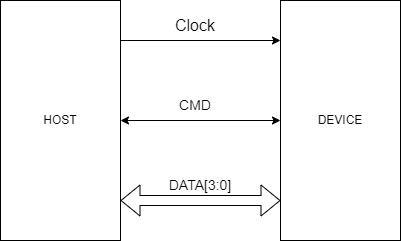
UDMA SD CARD INTERFACE

The SDIO (Secure digital I/O) card provides high speed data I/O with low power consumption for mobile electronic devices. Host devices supporting SDIO can connect the SD slot with I/O devices like Bluetooth, wireless, LAN, GPS Receiver, Digit Camera etc.

SDIO INTERFACE BUS:



FEATURES:

* It has a clock, command and 4-bit data bus.
* Supports quad mode.
* Five types of response supported:
  + No response
  + 48 bits with CRC
  + 48 bits with NO CRC.
  + 136 bits with BUSY check.
* Error output pin.
* End of transfer output pin.
* Four error status for command transfer supported:
  + No error
  + Response timeout
  + Response wrong direction
  + Response busy timeout

THEORY OF OPERATION:

Communication over the SD bus is based on command and data bit streams that are initiated by a start and terminated by a stop bit.

* Command: Operation is started by generating a command. A host can send a command to either a single card or to all the connected cards. A command is transferred serially on the CMD line. MSB is transmitted first and LSB is transmitted last. Transmission bit is 1 as command is transmitted from host to device.
  + Command token formatCommand token
    

| Bit position | 47 | 46 | [45:40] | [39:8] | [7:1] | 0 |
| --- | --- | --- | --- | --- | --- | --- |
| Width | 1 | 1 | 6 | 32 | 7 | 1 |
| Value | 0 | 1 | x | x | x | 1 |
| Description | Start bit | Transmission bit | cmd opcode | Cmd arg | crc | End bit |

* Response: A response is sent from an addressed card to the host as an answer to a previously received command. A response is transferred serially on the CMD line. Transmission bit is 0 as response is transmitted from device to host.
  + Response token format:



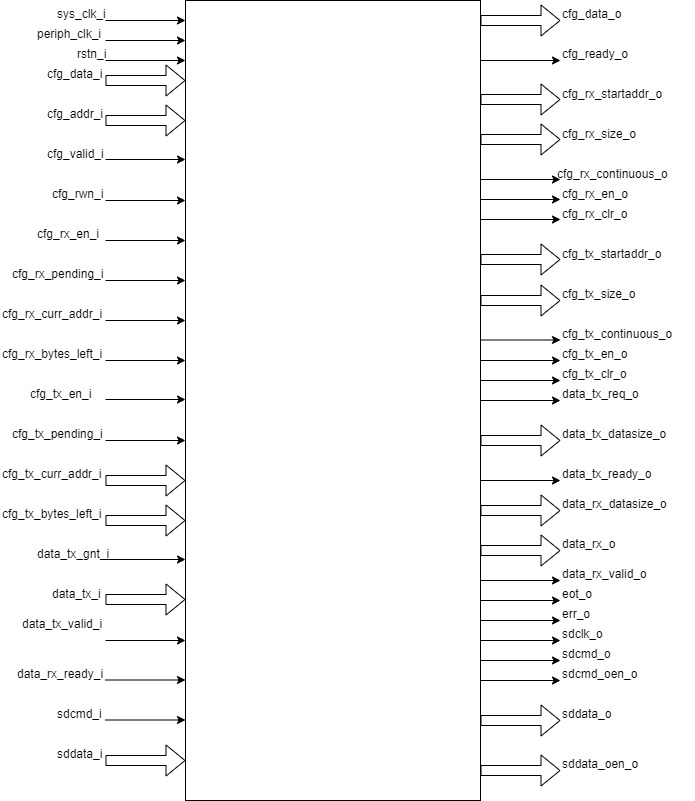
| Bit position | 47 | 46 | [45:40] | [39:8] | [7:1] | 0 |
| --- | --- | --- | --- | --- | --- | --- |
| Width | 1 | 1 | 6 | 32 | 7 | 1 |
| Value | 0 | 0 | x | x | x | 1 |
| Description | Start bit | Transmission bit | Command index | Card status | crc | End bit |

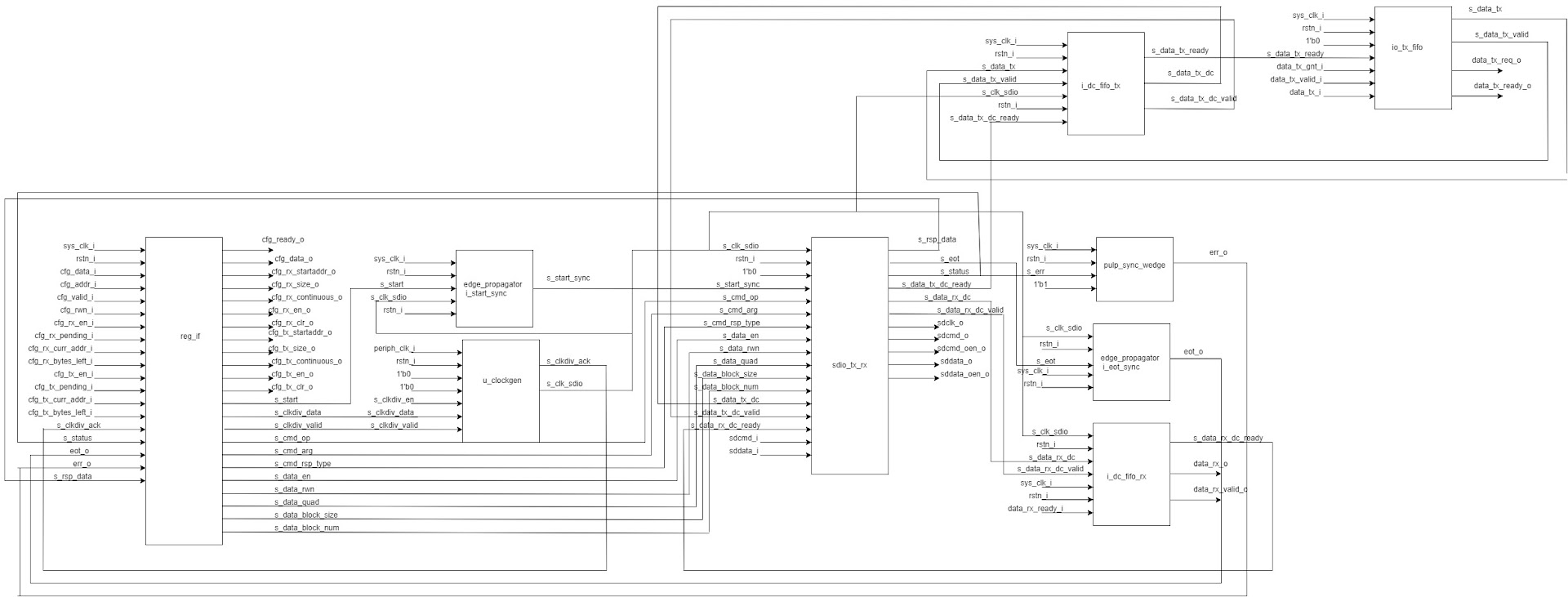
| Bit position | 135 | 134 | [133:128] | [127:1] | 0 |
| --- | --- | --- | --- | --- | --- |
| Width | 1 | 1 | 6 | 127 | 1 |
| Value | 0 | 0 | 111111 | x | 1 |
| Description | Start bit | Transmission bit | Reserved | Response content including CRC | End bit |

* Data: Data can be transferred from the card to the host or vice versa. It is transferred via the data lines.

Data transfer to/from the SD memory card is done in blocks. Data blocks are succeeded by CRC bits.

BLOCK DIAGRAM:

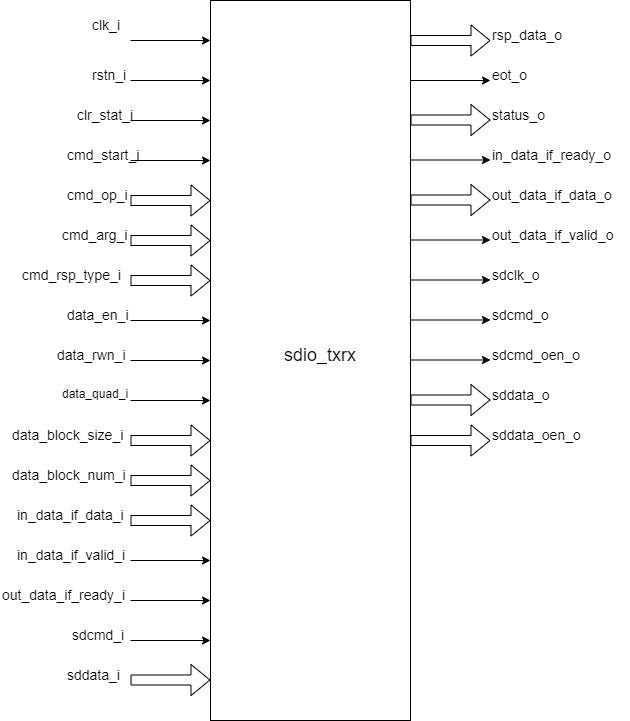




It contains a reg interface for writing and reading from the register and a SDIO TX/RX module which instantiates command and data modules. Command module handles command interface and data module handles data transfer.

SDIO TX/RX:

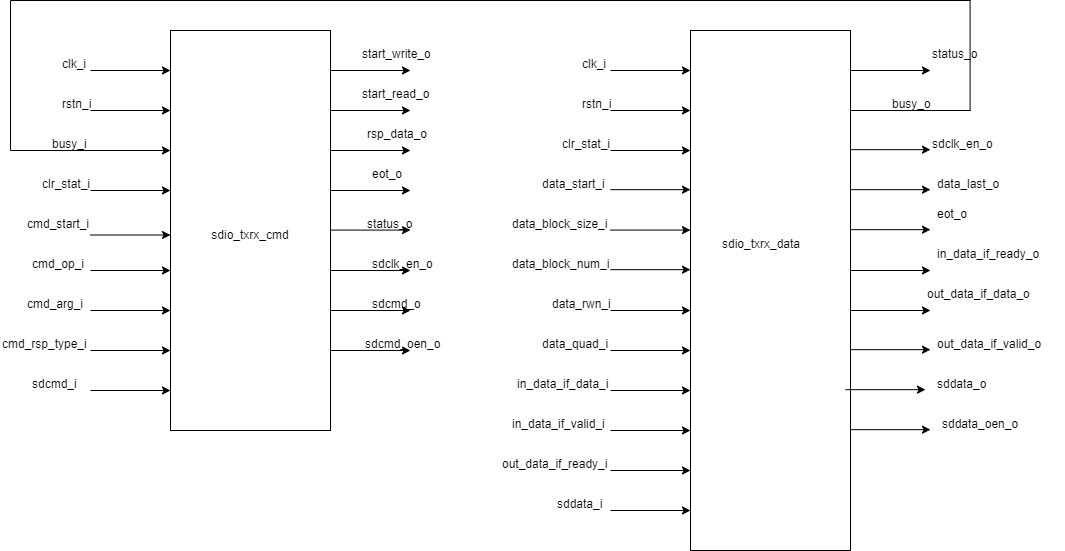
This module is responsible for sending and receiving command and data between host and device. It instantiates command and data modules.



* It uses the clock generated by udma\_clockgen as input clock.
* Synchronous start generated by the edge propagator is used to get the command start bit. This command start bit is sent to the command module which marks the start of the command.
* This module works in three states:
  + CMD\_ONLY: This is the default state.
    - State is set to WAIT\_EOT if there is no block to be transmitted else state is set to WAIT\_LAST.
  + WAIT\_LAST: Wait for the last piece of data to be transferred.
    - After transferring the last piece of data we go to state WAIT\_EOT.
    - Data module sends a high signal which indicates transfer of last data.
  + WAIT\_EOT: Wait for the end of transaction.
    - If command eot and data eot sent by command and data module respectively are high then go to state CMD\_ONLY and reset command eot and data eot.
* Status : 16 bit status output is transmitted through this block. This status can be read through SDIO\_REG\_STATUS. Non-negative status would generate and error.

| Bit position | [15:14] | [13:8] | [7:6] | [5:0] |
| --- | --- | --- | --- | --- |
| value | 00 | x | 00 | x |
| Description | reserved | Data status | reserved | Command status |

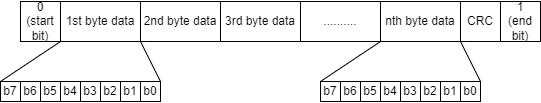
It instantiates two sub blocks: command and data.



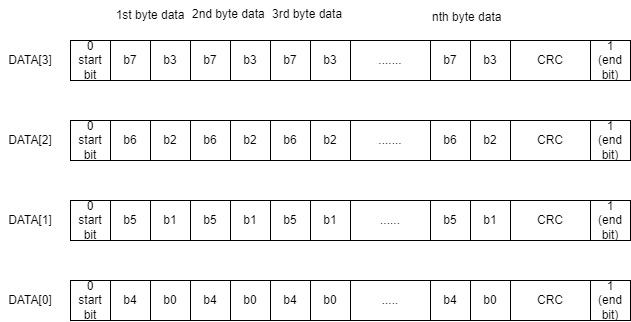
* Command block: This module handles the command interface.
  + It supports three types of response status:
    - Response timeout
    - Response wrong direction
    - Response busy timeout
  + It supports five types of responses: Response type is written to register

SDIO\_REG\_CMD\_OP.

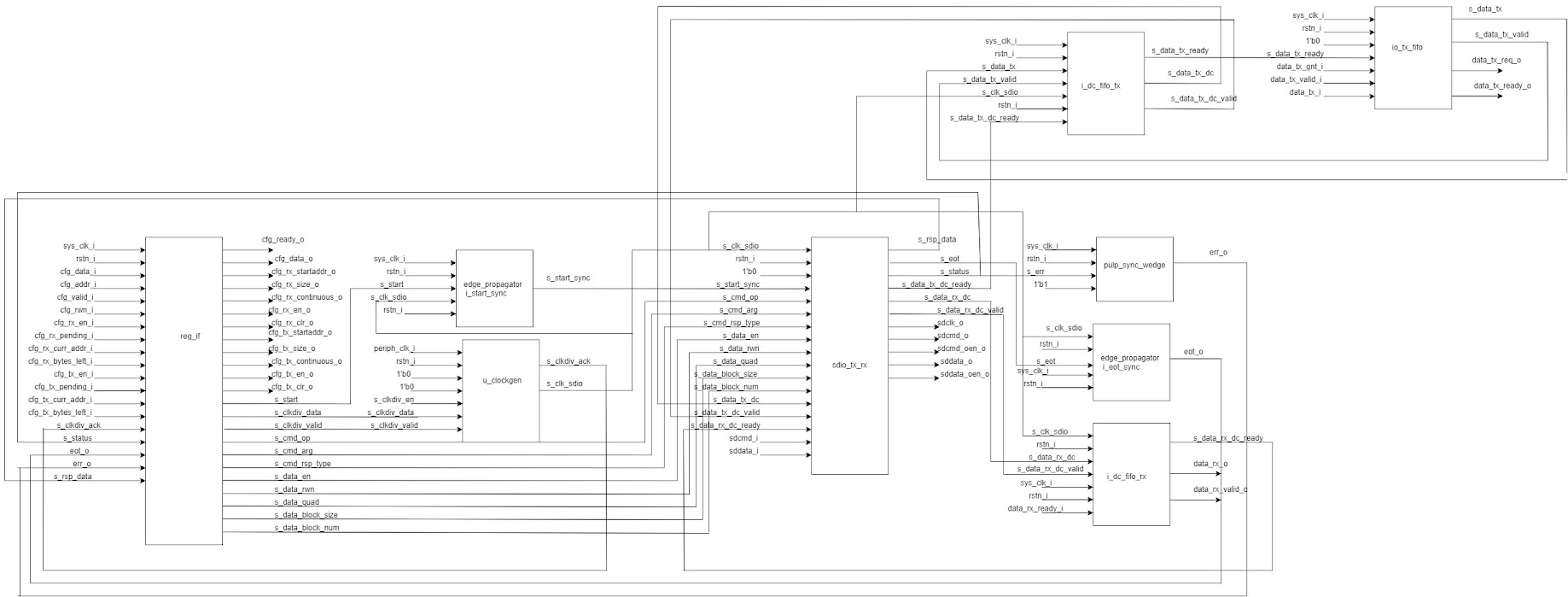
* + - Null response: No response.
    - 48 bits with CRC.
      * Supports CRC.
      * Response length is 38 bits.
    - 48 bits with NO CRC
      * Supports CRC.
      * Response length is 38 bits.
    - 136 bits
      * Supports CRC.
      * Response length is 133 bits.
    - 48 bits with a busy check.
      * Supports CRC.
      * Response length is 38 bits.
      * Supports busy signal.
  + Command output enable signal: sdcmd\_oen\_o is an active low signal. It is enabled during transfer of command and is disabled during reception of response.
  + It goes through twelve states:
    - ST\_IDLE: Default state when the system is IDLE.
      * Clock is disabled initially.
      * If the command start bit is high then state is set to ST\_TX\_START and clock is enabled.
    - ST\_TX\_START: Send the start bit to start the transaction.
      * Start bit is sent through sdcmd\_o.
      * State is set to ST\_TX\_DIR.
    - ST\_TX\_DIR: Set the transmission bit of the command.
      * Transmission bit is sent through sdcmd\_o.
      * CRC is enabled.
      * State is set to ST\_TX\_SHIFT and 38 bit command data is transmitted.
    - ST\_TX\_SHIFT:
      * MSB of the command is sent as output through sdcmd\_o.
      * CRC is enabled.
      * Command data is shifted to the left by 1 bit.
      * If the 38 bit command data is transmitted go to state ST\_TX\_CRC and start counting the CRC bits. There are 7 CRC bits.
    - ST\_TX\_CRC: Send CRC output and shifts CRC.
      * CRC output is sent as output through sdcmd\_o.
      * CRC is enabled.
      * State is set to ST\_TX\_STOP after successfully transmitting the CRC bits.
    - ST\_TX\_STOP: Transmit the stop bit of the command.
      * Stop bit is sent through sdcmd\_o.
      * Start read is enabled which indicate we can read from data block.
      * If the response is enabled, set the state to ST\_RX\_START.
      * If the response is disabled, go to state ST\_WAIT.
    - ST\_RX\_START: Initiates the reception of response.
      * Response is received via sdcmd\_i
      * State is set to ST\_RX\_DIR if the start bit is received.
      * If the start bit is not received till 38 clock cycle then response status is set to response timeout.
    - ST\_RX\_DIR: Check if the received command indicates the correct direction.
      * Direction bit is received and state is set ST\_RX\_SHIFT.
      * Response data is received.
      * Receiving an incorrect bit would set the response status to response wrong direction and the state is set to ST\_IDLE.
    - ST\_RX\_SHIFT: Shift in response data.
      * CRC is calculated.
      * If the response data is received and response crc is enabled then state is set to ST\_TX\_CRC.
      * If response data is received and response crc is disabled and response busy is enabled then go to state ST\_WAIT\_BUSY.
      * If response count is completed but response crc and response busy are not enabled then go to ST\_WAIT.
    - ST\_RX\_CRC:
      * If CRC is received then go to ST\_WAIT..
    - ST\_WAIT\_BUSY:
      * If a low busy signal is received from the data block then we go to ST\_WAIT.
      * If the busy signal is high till 256 clock cycle status is set to response busy timeout.
    - ST\_WAIT:
      * After waiting for 8 clock cycles, high output is asserted through command eot output and high start write output is sent which indicates successful write command.
      * Set the state to ST\_IDLE.
  + Single bit is transferred at every posedge of the clock. Transmitting a 38 bit command data would take 38 clock cycles and a 7 bit crc would take 7 clock cycles. Similarly receiving a response would take response length clock cycle.
  + Response content is received through sdcmd\_i and is sent as output through rsp\_data\_o.
* Data block: This module is responsible for handling data transfer.
  + Support response status timeout.
  + Supports five types of responses:
    - Null response: No response.
    - 48 bits with CRC.
      * Supports CRC.
      * Response length is 38 bits.
    - 48 bits with NO CRC
      * Supports CRC.
      * Response length is 38 bits.
    - 136 bits
      * Supports CRC.
      * Response length is 133 bits.
    - 48 bits with a busy check.
      * Supports CRC.
      * Response length is 38 bits.
      * Supports busy signal.
  + Supports 16 bit CRC.
  + Data output enable signal: sddata\_oen\_o is an active low signal. It is enabled during transfer of data and is disabled during reception of data.
  + Data can be transmitted in 2 modes:
    - Single count mode: Data is transferred only on DATA[0] pin. LSB is transmitted first and MSB is transmitted last.



* + - Quad mode: Data is transferred on all the four data pins.



* + States:
    - ST\_IDLE:
      * For read operation go to state ST\_RX\_START.
      * For write operation go to ST\_TX\_START.
      * Read and write operation is decided by data\_rwn\_i.
    - ST\_TX\_START:
      * Send the start bit through sddata\_o to start the transaction.
      * Go to state ST\_TX\_SHIFT.
      * one block is transmitted.
    - ST\_TX\_SHIFT:
      * Data output is enabled.
      * CRC is calculated.
      * Direction bit is sent.
      * If the whole block is transmitted go to state ST\_TX\_CRC.
    - ST\_TX\_CRC:
      * Output crc through sddata\_o.
      * state is set to ST\_TX\_END.
    - ST\_TX\_END
      * Send ‘F’ through sddata\_o.
      * Go to state ST\_TX\_CRCSTAT.
    - ST\_TX\_CRCSTAT
      * Wait for 8 clock cycles and go to state ST\_TX\_BUSY
    - ST\_TX\_BUSY
      * After waiting for 512 cycles go to timeout phase:
        + Sdio timeout counter is increased till it reaches 1023.
        + State is set to ST\_IDLE and high output is driven through eot output.
      * If 512 cycles is not reached and high bit is received in LSB of incoming data:
        + If all the blocks are transmitted then the state is set to ST\_IDLE and eot is asserted.
        + If the whole block is not transmitted then go to ST\_TX\_START and transmit the next block.
    - ST\_RX\_START:
      * Data is received through sddata\_i.
      * Go to state ST\_RX\_SHIFT if the start bit is received.
      * If the start bit is not received, set the status to response timeout and go to ST\_IDLE.
    - ST\_RX\_SHIFT:
      * CRC is calculated.
      * After receiving the block state is set to ST\_RX\_CRC.
    - ST\_RX\_CRC:
      * CRC is calculated.
      * After receiving the 16 bit CRC, if the all the blocks are received assert eot else go to ST\_RX\_START and receive another block.
    - ST\_WAIT
      * After waiting for 8 clock cycle assert eot and go to state ST\_IDLE.
  + This block contains a data input, data output, ready output and valid output.
    - Data input: Receive the data from tx fifo and transfer it to device.
    - Data output: Transmits the data received from device to rx fifo.
    - Ready output: Indicate if block is ready to write data to tx fifo.
    - Valid output: Indicate if there is valid data to be written to rx fifo.

****

**(u\_clockgen) udma\_clkgen:-**

**Ports:-**

input logic clk\_i,

input logic rstn\_i,

input logic dft\_test\_mode\_i,

input logic dft\_cg\_enable\_i,

input logic clock\_enable\_i,

input logic [7:0] clk\_div\_data\_i,

input logic clk\_div\_valid\_i,

output logic clk\_div\_ack\_o,

output logic clk\_o

**Theory of operation :-**

* This is a Integer clock divider with async configuration interface.
* The module will be in four modes namely IDLE, STOP, WAIT, RELEASE.
* The clock\_enable\_i should be high and the **module should enable the clock** for the output to be visible.
* When the module is in reset mode by making rstn\_i low,then the mode is set to IDLE.The multiplexer will select the input clock to be the output clock of the module.The clock divider is enabled.**The clock is enabled by the module**.
* The signal clk\_div\_valid\_i is sent to a pulp\_sync\_wedge module as serial\_i and output of the module is clk\_div\_valid\_sync which represents the r\_edge\_o of the pulp\_sync\_edge.
* Now at every positive edge of the clk\_i,If clk\_div\_valid\_sync is high,then the clk\_div\_data\_i is read and the clock divider value is updated.Also ,the next state of the module is change to STOP state, so that in next clock cycle as state is in STOP mode, the multiplexer sets the output to the clock divider output and schedules the update of **clock divider config** to next clock cycle by changing the next state to WAIT.The clock is disabled in this state to make the config changes.
* At the next clock edge as it is in WAIT state,here the **clock divider config** is updated ,which means the counter value is set to default 0 and output clock is made low.The next state is RELEASE state.
* In the next clock edge where the state is RELEASE,from this moment the clock divider starts working with a new clock divider value from the start(as counter i made 0).The next state is again back to IDLE state.
* In the next clock edge(IDLE state),**The clock is enabled** so that we can see the output in the output pin.The module will remain in this state and clock divider will be toggle the output clock signal after the counter reaches a value equal to **(clock divisor value -1)** and at this moment the counter also becomes 0 so that it can be incremented by one unit again at every clock edge.This will continue until again clk\_div\_valid\_sync become high again then The clock divider value is updated and the state goes to STOP mode for next clock cycle to reset the clock divisor and counter to 0.
* The clk\_o is nothing but the output signal of the clock divider.

**Pulp\_sync\_wedge:-**

**Ports:-**

input logic clk\_i,

input logic rstn\_i,

input logic en\_i,

input logic serial\_i,

output logic r\_edge\_o,

output logic f\_edge\_o,

output logic serial\_o

The module takes an input(serial\_i) and a clock signal.There is a submodule which contains a 2 bit shift register which will be storing the signal value at every clock edge by shifting to right and storing new signal value in MSB.The output of the shift register is the LSB which is connected in cascaded fashion to a module which writes the output serial\_o .At every clock positive edge , the serial\_o is updated with the current LSB of the shift register and LSB is updated by shift register with new value ,both happening in a parallel non blocking fashion.Whenever the LSB of the shift register changes to high and serial\_o was low,then r\_edge\_o is made high ,but at next clock cycle as serial\_o is updated to LSB of shift register ,r\_edge\_o goes low (So r\_edge\_o stays high for only one clock cycle).

**Edge Propagator:-**

**Ports:-**

input logic clk\_tx\_i,

input logic rstn\_tx\_i,

input logic edge\_i,

input logic clk\_rx\_i,

input logic rstn\_rx\_i,

output logic edge\_o

**Theory of operation:-**

* The main purpose of the module is to propagate the input value in edge\_i for a period of time.
* So,when rstn\_tx\_i is low, then the output is low.
* In active mode ,Whenever the edge\_i is high it is stored in a register and reflected in a signal in the next positive edge of clk\_tx\_i.The signal which is sensitive to edge\_i at every positive edge of clk\_tx\_i is sent to pulp\_sync\_wedge.This signal will remain high until we get an output from pulp\_sync\_wedge(serial\_o)( clock clk\_rx\_i ) for the signal sensitive to edge\_i.Then after three clock cycles after we get a output from pulp\_sync\_wedge we will make the signal low .After this the signal is again set sensitive to edge\_i .So everything repeats after the edge\_i is triggered again.Now the output of the edge propagator (edge\_o) is nothing but the rising edge pulp\_sync\_wedge based signal,i.e.,r\_edge\_o, which asserts one clock cycle after the signal sensitive to edge\_i is made high.

**udma\_dc\_fifo:**

**Ports:-**

input logic src\_clk\_i,

input logic src\_rstn\_i,

input logic [DATA\_WIDTH-1:0] src\_data\_i,

input logic src\_valid\_i,

output logic src\_ready\_o,

input logic dst\_clk\_i,

input logic dst\_rstn\_i,

output logic [DATA\_WIDTH-1:0] dst\_data\_o,

output logic dst\_valid\_o,

input logic dst\_ready\_i

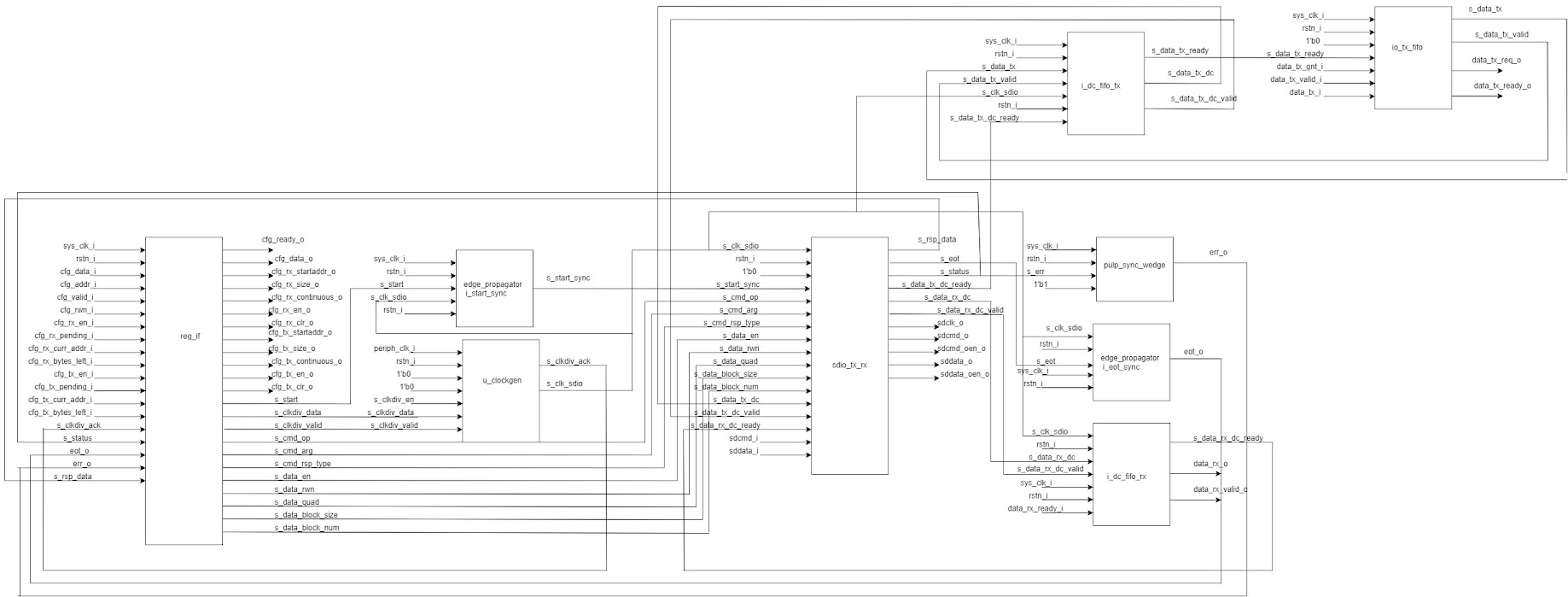
**Theory of operation:-**

* The module contains two sub modules ,one connected to the source to receive the data and enter the data into FIFO and another module connected to the destination and works on sending the FIFO data to destination.
* din:
  + This is connected to the source.
    - input clk;
    - input rstn;
    - input [DATA\_WIDTH - 1 : 0] data;
    - input valid;
    - output ready;
    - output [BUFFER\_DEPTH - 1 : 0] write\_token;
    - input [BUFFER\_DEPTH - 1 : 0] read\_pointer;
    - output [DATA\_WIDTH - 1 : 0] data\_async;
  + The din contains the dc\_buffer module which contains the actual FIFO , dc\_token\_ring module which contains the logic to compute the write pointers and finally the dc\_full\_detector module which contains the logic to deduce whether FIFO is full or not.
* dout:
  + input [DATA\_WIDTH - 1 : 0] data\_async;
  + input clk;
  + input rstn;
  + output [DATA\_WIDTH - 1 : 0] data;
  + output valid;
  + input ready;
  + input [BUFFER\_DEPTH - 1 : 0] write\_token;
  + output [BUFFER\_DEPTH - 1 : 0] read\_pointer;
  + dout contains the dc\_token\_ring module which contains the logic to compute the read\_pointer to read data from when FIFO is not empty .
  + It will also be taking write\_token from the din module and perform the logic of dc\_synchronizer on it to get the synchronized version of write\_token.This synchronized write\_token along with read\_pointer is used to deduce whether the FIFO is empty or not ,using bit manipulation.
  + The dc\_synchronizer synchronizes the write\_token with respect to clk ,which means at every clock edge the value of the write\_token is stored but it is reflected in the output in the next clock edge.
  + If the FIFO is empty then we cannot read from the FIFO.

**io\_tx\_fifo:-**

* This is a TX FIFO with outstanding request support.
* Ports:-
  + input logic clk\_i,
  + input logic rstn\_i,
  + input logic clr\_i,
  + output logic req\_o,
  + input logic gnt\_i,
  + output logic [DATA\_WIDTH-1 : 0] data\_o,
  + output logic valid\_o,
  + input logic ready\_i,
  + input logic valid\_i,
  + input logic [DATA\_WIDTH-1 : 0] data\_i,
  + output logic ready\_o
* Theory of operation:-
  + The module contains a FIFO module which is a synchronous FIFO with configurable width and depth and logic to keep record of any outstanding requests and if requests are solved and accordingly update the record.
  + The FIFO module does three things which are updating the buffer , keeping record of the number of elements in the buffer and updating the write and read pointers in the buffer.
  + If the signal valid\_i is high and the buffer is not full then at the current position at the write pointer of the buffer array the data from data\_i is stored and the write pointer is incremented.
  + If the signal ready\_i is high and the buffer is not empty then the read pointer is incremented.
  + The current read pointer position in the buffer array is read into data\_o always.The read pointer is updated at every clock positive edge as mentioned in above bullet point.
  + While above functions are happening at every clock positive edge the number of elements in the buffer is also updated and recorded after the operation.
  + Coming to the logic for the outstanding requests,We calculate the number of free spaces left in the buffer and if it is equal to the number of the outstanding requests then we signal to stop further requests.If there is no signal to stop the requests and the buffer is ready to accept requests (which means it is not full) then we can accept the requests.
  + So at every clock positive edge if the module and accept requests based on above conditions and the request is granted by making gnt\_i signal high then if valid\_i is low(meaning the request is not valid) or The buffer is not ready (meaning the buffer is full) then the number of outstanding requests will be incremented.If valid\_i is high and the buffer is ready then the number of outstanding requests will be decremented.

**Whole operation:-**

****

* As we can see in the above block diagram,some of the input signals go into the register interface and few signals are generated by the register interface which are used for various operations in SDIO.
* We have a module called u\_clockgen which takes in a few parameters from the register and generates a sdio clock from the peripheral clock (u\_clockgen is a Integer clock divider with async configuration interface).
* The SDIO clock generated will be used in the SDIO module.There is a edge propagator module which takes in s\_start from register which is in sync with system clock sys\_clk\_i and sdio clock as input ,finally generates the resign edge of the s\_start in sync with the sdio clock ,So this module generates resign edge of s\_start in sync with the sdio clock instead of sys\_clk\_i.
* The s\_start\_sync ,sdio clock and signals from registers go to sdio\_tx\_rx where actual logic for transmission and receiving is executed.sdio\_tx\_rx runs in sync with sdio clock.
* The signal s\_err from the module sdio\_tx\_rx is synchronized to sys\_clk\_i using the module pulp\_sync\_wedge and set to register to be stored.
* The s\_eot signal from sdio\_tx\_\_rx is in sync with sdio clock ,there is edge propagator with name i\_eot\_sync which generates the rising edge of the signal s\_eot in sync with sys\_clk\_i and the synchronized signal is sent to register to be stored.
* For communication between( SDIO and uDMA )and (SDIO and I/O) we use FIFOs.
* FIFOs for SDIO and uDMA:-
  + There are three FIFOs in total , i\_dc\_fifo\_tx , u\_dc\_fifo\_rx and io\_tx\_fifo.
  + **u\_dc\_fifo\_rx** is a **udma\_dc\_fifo** which is used to transfer the data from the clock domain of sdio clock domain(source) and sys\_clk\_i (system clock)(destination).So the data from peripheral I/O gets into sdio\_tx\_rx module, to communicate this received data from the sdio\_tx\_rx to uDMA we use this FIFO **u\_dc\_fifo\_rx.**
  + **i\_dc\_fifo\_tx** is different clock FIFO between sys\_ck\_i(source) and sdio clock(destination) .So basically The data which comes as input to the top module from uDMA is sent first to **io\_tx\_fifo** which is a TX FIFO with outstanding request support.The data in this FIFO goes to **i\_dc\_fifo\_tx** which is FIFO between sys\_clk\_i as source and sdio clock as destination ,So through this FIFO named **i\_dc\_fifo\_tx** ,the data can be sent to the **sdio\_tx\_rx** module in SDIO.So the data from the uDMA first gets into the FIFO named **io\_tx\_fifo** and then from there into a FIFO named **i\_dc\_fifo\_tx**  and finally to **sdio\_tx\_rx**.