reserved
[0]	0x0	RW	reset; 0x0 - no reset; 0x1 - reset applied

4.1.9 Voltage information register

This register contains the value of power supply voltage expressed in mV. It is read-only register and its value is configured in HDL.

Table 11: Software reset register

bit #	reset value	access	description
[31:0]		R	power supply voltage [mV]

4.1.10 Capabilities information register

Table 12: Capabilities information register

bit #	reset value	access	description
[31:0]			reserved

4.1.11 Command events status register

This register holds all pending event flags related to command transactions. Write operation to this register clears all flags.

Table 13: Command events status register

bit #	reset value	access	description
[31:5]			reserved
[4]	0x0	RW	index error event
[3]	0x0	RW	crc error event
[2]	0x0	RW	timeout error event
[1]	0x0	RW	error event (logic sum of all error events)
[0]	0x0	RW	command transaction successful completion event

4.1.12 Command transaction events enable register

This register acts as event *and* mask. To enable given event, corresponding bit must be set to 1.

Table 14: Command transaction events enable register

bit #	reset value	access	description
[31:5]			reserved
[4]	0x0	RW	enable index error event
[3]	0x0	RW	enable crc error event
[2]	0x0	RW	enable timeout error event
[1]	0x0	RW	enable error event
[0]	0x0	RW	enable command transaction successful completion event

4.1.13 Data transaction events status register

This register holds all pending event flags related to data transactions. Write operation to this register clears all flags.

Table 15: Data transaction events status register

bit #	reset value	access	description
[31:4]			reserved
[3]	0x0	RW	fifo error event
[2]	0x0	RW	crc error event
[1]	0x0	RW	error event (logic sum of all error events)
[0]	0x0	RW	data transaction successful completion event

4.1.14 Data transaction events enable register

This register acts as event *and* mask. To enable given event, corresponding bit must be set to 1.

Table 16: Data transaction events enable register

bit #	reset value	access	description
[31:4]			reserved
[3]	0x0	RW	enable fifo error event
[2]	0x0	RW	enable crc error event
[1]	0x0	RW	enable error event
[0]	0x0	RW	enable data transaction successful completion event

4.1.15 Block size register

This register controls the number of bytes to write/read in a single block. Data transaction will transmit number of bytes equal to size of block times blocks count. Register value is calculated by following formula:

$$REG = size\_of\_block - 1 \tag{4}$$

Table 17: Block size register

bit #	reset value	access	description
[31:12]			reserved
[11:0]	0x200	RW	number of byes in a single block minus 1

4.1.16 Block count register

This register controls the number of blocks to write/read in data transaction. Data transaction will transmit number of bytes equal to value blocks count times block size. Register value is calculated by following formula:

$$REG = number\_of\_blocks - 1 \tag{5}$$

Table 18: Block count register

bit #	reset value	access	description
[31:16]			reserved
[15:0]	0x0	RW	number of blocks in data transaction minus 1

4.1.17 DMA destination / source register

This registers configures the DMA source / destination address. For write transactions, this address points to the beginning of data block to be sent. For read transactions, this address points to the beginning of data block to be written.

Table 19: DMA destination / source register

bit #	reset value	access	description
[31:0]	0x00000000	RW	address