sdio_controller_host

20/05/2019

Marco Paci

1 Introduction

This document provides a short description of the signals, the main internal component and the way to use the sdio_controller_host. The document alone may be used for a basic understanding of how i made up the cores developed for this project. So this doc will not be a specific documentation about how i developed it and the chose i made in the development, but simply a clarification to fasten up the job for anyone wants to go on with the project.

The sdio_controller_host was developed based on the STM32 sdio_host_controller described in the reference manual. The job i did is not 100% done, but it does implement all the main functions of an sdio controller.

For a complete reference of the work,the reader should rely on the STM32 reference manual “RM0090” section about SDIO, and on the Wishbone specification document.

2 Description of cores

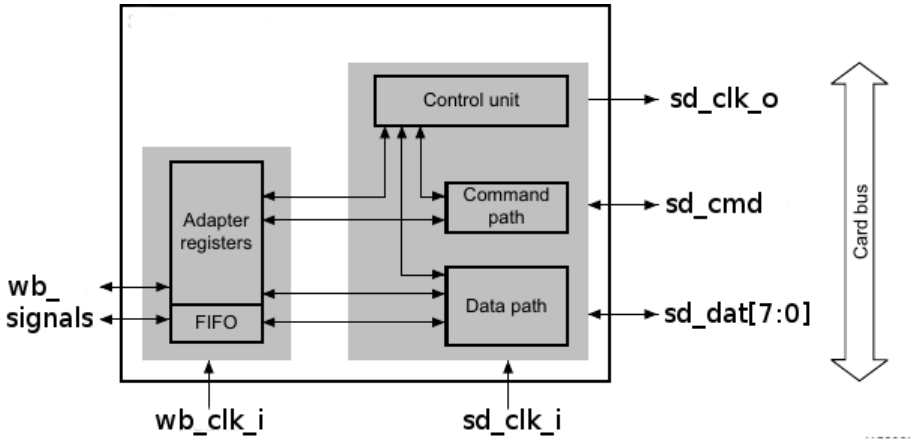
Sdio core consists of two parts:

1. The wishbone I/F block which act as a slave unit and is connected to the host with wb standard slave signal. This unit does work under the *wb_clk*;

2. The SDIO controller block does work under *sd_clk_i* and handles all the main function of the controller, such as sending and receiving data and commands to/from the card and generating the clock used for the communication of the card.

sd_dat_0 is the wire used by default for communicating with the card. Setting the configuration registers of the core, user can change and make choose to use 4 or all 8 wires for exchange data with the external card.

sd_cmd wire is used for sending and receiving cmd signals from the external card. every data exchange between the host and the card is synchronous to the *sd_clk_o*. This clock derives from the *sd_clk_i*, and depends on the way user sets the configuration register of this clock. This clock is also an output to the card, so the host and the card can work synchronously with the same clock.



2.1 sd_top.sv

sd_top module is the container of all the other submodules. The naming of the signals is kinda clear, all *wb_** signals are the ones interfacing with the wishbone host, the *sd_** signals are the one going to the card, the only signal non following this nomenclature is *sd_clk_i*, that is an input in the controller. *sd_cmd* and *sd_dat* signals are bidirectional wires With the card, so they can works as receivers and as sender to the card, sd_clk_o is an output.

Specifically, in this module there are the **wb_interface_slave**, **sd_fifo_filler**, **sd_cmd_adapter**, **sd_dpsm** and **sd_clk_divider**. Other then this there are multiple "bistable_domain_cross" modules for synchronization manners.

2.2 wb_interface_slave.sv

All of the register of the controller are accessible through this module, except for the fifo registers. During commands or data transfers configuration registers (all of them) are not writable, but they are readable. This is due tot the fact that changing configuration registers during a transfer would change the way the external card and the controller communicates, generating problems.

The complete description of the register can be found in the STM32 documentation. Not all the options described in the registers are supported by this project, but the most valuable are. Important register among all are the **sd_cmd** and **sd_dtctrl** registers, that start a command or a data transfer/receive when a specific bit is set to 1 in them. Another important register is the **sd_status** register, since the user, reading it, can check the way the transfer is going, and pooling it's the only way for now to see if the tx/rx is over (interrupt are not implemented).

The offset of the registers goes from *0x00* to *0x7F*. *0x80* is the address to access the fifo. When in wb is set an adress from *0x00* to *0x7F*, the slave wb signals are driven by this interface, if the address is *0x80* the wb slave signals are driven by the fifo interface.

2.3 sd_fifo_filler.sv

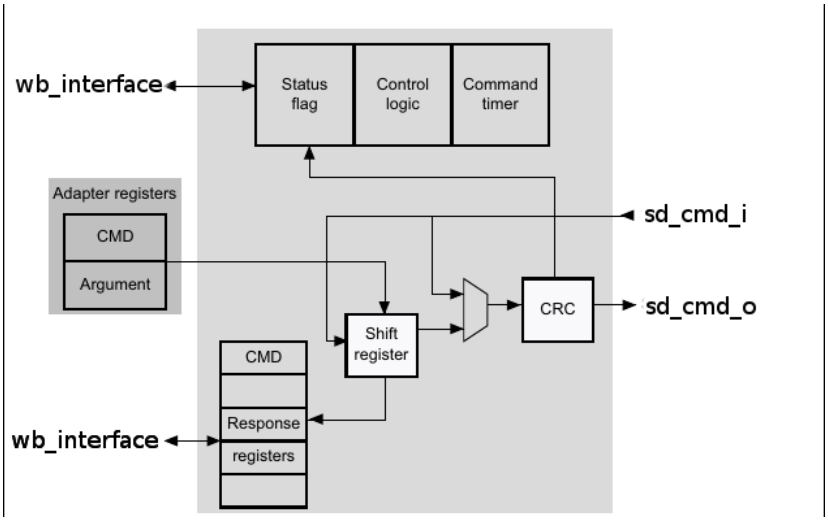
This module is necessary to establish a channel of communication between the data to/from the memory and those from/to the card. Is composed of 2 fifos of 32 words, one for receive and one for transfer. They work with different clock, the receive fifo writes data on the fifo in *sd_clk_o* and data are read from the wb interface (with a DMA) in *wb_clk*. The other fifo is written in *wb_clk* and is read in *sd_clk_o*. Neither one of the fifo is accessible if there is no data tx or rx in act. Every time a data tx/rx start they are reset and becomes accessible from wb. When the receive fifo is empty or the tx fifo is full they can't be red or written. In this project every time i talk about impossibility to access the registers, this mean on the project as not setting the *wb_ack_slave* signal to 1. This situation may be probably handled in better way using *wb_rty* or *wb_err* signals, to not stall wishbone and steal bus cycles.

The fifo not only communicates with wishbone, but it does communicates to with the **sd_dpsm**. All the signal in the communication are synchronized with the right clocks. The **sd_dpsm**, depending if we are transferring or receiving data, will write or read data to/from the fifo.

In a later section there will be an example of data transfer and how to set up all the register and the DMA to do a transfer, so everything may be more clear.

2.4 sd_clock_divider.sv

This unit simply receives as input the *sd_clk_i* and, based on the **sd_clkcr** and **sd_power** register, generates and modules the clock and gives in output the *sd_clk_o*, which is the clock of the controller and of the external card.



2.5 sd_cmd_adapter.sv

The description of what this unit does is well described on the STM32 document(pg 1025). The commands are sent from the host to the card to inform the card of what we are going to do or how we want to do something. For example if i want to tell the card to work on all 8 data wires, a command with a specific argument must be sent to the card. Same stuff if it's necessary to start sending data or to receive data from the card etc.. All that happen between the host and the card is controlled by the commands.

This modules access to 2 register that are in the wb_interface: **sd_cmd** and **sd_arg**. **sd_arg** register handles the 32 bit of argument to send to the card, in **sd_cmd** register there are configuration bit that changes the behavior of this controller, and the start bit (bit 10), that when written to 1 makes this module start.

So to send a command first of all the **sd_arg** register must be set, then the **sd_cmd** register is set with the right index bits and the start bit set to 1. This way the module will start sending the command. To see if the command has been sent successfully, the driver must pool the *sd_status* register and check if it's values are right. If there was a fail in the CRC checksum the status will present it and the send must be retaken. If all went good, the cmd has been sent successfully. Also if there was a response or a long response, it is saved in the **sd_resp1/2/3/4** register and they can be red setting their address in a wishbone read.

All the information about how a command is formed and the way the communication happen can be found in the STM32 doc or in the SD specification Physical Layer.

2.6 sd_dpsm.sv

Just like the **sd_cmd_adapter** any particular information on how this work can be found in the STM32 documentation or on the SD physical layer specification.

The register accessed by this module are the **sd_clkcr**, **sd_dlen**, **sd_dctrl**, **sd_dtimer**. Tuning this register, user can set up if to use 1, 4 or 8 wires in parallel to send data (obviously before doing that one must send the right command to the sd_card and inform the card of this information). Also depending on those configuration bits, the dpsm may go in transfer or receive mode. Timer register set the time to wait before a timeout error, dlen the size of the data to send.

After doing all the setup and sending the command to the card telling that we are going to do tx or rx, when a bit (bit 0) of the **sd_dctrl** register is set to 1, the module start to work.

Data are sent or received in blocks, at the end of each block is attached a CRC checksum. If any block present a fail in the CRC the transmission or reception is aborted. Another way to make it abort is due to fifo under run or overrun error. Those error may happen if the DMA is not fast enough to read or write the data in the FIFO. Anyway putting some constraint on the *sd_clk_o* speed with respect to the *wb_clk*, this situation should not happen.

The data from the fifo are red in words, those word are red from the least significant byte to the most one, but every byte is sent from the most significant bit to the least (if we use 8 data line that's not a problem, but care must be taken when using 1 or 4 data wires).

As for the **cmd_adapter**, to check if the transmission is over and went right, the **sd_status** register must be red. During the transmission is possible to read the **sd_dcount** and the **sd_fifocount** register, to check how many data and how many words in the fifo are remained to be sent.

3 Procedure to transfer data

In this example we are gonna transfer 512 bytes of data on a single wire to the card. The following are the step to do:

- Do the card identification process (as described in the STM32 reference document pg 1035)
- Set the *sd_clk_o* frequency writing in the register **sd_clkcr**
- Send the CMD7 to the card(selection command)
- Set the sd data lenght register, data timer register and all the other configuration register with the value wanted
- Set the DMA controller to write 512 bytes of data from the desired memory address to the FIFO memory address(it won't write anything till data transfer start)
- write in the **sd_adr** register the address of the card where you want to write the data
- write the **sd_cmd** register with CMD index 24 (write_block), waitResp '1' (response from card expecter) and enable bit '1'.
- Pool the status register till the transfer signals it is over, then program the **sd_dtctrl** with data enable '1' dtdir '0'(write) dtmode'0'(block trasfer (the stream trasfer is not implemented)), dt block size with 0x09(512bytes). After this register transfer will start, and the DMA will be able to write in the fifo.
- Pool the status register and wait for completion bit.