IMPLEMENTATION OF SD CONTROLLER USING FPGA(NEXYS 4)

Start Date: 12th June, 2017
Finished By: 5th July,2017

Project done by: Navneeth Narayanan (108115058)
K Ashish Kumar(108115038)
Apratim Khobragade (108115014)
Prakhar Gupta (108115064)

Under the Guidance of Dr. G. Lakshmi Narayanan
Assisted by Mr. Bibin Sam Paul

INDEX

Sl.No	TITLE
1.	Project Overview
2.	Serial Peripheral Interface
3.	FAT 32 System
4.	VHDL Code
5.	Output

PROJECT OVERVIEW

The objective of this project is to interface an sd card with an fpga board. This project involved the usage of a micro sd card and a nexys4 ddr fpga board.The software used included xilinx (platform for vhdl coding) , Hex Editor (to access the different sectors of the sd card) and the adept software (TO use the nexys board).The code for doing this project involves the usage of four main modules dealing with the following :-

1) SD Controller
2) RAM
3) CPU
4) DISPLAY

The interfacing is basically done in two stages , one being interfacing the FPGA board with the SD controller and the other the controller with the microSD card. This is where the SD controller module comes into play.The module specifically deals with the latter part of the interfacing , namely the interfacing of the controller with the sd card. It deals with the connections to and the control of the various parts of the microsd such as the MOSI (Master Out Slave In) , the MISO (Master In Slave Out) , and the chip select among others. This is then linked with the main module, which is the CPU module.

RAM module:

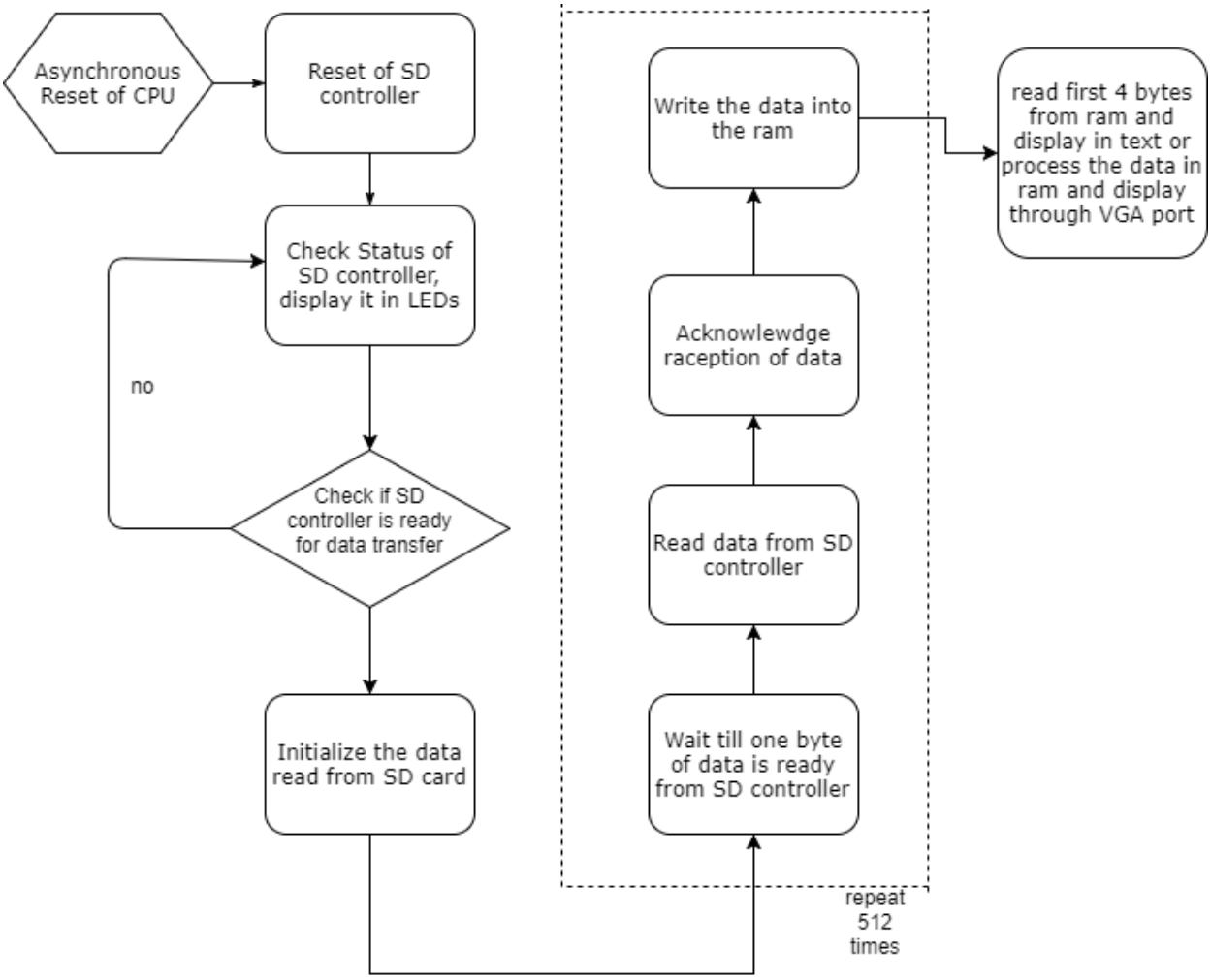
The incoming data from sd card is 512 bytes long per read cycle. So we need to to store these 512 bytes for manipulation. The RAM module has clock, read and write address, and write enable as inputs and has two seperate 8-bit parallel lines for data input and output. The data from RAM is changed according to the read address at the frequency of the input clock, so after the CPU gives read address, it must wait at least for one clock cycle before reading the data. However, the write operation in asynchronous the processor puts the required data and write address and then gives a high pulse on the write enable signal.

Display module:

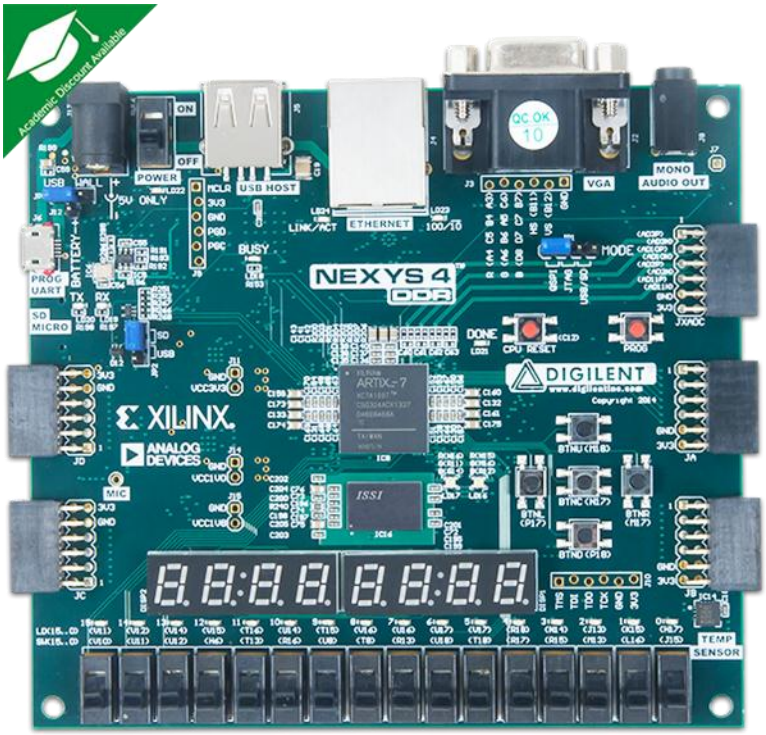
The 8 digit seven segment display module has active low inputs for anode and cathode, The anode control selects which digit of the 7 segment display is illuminated, and the cathode combination controls what pattern is shown in the display. A clock is given to this module which changes which digit of the 7 segment is illuminated. Other input is the 32 bit data line. The cathodes are controlled by the data occupying the place for the current illuminated digit in the 32 bit data line.

CPU FSM:

The CPU joins the SD controller, clock, ram and display peripheral. It is implemented as an FSM. The states of SD controller and CPU are debugged in the 7 segment display. As the CPU is reset, it resets the SD controller and waits till it is ready for read or write operations. And displays the process in the 7 seg display. Then the CPU issues a read command from a hard programmed block address (which is found from the hex editor in laptop). Then it read 512 bytes of DATA from the SD controller. While reading the data from the controller, it has to wait for the data ready pin to go high, then it has to read the data from the 8 bit data pins. Then the CPU has to acknowledge through the data acknowledge pin and store the data in the proper address of the ram. After all 512 bytes are read the CPU read the first fr bytes from RAMM and displays it in the 7 segment display. The image bytes can also be displayed as an image in the monitor through the VGA port.



NEXYS 4 FPGA :



Featuring the same Artix™-7 field programmable gate array (FPGA) from Xilinx®, the Nexys 4 DDR is a ready-to-use digital circuit development platform designed to bring additional industry applications into the classroom environment. The Artix-7 FPGA is optimized for high-performance logic, and offers more capacity, higher performance, and more resources than earlier designs. With its large, high-capacity FPGA (Xilinx part number XC7A100T-1CSG324C) and collection of USB, Ethernet, and other ports, the Nexys 4 DDR can host designs ranging from introductory combinational circuits to powerful embedded processors. Several built-in peripherals, including an accelerometer, a temperature sensor, MEMs digital microphone, speaker amplifier, and plenty of I/O devices allow the Nexys 4 DDR to be used for a wide range of designs without needing any other components. The most notable improvement is the replacement of the 16 MiB CellularRAM with a 128 MiB DDR2 SDRAM memory. Digilent will provide a VHDL reference module that wraps the complexity of a DDR2 controller and is backwards compatible with the asynchronous SRAM interface of the CellularRAM, with certain limitations. Some of the peripherals that the Nexys 4 has are given below:

- UART/JTAG USB port
- Pmod port for XADC signals
- Audio connector
- Ethernet connector
- USB host connector
- microSD card connector
- 12-bit VGA output
- Four Pmod ports
- Power jack

And some of the features of this fpga are as follows:

- USB-UART Bridge
- 10/100 Ethernet PHY
- PWM audio output
- 3-axis accelerometer
- 16 user switches
- 16 user LEDs
- Two tri-color LEDs
- PDM microphone
- Temperature sensor
- Two 4-digit 7-segment displays
- USB HID Host for mice, keyboards and memory sticks
- Pmod for XADC signals
- 12-bit VGA output

SERIAL PERIPHERAL INTERFACE

- The ***Serial Peripheral Interface bus (SPI)*** is a synchronous serial communication interface specification used for short distance communication, primarily in embedded systems. Typical applications include ***Secure Digital cards*** and ***liquid crystal displays*****.**
- SPI devices communicate in full duplex mode using a ***master-slave architecture with a single master***. The master device originates the frame for reading and writing. Multiple slave devices are supported through selection with individual slave select (SS) lines.

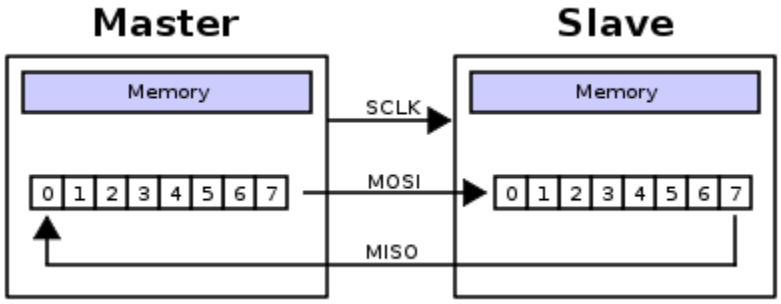
- The SPI bus specifies four logic signals:
  - ***SCLK***: Serial Clock (output from master).
  - ***MOSI***: Master Output Slave Input, or Master Out Slave In (data output from master).
  - ***MISO***: Master Input Slave Output, or Master In Slave Out (data output from slave).
  - ***SS***: Slave Select (often active low, output from master).

DATA TRANSMISSION:

- To begin communication, the bus master configures the clock, using a frequency supported by the slave device, typically up to a few MHz. The master then selects the slave device with a logic level 0 on the select line.
- During each SPI clock cycle, a full duplex data transmission occurs. The master sends a bit on the MOSI line and the slave reads it, while the slave sends a bit on the MISO line and the master reads it.
- Transmissions normally involve two shift registers of some given word size, such as eight bits, one in the master and one in the slave; they are connected in a virtual ring topology. Data is usually shifted out with the most-significant bit first, while shifting a new least-significant bit into the same register. At the same time, Data from the counterpart is shifted into the least-significant bit register. After the register bits have been shifted out and in, the master and slave have exchanged

register values. If more data needs to be exchanged, the shift registers are reloaded and the process repeats.

- Transmission may continue for any number of clock cycles. When complete, the master stops toggling the clock signal, and typically deselects the slave.



SPI COMMAND SET :

- Each command is expressed in abbreviation like GO_IDLE_STATE or CMD<n>, <n> is the number of the command index and the value can be 0 to 63.

Command Index	Argument	Response	Data	Abbreviation	Description
CMD0	None(0)	R1	No	GO_IDLE_STATE	Software reset.
CMD1	None(0)	R1	No	SEND_OP_COND	Initiate initialization process.
ACMD41(*1)	*2	R1	No	APP_SEND_OP_COND	For only SDC. Initiate initialization process.
CMD8	*3	R7	No	SEND_IF_COND	For only SDC V2. Check voltage range.
CMD9	None(0)	R1	Yes	SEND_CSD	Read CSD register.
CMD10	None(0)	R1	Yes	SEND_CID	Read CID register.
CMD12	None(0)	R1b	No	STOP_TRANSMISSION	Stop to read data.
CMD16	Block length[31:0]	R1	No	SET_BLOCKLEN	Change R/W block size.
CMD17	Address[31:0]	R1	Yes	READ_SINGLE_BLOCK	Read a block.
CMD18	Address[31:0]	R1	Yes	READ_MULTIPLE_BLOCK	Read multiple blocks.
CMD23	Number of blocks[15:0]	R1	No	SET_BLOCK_COUNT	For only MMC. Define number of blocks to transfer with next multi-block read/write command.
ACMD23(*1)	Number of blocks[22:0]	R1	No	SET_WR_BLOCK_ERASE_COUNT	For only SDC. Define number of blocks to pre-erase with next multi-block write command.
CMD24	Address[31:0]	R1	Yes	WRITE_BLOCK	Write a block.
CMD25	Address[31:0]	R1	Yes	WRITE_MULTIPLE_BLOCK	Write multiple blocks.
CMD55(*1)	None(0)	R1	No	APP_CMD	Leading command of ACMD<n> command.
CMD58	None(0)	R3	No	READ_OCR	Read OCR.

*1:ACMD<n> means a command sequense of CMD55-CMD<n>.
*2: Rsv(0)[31], HCS[30], Rsv(0)[29:0]
*3: Rsv(0)[31:12], Supply Voltage(1)[11:8], Check Pattern(0xAA)[7:0]

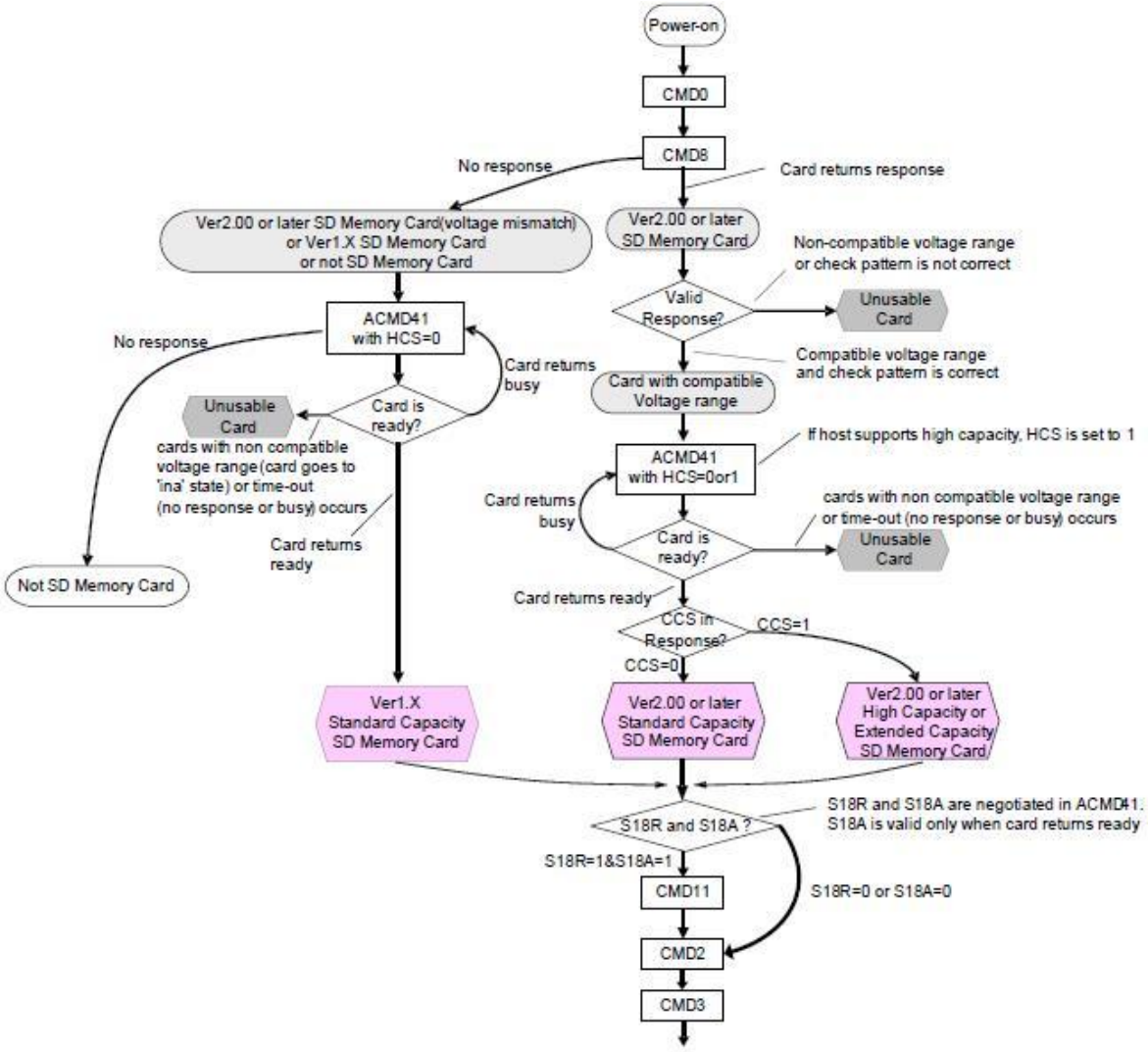


Figure 4-2: Card Initialization and Identification Flow (SD mode)

FAT32 SYSTEM

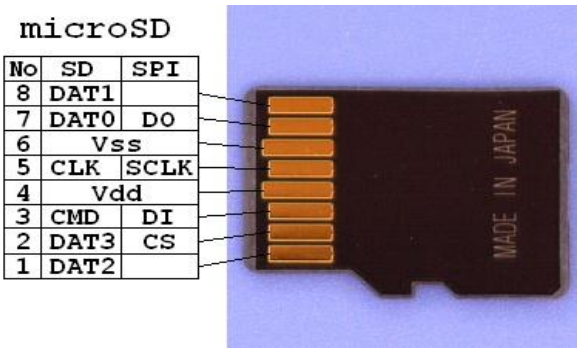
- FAT32 is a file system for storage devices (such as hard drives , USB drives and solid state drives) with 32-bit File Allocation Table.
- In the place of 2 bytes or 16 bits per FAT entry (as in FAT16) FAT32 uses 4 bytes or 32 bits.

FAT LAYOUT :

- First the ***Boot sector*** (at relative address 0), and possibly other stuff. Together these are the Reserved Sectors. Usually the boot sector is the only reserved sector.
- Then the ***FATs*** (following the reserved sectors; the number of reserved sectors is given in the boot sector, bytes 14-15; the length of a sector is found in the boot sector, bytes 11-12).
- Then the ***Root Directory*** (following the FATs; the number of FATs is given in the boot sector, byte 16; each FAT has a number of sectors given in the boot sector, bytes 22-23).
- Finally the ***Data Area*** (following the root directory; the number of root directory entries is given in the boot sector, bytes 17-18, and each directory entry takes 32 bytes; space is rounded up to entire sectors).

SD CARD

- The ***Secure Digital Memory Card(SDC)*** is the de facto standard memory card for mobile equipments. The pinout and interfacing of SD Card with Nexys 4 DDR is shown below:



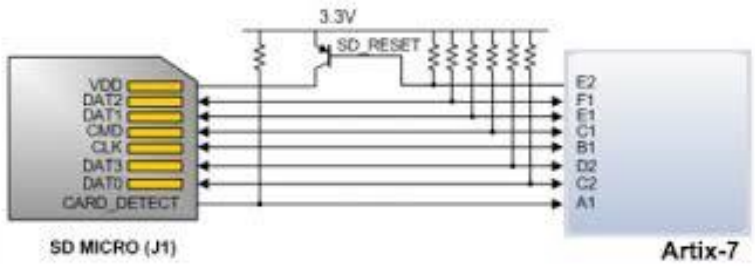


Figure 21. Artix-7 microSD card connector interface (PIC24 connections not shown).

CPU vhdl code

```
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity cpu is
    Port ( main_clock : in  STD_LOGIC;
           seven_seg_a : out  STD_LOGIC_VECTOR (7 downto 0);
           seven_seg_c : out  STD_LOGIC_VECTOR (6 downto 0);
                     csm : out std_logic;
                     mosim : out std_logic;
                     misom : in std_logic;
                     sclkm : out std_logic;
                     sd_err_p : out std_logic;
                     sd_busy_p : out std_logic;
                     sd_error_code_p : out std_logic_vector(2 downto 0);
                     sd_type_p : out std_logic_vector(1 downto 0);
                     resetp : in std_logic;
                     r_main_switch : in std_logic;
                     read_complete_led : out std_logic;
                     data_out_available_led : out std_logic;
                     read_byte_once : out std_logic := '0'
                     --cst : out std_logic
                     );
end cpu;

architecture rtl of cpu is
signal rdp : std_logic;
signal wrp : std_logic;
signal dm_inp : std_logic;
signal dinp : std_logic_vector(7 downto 0);
signal doutp : std_logic_vector(7 downto 0);
--signal clkp : std_logic;

signal card_present_m: std_logic := '1';
signal card_wr_p_m: std_logic := '0';
```

```
signal rdm_d: std_logic := '0';
signal dout_avail_d : std_logic;
signal dout_taken_p : std_logic;
signal wr_m_d : std_logic := '0';
signal din_valid_p : std_logic;
signal din_taken_d : std_logic;
signal addr_p : std_logic_vector (31 downto 0);
signal erase_cnt_d : std_logic_vector(7 downto 0);
signal sd_fsm_p : std_logic_vector(7 downto 0);
signal text : std_logic_vector(31 downto 0);
signal led_clk : std_logic;
signal clock_rst : std_logic;
signal temp : std_logic_vector(7 downto 0);
signal sd_clk : std_logic;
signal clk_r : std_logic;
signal we_r : std_logic; -- write enable signal
signal wadd_r : std_logic_vector(8 downto 0); -- write address to store the data into
ram
signal radd_r : std_logic_vector(8 downto 0); -- read address to read the data from
the ram
signal data_in_r : std_logic_vector(7 downto 0); -- input data to store into ram
signal data_out_r : std_logic_vector(7 downto 0); -- output data from memory
type s_main is (reset_main, init_main, read_main, read_main_wait, read_byte_wait,
read_byte, read_complete, display_txt, write_to_ram, finish, read_from_ram,
delay_state);
signal main_state : s_main;
signal return_state : s_main;
signal sector_no : std_logic_vector(31 downto 0);
signal sd_busy_ram : std_logic;

begin
	display: entity work.display_value
		PORT MAP(
		value_in => text,
		aout => temp,
		cout => seven_seg_c,
		led_refresh_clk => led_clk
	);

	clock_peripheral: entity work.clock
		PORT MAP(
		refclk => main_clock,
		clk1 => clk_r,
		clk2 => led_clk,
		rst => clock_rst,
		clk3 => sd_clk
	);
	seven_seg_a <= not temp;
```

```
--text(7 downto 0) <= sd_fsm_p(7 downto 0);

sd_controller_1: entity work.sd_controller
       PORT MAP (
       cs => csm,
       mosi =>mosim,
       miso => misom,
       sclk =>sclkm,
       card_present => card_present_m,
       card_write_prot => card_wr_p_m,
       rd => rdp,
       rd_multiple => rdm_d,
       dout => doutp,
       dout_avail => dout_avail_d,
       dout_taken => dout_taken_p,
       wr => wrp,
       wr_multiple =>wr_m_d,
       din => dinp,
       din_valid => din_valid_p,
       din_taken => din_taken_d,
       addr => addr_p,
       erase_count => erase_cnt_d,
       sd_error => sd_err_p,
       --sd_busy => sd_busy_p,
       sd_busy => sd_busy_ram,
       sd_error_code => sd_error_code_p,
       reset => resetp,
       clk => sd_clk,
       sd_type => sd_type_p,
       sd_fsm => sd_fsm_p
       );

       our_ram : entity work.ram
       PORT MAP(
       Clk => clk_r,
       we => we_r,
       wadd => wadd_r,
       radd => radd_r,
       data_in => data_in_r,
       data_out => data_out_r
);

main : process(main_clock)
variable byte_no : unsigned(8 downto 0) := to_unsigned(0, 9);
variable check_init : unsigned(7 downto 0);
variable ram_clk_cntr : unsigned(6 downto 0) := to_unsigned(0, 7);
variable delay : unsigned(15 downto 0) := to_unsigned(0, 16);
```

```
begin
if rising_edge(main_clock) then
if (r_main_switch = '1') then
main_state <= reset_main;
end if;
check_init := unsigned(sd_fsm_p(7 downto 0));


case main_state is

when reset_main =>
byte_no := to_unsigned(0, 9);
rdp <= '0';
read_complete_led <= '0';
main_state <= init_main;
text(15 downto 8) <= x"01";



when init_main =>
if check_init = 17 then
main_state <= read_main_wait;
end if;
sector_no <= x"00004100";
addr_p <= sector_no;
text(15 downto 8) <= x"02";

when read_main_wait =>
if (sd_busy_ram = '0') then
main_state <= read_main;
end if;
text(15 downto 8) <= x"03";


when read_main =>
dout_taken_p <= '0';
rdp <= '1';
main_state <= delay_state;
delay := to_unsigned(10000, 16);
return_state <= read_byte_wait;

when read_byte_wait =>
dout_taken_p <= '0';
data_out_available_led <= dout_avail_d; --temp led
if(dout_avail_d = '1') then
main_state <= delay_state;
delay := to_unsigned(10000, 16);
return_state <= read_byte;
```

```
end if;

when read_byte =>
read_byte_once <= '1'; --temp led
data_in_r <= doutp;

if (byte_no = 1) then
text(31 downto 24) <= data_in_r;
elsif (byte_no = 2) then
text(23 downto 16) <= data_in_r;
elsif (byte_no = 3) then
text(15 downto 8) <= data_in_r;
elsif (byte_no = 4) then
text(7 downto 0) <= data_in_r;
end if;
dout_taken_p <= '1';
wadd_r <= std_logic_vector(byte_no);
return_state <= read_byte_wait;
main_state <= delay_state;
delay := to_unsigned(10000, 16);
ram_clk_cntr := to_unsigned(0, 7);
byte_no := byte_no + 1;
--text(15 downto 8) <= x"06";

when read_complete =>
read_complete_led <= '1';

--text(15 downto 8) <= x"07";
main_state <= display_txt;

when display_txt =>
radd_r <= std_logic_vector(byte_no);
return_state <= display_txt;
main_state <= read_from_ram;
ram_clk_cntr := to_unsigned(0, 7);
if (byte_no = 511) then
return_state <= finish;
end if;


when read_from_ram =>
if(ram_clk_cntr = 127) then
main_state <= return_state;
byte_no := byte_no + 1;
end if;

when write_to_ram =>
we_r <= '1';
```

```
        ram_clk_cntr := ram_clk_cntr + 1;
        if (ram_clk_cntr = 127) then
        main_state <= return_state;
        we_r <= '0';
        end if;
        if(byte_no = 511) then
        we_r <= '0';
        main_state <= read_complete;
        byte_no := to_unsigned(0, 9);
        end if;

        when finish =>

        when delay_state =>
        if(delay = 0) then
        main_state <= return_state;
        end if;
        delay := delay - 1;

        end case;
        end if;
        --text(28 downto 20) <= std_logic_vector(byte_no(8 downto 0));
        end process main;


end rtl;
```

7 segment display driver code

```
----------------------------------------------------------------------------------
-
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity display_value is
    Port ( value_in : in  STD_LOGIC_VECTOR (31 downto 0);
           aout : out  STD_LOGIC_VECTOR (7 downto 0);
           cout : out  STD_LOGIC_VECTOR (6 downto 0);
                     led_refresh_clk : in std_logic
                     );
end display_value;

architecture rtl of display_value is
       type led_state is (d0, d1, d2, d3, d4, d5, d6, d7);
       signal led_dig : led_state;
       --signal led_refresh_clk : std_logic;
       signal value_in_1 : STD_LOGIC_VECTOR (31 downto 0);
begin


--firstpart <= allparts(15 downto 8);
get_digits: process(value_in)
                          begin
                          value_in_1 <= value_in;
                          end process get_digits;

showin7seg : process(led_refresh_clk)
                          begin
                          if rising_edge(led_refresh_clk) then
                          case led_dig is

                          when d0=>
                                 aout <= "00000001" ;
                                 case value_in_1(3 downto 0) is
                                        when "0000"=>
                                               cout <= "1000000";
                                        when "0001"=>
```

```
                        cout <= "1111001";
                when "0010"=>
                        cout <= "0100100";
                when "0011"=>
                        cout <= "0110000";
                when "0100"=>
                        cout <= "0011001";
                when "0101"=>
                        cout <= "0010010";
                when "0110"=>
                        cout <= "0000010";
                when "0111"=>
                        cout <= "1111000";
                when "1000"=>
                        cout <= "0000000";
                when "1001"=>
                        cout <= "0010000";
                when "1010"=>
                        cout <= "0001000";
                when "1011"=>
                        cout <= "0000011";
                when "1100"=>
                        cout <= "1000110";
                when "1101"=>
                        cout <= "0100001";
                when "1110"=>
                        cout <= "0000110";
                when "1111"=>
                        cout <= "0001110";
                when others=>
                        cout <= "0001000";
                end case;
        led_dig <= d1;

when d1=>
        aout <= "00000010" ;
        case value_in_1(7 downto 4) is
                when "0000"=>
                        cout <= "1000000";
                when "0001"=>
                        cout <= "1111001";
                when "0010"=>
                        cout <= "0100100";
                when "0011"=>
                        cout <= "0110000";
                when "0100"=>
                        cout <= "0011001";
                when "0101"=>
```

```
                        cout <= "0010010";
                    when "0110"=>
                        cout <= "0000010";
                    when "0111"=>
                        cout <= "1111000";
                    when "1000"=>
                        cout <= "0000000";
                    when "1001"=>
                        cout <= "0010000";
                    when "1010"=>
                        cout <= "0001000";
                    when "1011"=>
                        cout <= "0000011";
                    when "1100"=>
                        cout <= "1000110";
                    when "1101"=>
                        cout <= "0100001";
                    when "1110"=>
                        cout <= "0000110";
                    when "1111"=>
                        cout <= "0001110";
                    when others=>
                        cout <= "0001000";
                    end case;
                led_dig <= d2;

            when d2=>
                aout <= "00000100" ;
                case value_in_1(11 downto 8) is
                    when "0000"=>
                        cout <= "1000000";
                    when "0001"=>
                        cout <= "1111001";
                    when "0010"=>
                        cout <= "0100100";
                    when "0011"=>
                        cout <= "0110000";
                    when "0100"=>
                        cout <= "0011001";
                    when "0101"=>
                        cout <= "0010010";
                    when "0110"=>
                        cout <= "0000010";
                    when "0111"=>
                        cout <= "1111000";
                    when "1000"=>
                        cout <= "0000000";
                    when "1001"=>
```

```
                        cout <= "0010000";
                    when "1010"=>
                        cout <= "0001000";
                    when "1011"=>
                        cout <= "0000011";
                    when "1100"=>
                        cout <= "1000110";
                    when "1101"=>
                        cout <= "0100001";
                    when "1110"=>
                        cout <= "0000110";
                    when "1111"=>
                        cout <= "0001110";
                    when others=>
                        cout <= "0001000";
                    end case;
                led_dig <= d3;

            when d3=>
                aout <= "00001000" ;
                case value_in_1(15 downto 12) is
                    when "0000"=>
                        cout <= "1000000";
                    when "0001"=>
                        cout <= "1111001";
                    when "0010"=>
                        cout <= "0100100";
                    when "0011"=>
                        cout <= "0110000";
                    when "0100"=>
                        cout <= "0011001";
                    when "0101"=>
                        cout <= "0010010";
                    when "0110"=>
                        cout <= "0000010";
                    when "0111"=>
                        cout <= "1111000";
                    when "1000"=>
                        cout <= "0000000";
                    when "1001"=>
                        cout <= "0010000";
                    when "1010"=>
                        cout <= "0001000";
                    when "1011"=>
                        cout <= "0000011";
                    when "1100"=>
                        cout <= "1000110";
                    when "1101"=>
```

```
                        cout <= "0100001";
                    when "1110"=>
                        cout <= "0000110";
                    when "1111"=>
                        cout <= "0001110";
                    when others=>
                        cout <= "0001000";
                    end case;

            led_dig <= d4;

        when d4=>
            aout <= "00010000" ;
            case value_in_1(19 downto 16) is
                    when "0000"=>
                        cout <= "1000000";
                    when "0001"=>
                        cout <= "1111001";
                    when "0010"=>
                        cout <= "0100100";
                    when "0011"=>
                        cout <= "0110000";
                    when "0100"=>
                        cout <= "0011001";
                    when "0101"=>
                        cout <= "0010010";
                    when "0110"=>
                        cout <= "0000010";
                    when "0111"=>
                        cout <= "1111000";
                    when "1000"=>
                        cout <= "0000000";
                    when "1001"=>
                        cout <= "0010000";
                    when "1010"=>
                        cout <= "0001000";
                    when "1011"=>
                        cout <= "0000011";
                    when "1100"=>
                        cout <= "1000110";
                    when "1101"=>
                        cout <= "0100001";
                    when "1110"=>
                        cout <= "0000110";
                    when "1111"=>
                        cout <= "0001110";
                    when others=>
                        cout <= "0001000";
```

```
                end case;
        led_dig <= d5;

when d5=>
        aout <= "00100000" ;
        case value_in_1(23 downto 20) is
                when "0000"=>
                        cout <= "1000000";
                when "0001"=>
                        cout <= "1111001";
                when "0010"=>
                        cout <= "0100100";
                when "0011"=>
                        cout <= "0110000";
                when "0100"=>
                        cout <= "0011001";
                when "0101"=>
                        cout <= "0010010";
                when "0110"=>
                        cout <= "0000010";
                when "0111"=>
                        cout <= "1111000";
                when "1000"=>
                        cout <= "0000000";
                when "1001"=>
                        cout <= "0010000";
                when "1010"=>
                        cout <= "0001000";
                when "1011"=>
                        cout <= "0000011";
                when "1100"=>
                        cout <= "1000110";
                when "1101"=>
                        cout <= "0100001";
                when "1110"=>
                        cout <= "0000110";
                when "1111"=>
                        cout <= "0001110";
                when others=>
                        cout <= "0001000";
                end case;
        led_dig <= d6;

when d6=>
        aout <= "01000000" ;
        case value_in_1(27 downto 24) is
                when "0000"=>
                        cout <= "1000000";
```

```
                when "0001"=>
                       cout <= "1111001";
                when "0010"=>
                       cout <= "0100100";
                when "0011"=>
                       cout <= "0110000";
                when "0100"=>
                       cout <= "0011001";
                when "0101"=>
                       cout <= "0010010";
                when "0110"=>
                       cout <= "0000010";
                when "0111"=>
                       cout <= "1111000";
                when "1000"=>
                       cout <= "0000000";
                when "1001"=>
                       cout <= "0010000";
                when "1010"=>
                       cout <= "0001000";
                when "1011"=>
                       cout <= "0000011";
                when "1100"=>
                       cout <= "1000110";
                when "1101"=>
                       cout <= "0100001";
                when "1110"=>
                       cout <= "0000110";
                when "1111"=>
                       cout <= "0001110";
                when others=>
                       cout <= "0001000";
                end case;
        led_dig <= d7;

when d7=>
        aout <= "10000000" ;
        case value_in_1(31 downto 28) is
                when "0000"=>
                       cout <= "1000000";
                when "0001"=>
                       cout <= "1111001";
                when "0010"=>
                       cout <= "0100100";
                when "0011"=>
                       cout <= "0110000";
                when "0100"=>
                       cout <= "0011001";
```

```
                        when "0101"=>
                               cout <= "0010010";
                        when "0110"=>
                               cout <= "0000010";
                        when "0111"=>
                               cout <= "1111000";
                        when "1000"=>
                               cout <= "0000000";
                        when "1001"=>
                               cout <= "0010000";
                        when "1010"=>
                               cout <= "0001000";
                        when "1011"=>
                               cout <= "0000011";
                        when "1100"=>
                               cout <= "1000110";
                        when "1101"=>
                               cout <= "0100001";
                        when "1110"=>
                               cout <= "0000110";
                        when "1111"=>
                               cout <= "0001110";
                        when others=>
                               cout <= "0001000";
                        end case;
                  led_dig <= d0;



            end case;
            end if;
            end process showin7seg;

end rtl;
```

Code for Clock peripheral in CPU

```
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity clock is
    Port ( refclk : in  STD_LOGIC;
           clk1 : out  STD_LOGIC;
           clk2 : out  STD_LOGIC;
           clk3 : out  STD_LOGIC;
                     rst : in STD_LOGIC
           );
end clock;

architecture rtl of clock is

constant clk1_max : natural := 10; --10 MHz
constant clk2_max : natural := 100000; --100 Hz
constant clk3_max : natural := 10; --10 MHz

begin

clk_output : process(refclk, rst)
variable count1 : natural range 0 to clk1_max;
variable count2 : natural range 0 to clk2_max;
variable count3 : natural range 0 to clk3_max;

       begin
       if rst = '1' then
      count1 := 0;
             count2 := 0;
             count3 := 0;
      clk1 <= '0';
             clk2 <= '0';
             clk3 <= '0';
       elsif rising_edge(refclk) then
            if count1 < clk1_max/2 then
                count1 := count1 + 1;
                clk1 <= '0';
```

```
            elsif count1 < clk1_max then
                clk1 <= '1';
                count1 := count1 + 1;
            else
                clk1 <= '0';
                count1 := 0;
            end if;

                        if count2 < clk2_max/2 then
                count2 := count2 + 1;
                clk2 <= '0';
            elsif count2 < clk2_max then
                clk2 <= '1';
                count2 := count2 + 1;
            else
                clk2 <= '0';
                count2 := 0;
            end if;

                        if count3 < clk3_max/2 then
                count3 := count3 + 1;
                clk3 <= '0';
            elsif count3 < clk3_max then
                clk3 <= '1';
                count3 := count3 + 1;
            else
                clk3 <= '0';
                count3 := 0;
            end if;
    end if;
end process clk_output;

end rtl;
```

Code for RAM in CPU

```
----------------------------------------------------------------------------------
-
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity ram is
Port ( Clk : in std_logic; -- processing clock
we : in std_logic; -- write enable signal
wadd : in std_logic_vector(8 downto 0); -- write address to store the data into ram
radd : in std_logic_vector(8 downto 0); -- read address to read the data from the ram
data_in : in std_logic_vector(7 downto 0); -- input data to store into ram
data_out : out std_logic_vector(7 downto 0)
); -- output data from memory
end ram;
architecture rtl of ram is
------------------------------------ RAM declaration
type ram_1 is array(511 downto 0) of std_logic_vector(7 downto 0);
signal ram1_1 : ram_1;
------------------------------------ Signal declaration
signal r_add : std_logic_vector(8 downto 0);

begin

process(Clk, we)
begin
if Clk'event and Clk = '1' then
if we = '1' then -- In this process writing the input data into ram
ram1_1(conv_integer(wadd)) <= data_in;
end if;
r_add <= radd;
end if;
end process;

data_out <= ram1_1(conv_integer(r_add)); -- Reading the data from RAM

end rtl;
```

Code for SD controller

```
----------------------------------------------------------------------------------
-
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- sd_busy:
-- Inactive when the card can accept a Read or Write command
-- Goes active for the duration of the command, input address is latched at this time
-- Goes inactive when Rd or Wr is dropped, or when command is complete, whichever is
later
--
-- sd_error:
-- Goes active immediately when an error is detected
-- Resets when RD or WR is raised for the next command (except for 110 or 111 status)
--
-- sd_error_code:
-- 000 No error (operation complete)
-- 001 SD Card R1 error (R1 bit 6-0)
-- 010 Read CRC error or Write Timeout error
-- 011 Data Response Token error (Token bit 3)
-- 100 Data Error Token error (Token bit 3-0)
-- 101 SD Card Write Protect switch
-- 110 Unusable SD card
-- 111 No SD card (no response from CMD0)
--
-- sd_type:
-- 00 No card
-- 01 SD V1
-- 10 SD V2
-- 11 SDHC

entity sd_controller is
generic (
      clockRate : integer := 50000000;       -- Incoming clock is 50MHz
      slowClockDivider : integer := 64;      -- Basic clock is 25MHz, slow clock for
       R1_TIMEOUT : integer := 10;
      WRITE_TIMEOUT : integer range 0 to 999 := 500
      );
port (
      cs : out std_logic;                    -- To SD card
      mosi : out std_logic;                  -- To SD card
      miso : in std_logic;                   -- From SD card
      sclk : out std_logic;                  -- To SD card
      card_present : in std_logic;     -- From socket - can be fixed to '1' if no
switch is present
```

```
      card_write_prot : in std_logic;  -- From socket - can be fixed to '0' if no
switch is present, or '1' to make a Read-Only interface

      rd : in std_logic;                    -- Trigger single block read
      rd_multiple : in std_logic;           -- Trigger multiple block read
      dout : out std_logic_vector(7 downto 0);      -- Data from SD card
      dout_avail : out std_logic;           -- Set when dout is valid
      dout_taken : in std_logic;       -- Acknowledgement for dout

      wr : in std_logic;                    -- Trigger single block write
      wr_multiple : in std_logic;           -- Trigger multiple block write
      din : in std_logic_vector(7 downto 0);  -- Data to SD card
      din_valid : in std_logic;        -- Set when din is valid
      din_taken : out std_logic;       -- Ackowledgement for din

      addr : in std_logic_vector(31 downto 0);      -- Block address
      erase_count : in std_logic_vector(7 downto 0); -- For wr_multiple only

      sd_error : out std_logic;        -- '1' if an error occurs, reset on next RD
or WR
      sd_busy : out std_logic;         -- '0' if a RD or WR can be accepted
      sd_error_code : out std_logic_vector(2 downto 0); -- See above, 000=No error


      reset : in std_logic;     -- System reset
      clk : in std_logic;       -- twice the SPI clk (max 50MHz)

      -- Optional debug outputs
      sd_type : out std_logic_vector(1 downto 0);   -- Card status (see above)
      sd_fsm : out std_logic_vector(7 downto 0) := "11111111" -- FSM state (see
block at end of file)
);

end sd_controller;

architecture rtl of sd_controller is
type states is (
      RST, RST2,                                    -- Initial FSM resetting
      INIT,                                             -- Send initial clock
pulses
      CMD0,                                             -- Send CMD0
      CMD8, CMD8R1, CMD8B2, CMD8B3, CMD8B4, CMD8GOTB4,    -- Send CMD8
      CMD55,                                        -- Send CMD55
      CMD41,                                        -- Send ACMD41
      POLL_CMD,                                     -- Wait for card initialised
      CMD58, CMD58R1, CMD58B2, CMD58B3, CMD58B4,    -- Send CMD58
      CMD59, CMD59R1,                          -- Send CMD59
```

```
        IDLE, IDLE2,                            -- wait for read or write pulse
        READ_BLOCK,                                 -- Initiate Read command
        READ_MULTIPLE_BLOCK,                    -- Initiate Read Multiple command
        READ_BLOCK_R1, READ_BLOCK_WAIT_CHECK,   -- Wait for data to appear
        READ_BLOCK_DATA,                        -- Receive bytes and output
        READ_BLOCK_SKIP,                        -- Skip remaining data if read is
aborted
        READ_BLOCK_CRC,                         -- Receive CRC bytes
        READ_BLOCK_CHECK_CRC,            -- Check final CRC=0
        READ_BLOCK_FINISH,               -- Wait until RD drops
        READ_MULTIPLE_BLOCK_STOP,
        READ_MULTIPLE_BLOCK_STOP_2,

        SEND_RCV,
        SEND_RCV_CLK1,
        SEND_CMD,
        SEND_CMD_1,
        SEND_CMD_2,
        SEND_CMD_3,
        SEND_CMD_4,
        SEND_CMD_5,

        SET_ERASE_COUNT_CMD,                    -- Send Set Erase Count
        SET_ERASE_COUNT_CMD_2,           -- Send ACMD23
        );

subtype t_error_code is std_logic_vector(2 downto 0);
constant ec_NoError : t_error_code := "000";
constant ec_R1Error : t_error_code := "001";
constant ec_CRCError      : t_error_code := "010";
constant ec_WriteTimeout  : t_error_code := "010";
constant ec_DataRespError : t_error_code := "011";
constant ec_DataError     : t_error_code := "100";
constant ec_WPError : t_error_code := "101";
constant ec_SDError : t_error_code := "110";
constant ec_NoSDError     : t_error_code := "111";

subtype t_card_type is std_logic_vector(1 downto 0);
constant ct_None : t_card_type := "00";
constant ct_SDV1 : t_card_type := "01";
constant ct_SDV2 : t_card_type := "10";
constant ct_SDHC : t_card_type := "11";

constant R1_IDLE : integer := 0;
constant R1_ERASE_RESET : integer := 1;
constant R1_ILLEGALCOMMAND : integer := 2;
constant R1_COMMANDCRCERROR : integer := 3;
constant R1_ERASESEQUENCEERROR : integer := 4;
```

```
constant R1_ADDRESSERROR : integer := 5;
constant R1_PARAMETERERROR : integer := 6;
constant R1_ZERO : integer := 7;
constant OCR1_CCS : integer := 6;

signal state, new_state, return_state, new_return_state, sr_return_state,
new_sr_return_state : states := RST;
signal set_return_state, set_sr_return_state : boolean := false;

-- Output signals to SD Card
signal new_sclk : std_logic := '0';
signal sCs, new_cs : std_logic := '1';

-- Output signals to higher level
signal set_davail : boolean := false;
signal sDavail : std_logic := '0';
signal transfer_data_out, new_transfer_data_out : boolean := false;
signal card_type, new_card_type : t_card_type := ct_None;
signal error, new_error : std_logic := '0';
signal error_code, new_error_code : t_error_code := ec_NoError;
signal new_busy : std_logic := '1';
signal sDin_taken, new_din_taken : std_logic := '0';

-- Shift registers
signal cmd_out, new_cmd_out : std_logic_vector(39 downto 0) := (others=>'1');
signal set_cmd_out : boolean := false;
signal data_in, new_data_in : std_logic_vector(7 downto 0);
signal new_crc7, crc7 : std_logic_vector(6 downto 0);
signal new_in_crc16, in_crc16 : std_logic_vector(15 downto 0);
signal new_out_crc16, out_crc16 : std_logic_vector(15 downto 0);
signal new_crcLow, crcLow : std_logic_vector(7 downto 0);
signal data_out, new_data_out : std_logic_vector(7 downto 0) := x"00";

signal address, new_address : std_logic_vector(31 downto 0);
signal wr_erase_count, new_wr_erase_count : std_logic_vector(7 downto 0);
signal set_address : boolean := false;
signal byte_counter, new_byte_counter : integer range 0 to 512 := 0;
signal set_byte_counter : boolean := false;
signal bit_counter, new_bit_counter : integer range 0 to 160 := 0;
signal slow_clock, new_slow_clock : boolean := true;
signal clock_divider, new_clock_divider : integer range 0 to slowClockDivider := 0;
signal multiple, new_multiple : boolean := false;
signal skipFirstR1Byte, new_skipFirstR1Byte : boolean := false;
signal din_latch : boolean := false;
signal last_din_valid : std_logic := '0';

begin
       -- This process updates all the state variables from the values calculated
```

```
-- by the calcStateVariables process
updateStateVariables: process(clk)
begin
	if rising_edge(clk) then
		if (reset='1') then
			state <= RST;
			return_state <= RST;
			sr_return_state <= RST;
			cmd_out <= (others=>'1');
			data_in <= (others=>'0');
			dout <= (others=>'0');
			address <= (others=>'0');
			data_out <= (others=>'1');
			card_type <= ct_None;
			byte_counter <= 0;
			bit_counter <= 0;
			crc7 <= (others => '0');
			in_crc16 <= (others => '0');
			out_crc16 <= (others => '0');
			crcLow <= (others => '0');
			error <= '1';
			error_code <= ec_NoSDError;
			sdAvail <= '0';
			error <= '0';
			slow_clock <= true;
			clock_divider <= 0;
			transfer_data_out <= false;
			sCs <= '1';
			sDin_taken <= '0';
			wr_erase_count <= "00000001";
			-- SD outputs
			sclk <= '0';
			cs <= '1';
			mosi <= '1';
			-- Interface outputs
			sd_type <= "00";
			sd_busy <= '1';
			sd_error <= '1';
			sd_error_code <= ec_NoSDError;
			dout <= "00000000";
			dout_avail <= '0';
			din_taken <= '0';
			multiple <= false;
			skipFirstR1Byte <= false;
		else
			-- State variables
			state <= new_state;
```

```
                        if (set_return_state) then return_state <=
new_return_state; end if;
                        if (set_sr_return_state) then sr_return_state <=
new_sr_return_state; end if;
                        if (set_cmd_out) then cmd_out <= new_cmd_out; end if;
                        data_in <= new_data_in;
                        if (set_address) then address <= new_address; end if;
                        data_out <= new_data_out;
                        if (set_byte_counter) then byte_counter <=
new_byte_counter; end if;
                        bit_counter <= new_bit_counter;
                        error <= new_error;
                        error_code <= new_error_code;
                        card_type <= new_card_type;
                        slow_clock <= new_slow_clock;
                        clock_divider <= new_clock_divider;
                        crc7 <= new_crc7;
                        in_crc16 <= new_in_crc16;
                        out_crc16 <= new_out_crc16;
                        crcLow <= new_crcLow;
                        transfer_data_out <= new_transfer_data_out;
                        sCs <= new_cs;
                        -- SD outputs
                        sclk <= new_sclk;
                        cs <= new_cs;
                        mosi <= new_data_out(7);
                        wr_erase_count <= new_wr_erase_count;
                        -- Interface outputs
                        sd_type <= new_card_type;
                        sd_busy <= new_busy;
                        sd_error <= new_error;
                        sd_error_code <= new_error_code;
                        if set_davail then -- NB can't do this at the same cycle
as we set data_in
                              sDavail <= '1';
                              dout <= data_in;
                              dout_avail <= '1';
                        elsif sDavail='1' and dout_taken='1' then
                              sDavail <= '0';
                              dout_avail <= '0';
                        end if;
                        multiple <= new_multiple;
                        skipFirstR1Byte <= new_skipFirstR1Byte;

                        -- This latches the din_valid and generates din_latch and
din_taken
                        if din_valid='0' or (wr='0' and wr_multiple='0') then
```

```
                                -- Reset din_latch when din_valid is false, or no
write in progress
                                sDin_taken <= '0';
                                din_taken <= '0';
                                din_latch <= false;
                          elsif din_valid='1' and last_din_valid='0' then
                                -- Set din_latch on rising edge of din_valid
                                sDin_taken <= '0';
                                din_taken <= '0';
                                din_latch <= true;
                          elsif din_latch and new_din_taken='1' then
                                -- Reset din_latch when din_taken rises
                                sDin_taken <= '1';
                                din_taken <= '1';
                                din_latch <= false;
                          end if;
                          last_din_valid <= din_valid;
                    end if;
              end if;
    end process;

       -- This process calculates all of the state variables
       -- It should not generate any latches
       -- Some values are initialised to a fixed value, and overridden later (new_X
<= '0')
       -- Some values are initialised to their current values (new_X <= X)
       -- Some values are initialised to Don't Care (new_X <= '-')
       -- Updating of the latter values is under control of the set_X signal
       calcStateVariables: process(miso,rd,rd_multiple,wr,wr_multiple,
              state,bit_counter,card_type,byte_counter,data_in,data_out,
              address,addr,dout_taken,error,cmd_out,return_state,clock_divider,
              error_code,crc7,in_crc16,out_crc16,slow_clock,card_present,

       card_write_prot,SDin_Taken,sCS,transfer_data_out,din_valid,din,din_latch,
              crcLow,sDavail,sr_return_state,multiple,skipFirstR1Byte)
       constant WriteTimeoutCount : integer := clockRate/18000 * WRITE_TIMEOUT;
       begin
              assert(WriteTimeoutCount > 0) report "WriteTimeoutCount is 0" severity
failure ;
              new_state <= state;
              new_return_state <= RST;
              set_return_state <= false;
              new_sr_return_state <= RST;
              set_sr_return_state <= false;
              new_bit_counter <= bit_counter;
              new_card_type <= card_type;
              new_cmd_out <= (others=>'-');
              set_cmd_out <= false;
```

```
		new_byte_counter <= byte_counter;
		set_byte_counter <= false;
		new_data_in <= data_in;
		set_davail <= false;
		new_din_taken <= sDin_taken;
		new_data_out <= data_out;
		new_address <= (others=>'-');
		set_address <= false;
		new_sclk <= '0';
		new_cs <= sCs;
		new_error <= error;
		new_error_code <= error_code;
		new_busy <= '1';
		new_crc7 <= crc7;
		new_in_crc16 <= in_crc16;
		new_out_crc16 <= out_crc16;
		new_crcLow <= crcLow;
		new_slow_clock <= slow_clock;
		new_clock_divider <= clock_divider;
		new_transfer_data_out <= transfer_data_out;
		new_multiple <= multiple;
		new_skipFirstR1Byte <= skipFirstR1Byte;
		new_wr_erase_count <= wr_erase_count;

		case state is

		when RST =>
			-- Reset, including error codes
			new_error_code <= ec_NoSDError;
			new_error <= '1';
			new_state <= RST2;

		when RST2 =>
			-- Reset, retaining error codes
			new_card_type <= ct_None;
			new_cs <= '1';
			new_slow_clock <= true;
			new_clock_divider <= slowClockDivider;
			new_byte_counter <= 20; set_byte_counter <= true;
			new_data_out <= "11111111";
			new_transfer_data_out <= false;
			new_sr_return_state <= INIT; set_sr_return_state <= true;
			if card_present='1' then
			-- Wait for card present indication before attempting
initialisation
				new_state <= SEND_RCV;
			end if;
```

```
		when INIT =>
			if byte_counter=0 then
				new_state <= CMD0;
			else
				new_state <= SEND_RCV;
			end if;

		when CMD0 =>
			-- Send CMD0
			new_cs <= '0';
			new_address <= (others=>'0'); set_address <= true;
			new_cmd_out <= x"4000000000"; set_cmd_out <= true;
			new_return_state <= CMD8; set_return_state <= true;
			new_state <= SEND_CMD;

		when CMD8 =>
			-- Check CMD0 response and send CMD8 or Error
			if data_in="00000001" then
				new_cmd_out <= x"48000001AA"; set_cmd_out <= true; --
Voltage is 1, Check pattern is AA
				new_return_state <= CMD8R1; set_return_state <= true;
				new_state <= SEND_CMD;
			else
				new_card_type <= ct_None;
				new_error <= '1';
				new_error_code <= ec_R1Error;
				new_state <= RST2;
			end if;

		when CMD8R1 =>
			-- Check R1 response to CMD8
			if data_in(R1_ILLEGALCOMMAND)='1' then -- Illegal command?
				new_card_type <= ct_SDV1; -- Yes, must be SD1
				new_state <= CMD55;
			else
				new_card_type <= ct_SDV2; -- No, could be SD2 (10) or SDHC
(11)
				new_sr_return_state <= CMD8B2; set_sr_return_state <=
true;
				new_state <= SEND_RCV;
			end if;

		when CMD8B2 =>
			-- Got first byte of CMD8 response
			new_sr_return_state <= CMD8B3; set_sr_return_state <= true;
			new_state <= SEND_RCV;

		when CMD8B3 =>
```

```
                -- Got second byte of CMD8 response
                new_sr_return_state <= CMD8B4; set_sr_return_state <= true;
                new_state <= SEND_RCV;

        when CMD8B4 =>
                -- Got third byte of CMD8 response
                -- Check operating voltage
                if data_in(3 downto 0) /= "0001" then
                        new_state <= RST;
                end if;
                -- Get byte 4 (check pattern)
                new_sr_return_state <= CMD8GOTB4; set_sr_return_state <= true;
                new_state <= SEND_RCV;

        when CMD8GOTB4 =>
                -- Got fourth byte of CMD8 response
                -- Check pattern
                if data_in = x"AA" then
                        new_state <= CMD55;
                else
                        new_state <= RST;
                end if;

        when CMD55 =>
                -- Send CMD55
                new_return_state <= CMD41; set_return_state <= true;
                new_cmd_out <= x"7700000000"; set_cmd_out <= true;
                new_state <= SEND_CMD;

        when CMD41 =>
                -- Send CMD41
                new_return_state <= POLL_CMD; set_return_state <= true;
                if card_type=ct_SDV1 then
                        new_cmd_out <= x"6900000000";
                else
                        new_cmd_out <= x"6940000000";
                end if;
                set_cmd_out <= true;
                new_state <= SEND_CMD;

        when POLL_CMD =>
                -- Poll until card ready, then send CMD58 or CMD59 depending on
type
                if (data_in(R1_IDLE) = '0') then -- In idle state?
                        if (card_type=ct_SDV1) then
                                new_state <= CMD59; -- SD1 ready now
                        else
                                new_state <= CMD58; -- SD2, SDHC determine
```

```
                        end if;
                else
                        new_state <= CMD55; -- Still in idle, repeat ACMD41
                end if;

        when CMD58 =>
                -- Send CMD58
                new_return_state <= CMD58R1; set_return_state <= true;
                new_cmd_out <= x"7A00000000"; set_cmd_out <= true;
                new_state <= SEND_CMD;

        when CMD58R1 =>
                -- Check R1 response to CMD58
                if data_in(R1_ILLEGALCOMMAND)='1' then
                        -- Illegal command - not an SD card
                        new_card_type <= ct_None;
                        new_error_code <= ec_SDError;
                        new_error <= '1';
                        new_state <= RST2;
                else
                        -- Go fetch byte 1
                        new_sr_return_state <= CMD58B2; set_sr_return_state <=
true;
                        new_state <= SEND_RCV;
                end if;

        when CMD58B2 =>
                -- Check CCS: 0=SD2 1=SDHC
                -- card_type already set to ct_SDV2 (10) in CMD8R1
                if (data_in(OCR1_CCS)='1') then -- OCR(30) = CCS
                        new_card_type <= ct_SDHC; -- SDHC
                end if;
                -- Go fetch byte 2
                new_sr_return_state <= CMD58B3; set_sr_return_state <= true;
                new_state <= SEND_RCV;

        when CMD58B3 =>
                -- Fetch byte 3
                new_sr_return_state <= CMD58B4; set_sr_return_state <= true;
                new_state <= SEND_RCV;

        when CMD58B4 =>
                -- Fetch byte 4
                new_sr_return_state <= CMD59; set_sr_return_state <= true;
                new_state <= SEND_RCV;

        when CMD59 =>
                -- Send CMD59
```

```
                        new_return_state <= CMD59R1; set_return_state <= true;
                        new_cmd_out <= x"7B00000001"; set_cmd_out <= true; -- Enable CRC
                        new_state <= SEND_CMD;

                when CMD59R1 =>
                        -- Check reply from CMD59
                        if data_in/="00000000" then
                                new_state <= RST;
                        end if;
                        -- Don't enter IDLE until Rd and Wr are down
                        if (rd='0') and (wr='0') and (rd_multiple='0') and
(wr_multiple='0') then
                                new_error_code <= ec_NoError;
                                new_error <= '0';
                                new_state <= IDLE;
                        end if;

                when IDLE =>
                        -- Generate 8 clocks when entering idle
                        new_slow_clock <= false;  -- Can run at full speed now
                        new_data_out <= "11111111";
                        new_bit_counter <= 7;
                        new_transfer_data_out <= false;
                        new_sr_return_state <= IDLE2; set_sr_return_state <= true;
                        new_state <= SEND_RCV;

                when IDLE2 =>
                        -- Sits in this state when idle
                        if card_present='0' then
                                -- Card gone!
                                new_state <= RST;
                        elsif data_in=x"00" then
                                -- Card still busy
                                new_state <= IDLE;
                        elsif rd='1' then
                                -- Initiate Read
                                new_cs <= '0';
                                new_error <= '0';
                                new_error_code <= ec_NoError;
                                new_address <= addr; set_address <= true;
                                new_multiple <= false;
                                new_state <= READ_BLOCK;
                        elsif rd_multiple='1' then
                                -- Initiate Read Multiple
                                new_cs <= '0';
                                new_error <= '0';
                                new_error_code <= ec_NoError;
                                new_address <= addr; set_address <= true;
```

```
                new_multiple <= true;
                new_state <= READ_MULTIPLE_BLOCK;
        elsif wr='1' or wr_multiple='1' then
                -- Initiate Write or Write Multiple
                if card_write_prot='0' then
                        new_cs <= '0';
                        new_error <= '0';
                        new_error_code <= ec_NoError;
                        new_address <= addr; set_address <= true;
                        if wr='1' then
                                new_multiple <= false;
                                new_wr_erase_count <= "00000001";
                        else
                                new_multiple <= true;
                                new_wr_erase_count <= erase_count;
                        end if;
                        new_state <= SET_ERASE_COUNT_CMD;
                else
                        new_error <= '1';
                        new_error_code <= ec_WPError;
                end if;
        else
                -- No command
                new_cs <= '1';
                new_busy <= '0';
        end if;

when READ_BLOCK =>
        -- Basic Read command
        if card_type=ct_SDHC then
                -- SDHC: Use block address
                new_cmd_out <= x"51" & address(31 downto 0);
        else
                -- SDV1,2: Use byte address
                new_cmd_out <= x"51" & address(22 downto 0) & "000000000";
        end if;
        set_cmd_out <= true;
        new_return_state <= READ_BLOCK_R1; set_return_state <= true;
        new_state <= SEND_CMD;

when READ_MULTIPLE_BLOCK =>
        -- Read Multiple command
        if card_type=ct_SDHC then
                -- SDHC: Use block address
                new_cmd_out <= x"52" & address(31 downto 0);
        else
                -- SDV1,2: Use byte address
                new_cmd_out <= x"52" & address(22 downto 0) & "000000000";
```

```
                        end if;
                        set_cmd_out <= true;
                        new_return_state <= READ_BLOCK_R1; set_return_state <= true;
                        new_state <= SEND_CMD;

                when READ_BLOCK_R1 =>
                        -- Get R1 response to Read or Read Multiple command
                        if data_in/="00000000" then -- Some error
                                new_error <= '1';
                                new_error_code <= ec_R1Error;
                                new_state <= READ_BLOCK_FINISH;
                        else
                                new_sr_return_state <= READ_BLOCK_WAIT_CHECK;
set_sr_return_state <= true;
                                new_state <= SEND_RCV;
                        end if;

                when READ_BLOCK_WAIT_CHECK =>
                        -- Wait for Read token, or Error token
                        new_in_crc16 <= (others=>'0');
                        if rd='0' and rd_multiple='0' then
                                -- Abort transfer
                                new_state <= READ_BLOCK_FINISH; -- And then to IDLE
                        elsif (data_in="11111110") then
                                new_transfer_data_out <= true;
                                new_byte_counter <= 512; set_byte_counter <= true;
                                new_sr_return_state <= READ_BLOCK_DATA;
set_sr_return_state <= true; -- Wait for dout_taken to drop
                                new_state <= SEND_RCV;
                        elsif (data_in(7 downto 4)="0000") then
                                -- Check for error token 0000XXXX
                                -- Flag error and wait for RD to drop
                                new_error <= '1';
                                new_error_code <= ec_DataError;
                                new_state <= READ_BLOCK_FINISH;
                        else
                                new_state <= SEND_RCV;
                        end if;

                when READ_BLOCK_DATA =>
                        -- Read a byte of data from the card
                        if rd='0' and rd_multiple='0' then
                                -- Abort transfer
                                new_state <= READ_BLOCK_SKIP; -- And then to IDLE
                        else
                                if byte_counter=0 then
                                        new_transfer_data_out <= false;
                                        new_sr_return_state <= READ_BLOCK_CRC;
```

```
                              set_sr_return_state <= true;
                        end if;
                        new_state <= SEND_RCV;
                  end if;

            when READ_BLOCK_SKIP =>
                  -- Skip all remaining bytes without transferring them
                  new_transfer_data_out <= false;
                  if multiple then
                        -- Special stop mechanism for Read Multiple
                        new_state <= READ_MULTIPLE_BLOCK_STOP;
                  elsif (byte_counter=0) then
                        -- After last byte, read the first CRC byte
                        new_sr_return_state <= READ_BLOCK_CRC; set_sr_return_state
<= true;
                        new_state <= SEND_RCV;
                  else
                        -- Keep skipping bytes
                        new_sr_return_state <= READ_BLOCK_SKIP;
set_sr_return_state <= true;
                        new_state <= SEND_RCV;
                  end if;

            when READ_BLOCK_CRC =>
                  -- Read second CRC byte
                  new_sr_return_state <= READ_BLOCK_CHECK_CRC; set_sr_return_state
<= true;
                  new_state <= SEND_RCV;

            when READ_BLOCK_CHECK_CRC =>
                  -- After reading all the data and the two CRC bytes, the result
should be zero
                  if in_crc16/="0000000000000000" then
                        new_error <= '1';
                        new_error_code <= ec_CRCError;
                        new_state <= READ_BLOCK_FINISH;
                  elsif multiple and rd_multiple='1' then
                        -- Start looking for a further data block
                        new_sr_return_state <= READ_BLOCK_WAIT_CHECK;
set_sr_return_state <= true;
                        new_state <= SEND_RCV;
                  else
                        -- Transfer complete
                        new_state <= READ_BLOCK_FINISH;
                  end if;

            when READ_BLOCK_FINISH =>
                  new_transfer_data_out <= false;
```

```
                -- Wait for RD to fall after last byte has been transferred
                if (rd='0') and (rd_multiple='0') then
                        if multiple then
                                new_state <= READ_MULTIPLE_BLOCK_STOP;
                        else
                                new_state <= IDLE;
                        end if;
                end if;

        when READ_MULTIPLE_BLOCK_STOP =>
                -- Send CMD12
                new_skipFirstR1Byte <= true;
                new_return_state <= READ_MULTIPLE_BLOCK_STOP_2; set_return_state
<= true;
                new_cmd_out <= x"4C00000000"; set_cmd_out <= true;
                new_state <= SEND_CMD;

        when READ_MULTIPLE_BLOCK_STOP_2 =>
                -- Check R1 and wait for not-busy when we get to IDLE
                if data_in/="00000000" then
                        new_state <= RST;
                else
                        if rd_multiple='0' then
                                new_state <= IDLE;
                        end if;
                end if;

        when SET_ERASE_COUNT_CMD =>
                -- Send CMD55
                new_return_state <= SET_ERASE_COUNT_CMD_2; set_return_state <=
true;
                new_cmd_out <= x"7700000000"; set_cmd_out <= true;
                new_state <= SEND_CMD;

        when SET_ERASE_COUNT_CMD_2 =>
                -- Send ACMD23
                new_cmd_out <= x"57000000" & wr_erase_count;
                if wr='1' then
                        new_return_state <= WRITE_BLOCK_CMD;
                else
                        new_return_state <= WRITE_MULTIPLE_BLOCK_CMD;
                end if;
                set_cmd_out <= true;
                set_return_state <= true;
                new_state <= SEND_CMD;
```

```
            when SEND_RCV =>
                  -- Send the byte in data_out while simultaneously receiving one
into data_in
                  -- ** Must enter with bit_counter = 7 **
                  -- Update CRC7 and CRC16 from output stream
                  -- Update CRC16 from input stream
                  -- Decrement byte_counter
                  -- Leave data_out as 11111111
                  -- Leave bit_counter as 7 for next time

                  -- When we enter SPI Clock should be low, we set the output data,
wait half a cycle, raise
                  -- the clock, latch the input data, then wait a further half
cycle before dropping the clock
                  -- The output data (MOSI) follows data_out(7)

                  -- Clock is low, output data is set
                  if slow_clock=false or clock_divider=0 then
                        new_clock_divider <= slowClockDivider;
                        new_sclk <= '1';
                        -- Update output CRCs
                        new_crc7 <= crc7(5 downto 3) & (crc7(2) xor crc7(6) xor
data_out(7)) & crc7(1 downto 0) & (crc7(6) xor data_out(7));
                        new_out_crc16 <= out_crc16(14 downto 12) & (data_out(7)
xor out_crc16(15) xor out_crc16(11)) & out_crc16(10 downto 5) &
                        (data_out(7) xor out_crc16(15) xor out_crc16(4)) &
out_crc16(3 downto 0) & (data_out(7) xor out_crc16(15));
                        -- Update input data
                        new_data_in <= data_in(6 downto 0) & miso;
                        -- Update input CRC
                        new_in_crc16 <= in_crc16(14 downto 12) & (miso xor
in_crc16(15) xor in_crc16(11)) & in_crc16(10 downto 5) &
                              (miso xor in_crc16(15) xor in_crc16(4)) & in_crc16(3
downto 0) & (miso xor in_crc16(15));
                        new_state <= SEND_RCV_CLK1;
                  else
                        new_clock_divider <= clock_divider - 1;
                  end if;

            when SEND_RCV_CLK1 =>
                  if slow_clock=false or clock_divider=0 then
                        new_clock_divider <= slowClockDivider;
                        if (bit_counter = 0) then
                              -- Reception handling - if DAvail and DTaken are
down, transfer new byte into output register and raise DAvail
                              if transfer_data_out then
                                    if (rd='1' or rd_multiple='1') then
```

```
                                        if sDavail='0' and dout_taken='0' then
                                                -- If we're ok to transfer data,
then do it
                                                -- otherwise wait here until
dout_taken rises
                                                set_davail <= true;
                                                new_byte_counter <= byte_counter
- 1; set_byte_counter <= true;
                                                -- Next byte
                                                new_bit_counter <= 7;
                                                if byte_counter=1 then
                                                        new_transfer_data_out <=
false;
                                                        new_sr_return_state <=
READ_BLOCK_CRC;
                                                        set_sr_return_state <=
true;
                                                end if;
                                                new_state <= SEND_RCV;
                                        end if;
                                else
                                        -- Abort transfer
                                        new_byte_counter <= byte_counter - 1;
set_byte_counter <= true;
                                        -- Next byte
                                        new_bit_counter <= 7;
                                        if byte_counter=1 then
                                                new_transfer_data_out <= false;
                                                new_sr_return_state <=
READ_BLOCK_CRC;
                                                set_sr_return_state <= true;
                                        end if;
                                        new_state <= SEND_RCV;
                                end if;
                        else
                                new_bit_counter <= 7;
                                new_state <= sr_return_state;
                                new_byte_counter <= byte_counter - 1;
set_byte_counter <= true;
                        end if;
                else
                        new_bit_counter <= bit_counter - 1;
                        new_data_out <= data_out(6 downto 0) & '1';
                        new_state <= SEND_RCV;
                end if;
        else
                new_sclk <= '1';
                new_clock_divider <= clock_divider - 1;
```

```
                end if;

            when SEND_CMD =>
                -- Send FF byte first
                new_bit_counter <= 7;
                new_data_out <= "11111111";
                new_sr_return_state <= SEND_CMD_1; set_sr_return_state <= true;
                new_state <= SEND_RCV;

            when SEND_CMD_1 =>
                -- Initialise CRC and byte counter
                new_crc7 <= "0000000";
                new_byte_counter <= 5; set_byte_counter <= true; -- 5 bytes are
CC NN NN NN NN
                new_state <= SEND_CMD_2;

            when SEND_CMD_2 =>
                -- Send one byte of the command and parameter
                if byte_counter=0 then
                    new_state <= SEND_CMD_3;
                else
                    new_data_out <= cmd_out(39 downto 32);
                    new_cmd_out <= cmd_out(31 downto 0) & x"FF"; set_cmd_out
<= true;
                    new_sr_return_state <= SEND_CMD_2; set_sr_return_state <=
true;
                    new_state <= SEND_RCV;
                end if;

            when SEND_CMD_3 =>
                -- Send the CRC
                new_data_out <= crc7 & '1';
                new_sr_return_state <= SEND_CMD_4; set_sr_return_state <= true;
                new_state <= SEND_RCV;

            when SEND_CMD_4 =>
                -- Receive the first byte, maybe R1
                new_byte_counter <= R1_TIMEOUT; set_byte_counter <= true;
                new_sr_return_state <= SEND_CMD_5; set_sr_return_state <= true;
                new_state <= SEND_RCV;

            when SEND_CMD_5 =>
                -- Check for R1 response, receive another byte if not
                if skipFirstR1Byte then
                    -- If doing a CMD12 then skip a byte before looking for R1
                    new_skipFirstR1Byte <= false;
                    new_state <= SEND_RCV;
                elsif data_in(R1_ZERO)='0' then
```

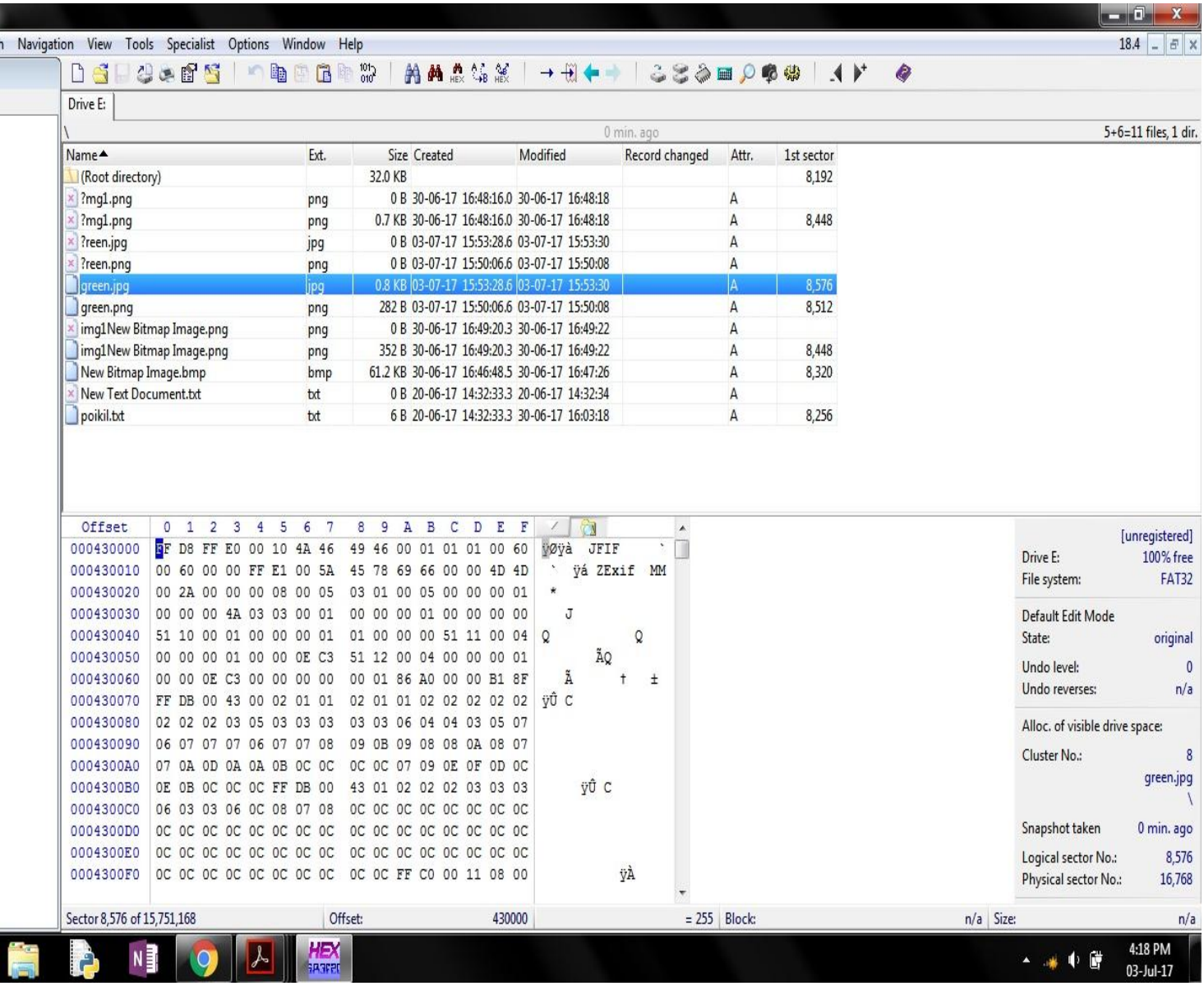
```
                        new_state <= return_state;
                    else
                        if byte_counter=0 then
--                              new_state <= RST2;
                                new_card_type <= ct_None;
                                new_error <= '1';
                                new_error_code <= ec_NoSDError;
                        else
                                new_state <= SEND_RCV; -- Will come back to
SEND_CMD_5
                        end if;
                    end if;
            end case;
        end process calcStateVariables;

        -- This calculates a debug output to determine the FSM state
        calcDebugOutputs: block
        begin
                with state select sd_fsm <=
                        x"00" when RST,
                        x"00" when RST2,
                        x"01" when INIT,
                        x"02" when CMD0,
                        x"03" when CMD8,
                        x"04" when CMD8R1,
                        x"04" when CMD8B2,
                        x"04" when CMD8B3,
                        x"04" when CMD8B4,
                        x"04" when CMD8GOTB4,
                        x"05" when CMD55,
                        x"06" when CMD41,
                        x"07" when POLL_CMD,
                        x"08" when CMD58,
                        x"08" when CMD58R1,
                        x"08" when CMD58B2,
                        x"08" when CMD58B3,
                        x"08" when CMD58B4,
                        x"09" when CMD59,
                        x"0A" when CMD59R1,
                        x"10" when IDLE,
                        x"11" when IDLE2,
                        x"20" when READ_BLOCK,
                        x"20" when READ_MULTIPLE_BLOCK,
                        x"21" when READ_BLOCK_R1,
                        x"22" when READ_BLOCK_WAIT_CHECK,
                        x"23" when READ_BLOCK_DATA,
                        x"24" when READ_BLOCK_SKIP,
                        x"25" when READ_BLOCK_CRC,
```

```
                    x"26" when READ_BLOCK_CHECK_CRC,
                    x"27" when READ_BLOCK_FINISH,
                    x"28" when READ_MULTIPLE_BLOCK_STOP,
                    x"29" when READ_MULTIPLE_BLOCK_STOP_2,
                    x"30" when SEND_RCV,
                    x"31" when SEND_RCV_CLK1,
                    x"32" when SEND_CMD,
                    x"33" when SEND_CMD_1,
                    x"34" when SEND_CMD_2,
                    x"35" when SEND_CMD_3,
                    x"36" when SEND_CMD_4,
                    x"37" when SEND_CMD_5,
                    x"40" when SET_ERASE_COUNT_CMD,
                    x"41" when SET_ERASE_COUNT_CMD_2,
                    x"42" when WRITE_BLOCK_CMD,
                    x"43" when WRITE_MULTIPLE_BLOCK_CMD,
                    x"44" when WRITE_BLOCK_INIT,
                    x"45" when WRITE_BLOCK_DATA,
                    x"46" when WRITE_BLOCK_DATA_TOKEN,
                    x"47" when START_WRITE_BLOCK_DATA,
                    x"48" when WRITE_BLOCK_SEND_CRC2,
                    x"49" when WRITE_BLOCK_GET_RESPONSE,
                    x"4A" when WRITE_BLOCK_CHECK_RESPONSE,
                    x"4B" when WRITE_BLOCK_WAIT,
                    x"4C" when WRITE_BLOCK_ABORT,
                    x"4D" when WRITE_BLOCK_TERMINATE,
                    x"4E" when WRITE_BLOCK_FINISH
                    ;
      end block calcDebugOutputs;
end rtl;
```

OUTPUT:

An image was stored in PNG format in the SD card. Using the winhex program the sector of storage of the image was found out. In SPI mode of communication the physical sector of the SD card is relevant.. This value is converted into 32 bit HEX value and coded into the FPGA. The screenshot of the hex editor has been included. We can see that the image named green.png is saved in the SD card. The sector number is 16,768 which is converted into hex as 0x4180 and hard coded into the program. The bit stream of the image is read from this sector of the SD card and stored into a RAM of the FPGA. The reading progress can be viewed and debugged using the 7-segment displays. The 7-seg display displays the states of the SD card and also the CPU. The error codes are displayed in the green LEDs. These show whether the SD card is inserted and the type of SD card. Once all the data has been stored in the RAM, it can



be accessed for further manipulation. For instance it can be read and the corresponding image can be displayed in a monitor through a VGA port. In our project the bit stream of the image has been displayed in the 7-seg displays. The first four bytes of data are displayed in 7 seg as 0xFFD8FFE0, which is confirmed to be correct from the hex editor.