SD Card Controller IP Specification

Marek Czerski

Tuesday 10th September, 2013

List of Figures

List of Tables

1 Introduction

This document descripes 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 an MMC/SD communication controller designed to be used in a System-on-Chip (Fig. 1). The IP core provides a simple interface for any MCU which utilizes the Wishbone bus. Communications between the MMC/SD card controller and MMC/SD card are performed according to the MMC/SD protocol.

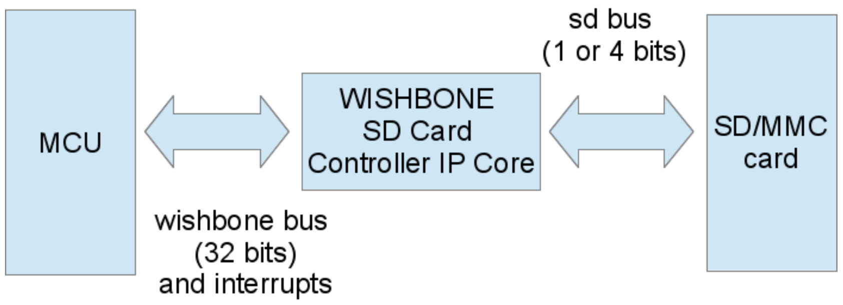
MCU
wishbone bus (32 bits) and interrupts
WISHBONE SD Card Controller IP Core
sd bus (1 or 4 bits)
SD/MMC card

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 tranfser),
- 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
├── doc
│   ├── references
│   └── src
├── rtl
│   └── verilog
├── sim
│   └── rtl_sim
│       ├── bin
│       ├── log
│       └── run
├── sw
│   └── example
└── syn
    └── quartus
        ├── bin
        ├── run
        └── src
```

`bench/verilog` - Verilog testbench sources,

`doc` - documentation files,

`doc/src` - documentation LaTeXsources,

`rtl/verilog` - IP core Verilog sources,

`sim/rtl_sim/bin` - simulation makefile and modelsim scripts,

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

`sim/rtl_sim/run` - simulation execution directory,

`sw/example` - baremetal 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 the simulation just change to the `sim/rtl_sim/run` 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 errors 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 passed. 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 end 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 the 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 the simulation via command-line, launches modelsim.

2.2.2 Simulation makefile environment variables

The simulation makefile uses a couple of environment variables to setup the 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
```

The example project consists of all Verilog sources from the `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 a .sof file in the `syn/quartus/run` directory. Other synthesis targets are:

`clean` - remove all synthesis output files,

`print_config` - prints project configuration of FPGA device,

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

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

2.3.2 Synthesis makefile environment variables

The synthesis makefile uses a 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

This IP core has a very simple interface:

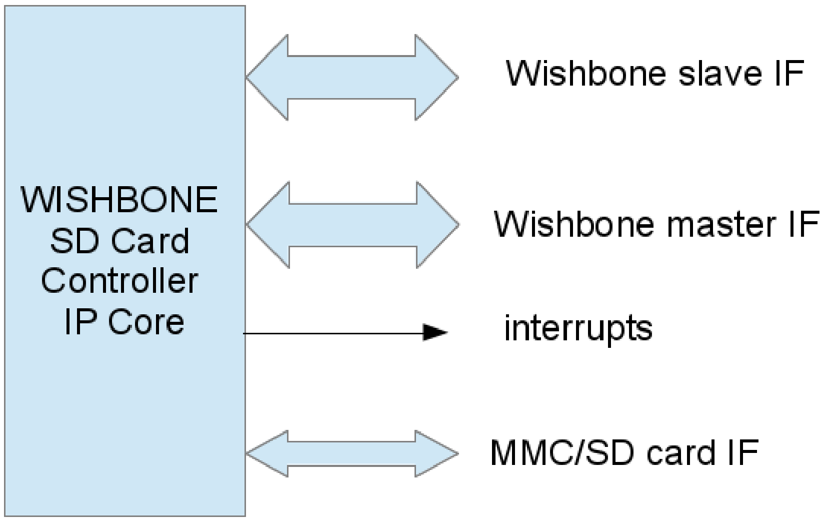

Figure 2: Wishbone SD Card Controller IP Core interface

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

3.1 Clock consideration

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

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

[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" commands refer to commands with data payload such as *block read*(CMD17) or *block write*(CMD24).

Table 1: Description of signals

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

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

3.3 Interrupt generation

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

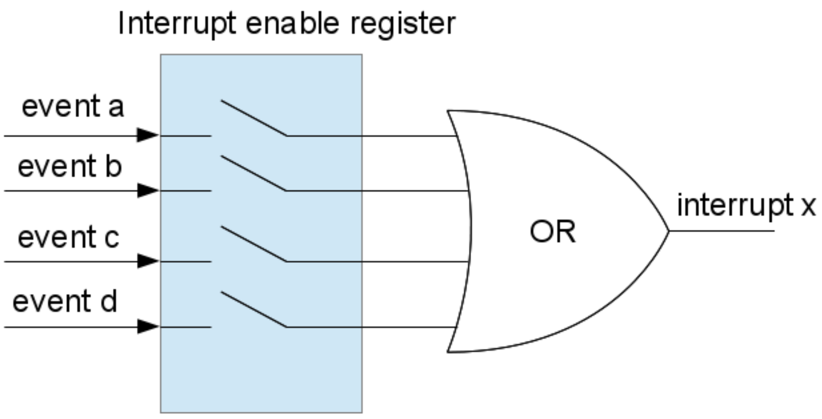

Figure 3: Interrupt generation scheme

3.3.1 Command transaction events

The command transaction end interrupt is driven in response to 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

The data transaction end interrupt is driven in response to 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 the one calculated by the core; in case of read transaction, the CRC calculated from received data did not match to the CRC field of the received data),

FIFO error - internal FIFO error (in case of write transaction, tx FIFO became empty before all data was sent; in case of read transaction, rx FIFO became full; both cases are caused by too slow of a wishbone bus or the wishbone bus being busy for too long)).

4 Software interface

Access to IP core registers is provided through a 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
`control`	0x1C	RW	IP core control settings
`timeout`	0x20	RW	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

A write operation to this register triggers a command transaction (The 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 the 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, the core will wait for as long as the 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 contain response data bits after a 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 Control register

Table 6: 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

4.1.5 Timeout register

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

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

Table 7: Timeout register

bit #	reset value	access	description
`[31:16]`			reserved
`[15:0]`	`0x0`	RW	timeout value

4.1.6 Clock divider register

The clock divider register controls division of the `sd_clk_i_pad` signal frequency. The output of this divider is routed to the MMC/SD interface clock domain. The register value is calculated by following formula:

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

Table 8: Clock divider register

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

4.1.7 Software reset register

Table 9: Software reset register

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

4.1.8 Voltage information register

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

Table 10: Software reset register

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

4.1.9 Capabilities information register

Table 11: Capabilities information register

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

4.1.10 Command events status register

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

Table 12: 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 succesful completion event

4.1.11 Command transaction events enable register

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

Table 13: 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 (logic sum of all error events)
[0]	0x0	RW	enable command transaction successful completion event

4.1.12 Data transaction events status register

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

Table 14: Data transaction events status register

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

4.1.13 Data transaction events enable register

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

Table 15: Data transaction events enable register

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

4.1.14 Block size register

This register controls the number of bytes to write/read in a single block. A data transaction will transmit a number of bytes equal to the block size times the block count.

Table 16: Block size register

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

4.1.15 Block count register

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

$$REG = number\_of\_blocks - 1 \tag{3}$$

Table 17: Block count register

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

4.1.16 DMA destination / source register

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

Table 18: DMA destination / source register

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