```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3   USE ieee.std_logic_arith.all;
4
5   ENTITY SPI_slave_FPGA IS
6     GENERIC(
7       cpol    : STD_LOGIC := '0';  --spi clock polarity mode
8       cpha    : STD_LOGIC := '1';  --spi clock phase mode
9       d_width : INTEGER := 8);     --data width in bits
10    PORT(
11      sclk         : IN     STD_LOGIC;  --spi clk from master
12      reset_n      : IN     STD_LOGIC;  --active low reset
13      ss_n         : IN     STD_LOGIC;  --active low slave select
14      mosi         : IN     STD_LOGIC;  --master out, slave in
15      rx_req       : IN     STD_LOGIC;  --'1' while busy = '0' moves data to the rx_data
    output
16      st_load_en   : IN     STD_LOGIC;  --asynchronous load enable
17      st_load_trdy : IN     STD_LOGIC;  --asynchronous trdy load input
18      st_load_rrdy : IN     STD_LOGIC;  --asynchronous rrdy load input
19      st_load_roe  : IN     STD_LOGIC;  --asynchronous roe load input
20      tx_load_en   : IN     STD_LOGIC;  --asynchronous transmit buffer load enable
21      tx_load_data : IN     STD_LOGIC_VECTOR(d_width-1 DOWNTO 0);  --asynchronous tx data
    to load
22      trdy         : BUFFER STD_LOGIC := '0';  --transmit ready bit
23      rrdy         : BUFFER STD_LOGIC := '0';  --receive ready bit
24      roe          : BUFFER STD_LOGIC := '0';  --receive overrun error bit
25      rx_data      : OUT    STD_LOGIC_VECTOR(d_width-1 DOWNTO 0) := (OTHERS => '0');
    --receive register output to logic
26      busy         : OUT    STD_LOGIC := '0';  --busy signal to logic ('1' during
    transaction)
27      miso         : OUT    STD_LOGIC := 'Z'); --master in, slave out
28  END SPI_slave_FPGA;
29
30  ARCHITECTURE logic OF SPI_slave_FPGA IS
31    SIGNAL mode    : STD_LOGIC;  --groups modes by clock polarity relation to data
32    SIGNAL clk     : STD_LOGIC;  --clock
33    SIGNAL bit_cnt : STD_LOGIC_VECTOR(d_width+8 DOWNTO 0);  --'1' for active transaction bit
34    SIGNAL wr_add  : STD_LOGIC;  --address of register to write ('0' = receive, '1' =
    status)
35    SIGNAL rd_add  : STD_LOGIC;  --address of register to read ('0' = transmit, '1' =
    status)
36    SIGNAL rx_buf  : STD_LOGIC_VECTOR(d_width-1 DOWNTO 0) := (OTHERS => '0');  --receiver
    buffer
37    SIGNAL tx_buf  : STD_LOGIC_VECTOR(d_width-1 DOWNTO 0) := (OTHERS => '0');  --transmit
    buffer
38  BEGIN
39    busy <= NOT ss_n;  --high during transactions
40
41    --adjust clock so writes are on rising edge and reads on falling edge
42    mode <= cpol XOR cpha;  --'1' for modes that write on rising edge
43    WITH mode SELECT
44      clk <= sclk WHEN '1',
45             NOT sclk WHEN OTHERS;
46
47    --keep track of miso/mosi bit counts for data alignmnet
48    PROCESS(ss_n, clk)
49    BEGIN
50      IF(ss_n = '1' OR reset_n = '0') THEN                          --this slave is not
    selected or being reset
51        bit_cnt <= (conv_integer(NOT cpha) => '1', OTHERS => '0'); --reset miso/mosi bit
    count
52      ELSE                                                        --this slave is selected
53        IF(rising_edge(clk)) THEN                                 --new bit on miso/mosi
54          bit_cnt <= bit_cnt(d_width+8-1 DOWNTO 0) & '0';         --shift active bit
    indicator
55        END IF;
```

```
56        END IF;
57      END PROCESS;
58
59      PROCESS(ss_n, clk, st_load_en, tx_load_en, rx_req)
60      BEGIN
61
62        --write address register ('0' for receive, '1' for status)
63        IF(bit_cnt(1) = '1' AND falling_edge(clk)) THEN
64          wr_add <= mosi;
65        END IF;
66
67        --read address register ('0' for transmit, '1' for status)
68        IF(bit_cnt(2) = '1' AND falling_edge(clk)) THEN
69          rd_add <= mosi;
70        END IF;
71
72        --trdy register
73        IF((ss_n = '1' AND st_load_en = '1' AND st_load_trdy = '0') OR reset_n = '0') THEN
74          trdy <= '0';    --cleared by user logic or reset
75        ELSIF(ss_n = '1' AND ((st_load_en = '1' AND st_load_trdy = '1') OR tx_load_en = '1'))
       THEN
76          trdy <= '1';    --set when tx buffer written or set by user
      logic
77        ELSIF(wr_add = '1' AND bit_cnt(9) = '1' AND falling_edge(clk)) THEN
78          trdy <= mosi;  --new value written over spi bus
79        ELSIF(rd_add = '0' AND bit_cnt(d_width+8) = '1' AND falling_edge(clk)) THEN
80          trdy <= '0';    --clear when transmit buffer read
81        END IF;
82
83        --rrdy register
84        IF((ss_n = '1' AND ((st_load_en = '1' AND st_load_rrdy = '0') OR rx_req = '1')) OR
      reset_n = '0') THEN
85          rrdy <= '0';    --cleared by user logic or rx_data has been requested or reset
86        ELSIF(ss_n = '1' AND st_load_en = '1' AND st_load_rrdy = '1') THEN
87          rrdy <= '1';    --set when set by user logic
88        ELSIF(wr_add = '1' AND bit_cnt(10) = '1' AND falling_edge(clk)) THEN
89          rrdy <= mosi;  --new value written over spi bus
90        ELSIF(wr_add = '0' AND bit_cnt(d_width+8) = '1' AND falling_edge(clk)) THEN
91          rrdy <= '1';    --set when new data received
92        END IF;
93
94        --roe register
95        IF((ss_n = '1' AND st_load_en = '1' AND st_load_roe = '0') OR reset_n = '0') THEN
96          roe <= '0';    --cleared by user logic or reset
97        ELSIF(ss_n = '1' AND st_load_en = '1' AND st_load_roe = '1') THEN
98          roe <= '1';    --set by user logic
99        ELSIF(rrdy = '1' AND wr_add = '0' AND bit_cnt(d_width+8) = '1' AND falling_edge(clk))
       THEN
100         roe <= '1';    --set by actual overrun
101       ELSIF(wr_add = '1' AND bit_cnt(11) = '1' AND falling_edge(clk)) THEN
102         roe <= mosi;  --new value written by spi bus
103       END IF;
104
105       --receive registers
106       --write to the receive register from master
107       IF(reset_n = '0') THEN
108         rx_buf <= (OTHERS => '0');
109       ELSE
110         FOR i IN 0 TO d_width-1 LOOP
111           IF(wr_add = '0' AND bit_cnt(i+9) = '1' AND falling_edge(clk)) THEN
112             rx_buf(d_width-1-i) <= mosi;
113           END IF;
114         END LOOP;
115       END IF;
116       --fulfill user logic request for receive data
117       IF(reset_n = '0') THEN
```

```
118         rx_data <= (OTHERS => '0');
119       ELSIF(ss_n = '1' AND rx_req = '1') THEN
120         rx_data <= rx_buf;
121       END IF;
122
123       --transmit registers
124       IF(reset_n = '0') THEN
125         tx_buf <= (OTHERS => '0');
126       ELSIF(ss_n = '1' AND tx_load_en = '1') THEN  --load transmit register from user logic
127         tx_buf <= tx_load_data;
128       ELSIF(rd_add = '0' AND bit_cnt(7 DOWNTO 0) = "00000000" AND bit_cnt(d_width+8) = '0'
    AND rising_edge(clk)) THEN
129         tx_buf(d_width-1 DOWNTO 0) <= tx_buf(d_width-2 DOWNTO 0) & tx_buf(d_width-1);
    --shift through tx data
130       END IF;
131
132       --miso output register
133       IF(ss_n = '1' OR reset_n = '0') THEN           --no transaction occuring or reset
134         miso <= 'Z';
135       ELSIF(rd_add = '1' AND rising_edge(clk)) THEN  --write status register to master
136         CASE bit_cnt(10 DOWNTO 8) IS
137           WHEN "001" => miso <= trdy;
138           WHEN "010" => miso <= rrdy;
139           WHEN "100" => miso <= roe;
140           WHEN OTHERS => NULL;
141         END CASE;
142       ELSIF(rd_add = '0' AND bit_cnt(7 DOWNTO 0) = "00000000" AND bit_cnt(d_width+8) = '0'
    AND rising_edge(clk)) THEN
143         miso <= tx_buf(d_width-1);                   --send transmit register data to master
144       END IF;
145
146     END PROCESS;
147   END logic;
148
```