Serial Peripheral Interface (SPI)

Last updated 9/28/26

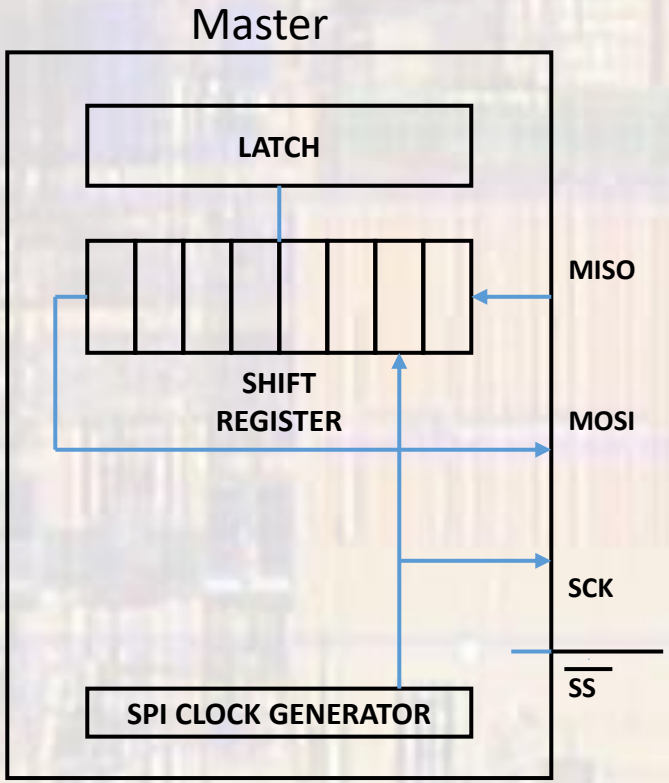
SPI

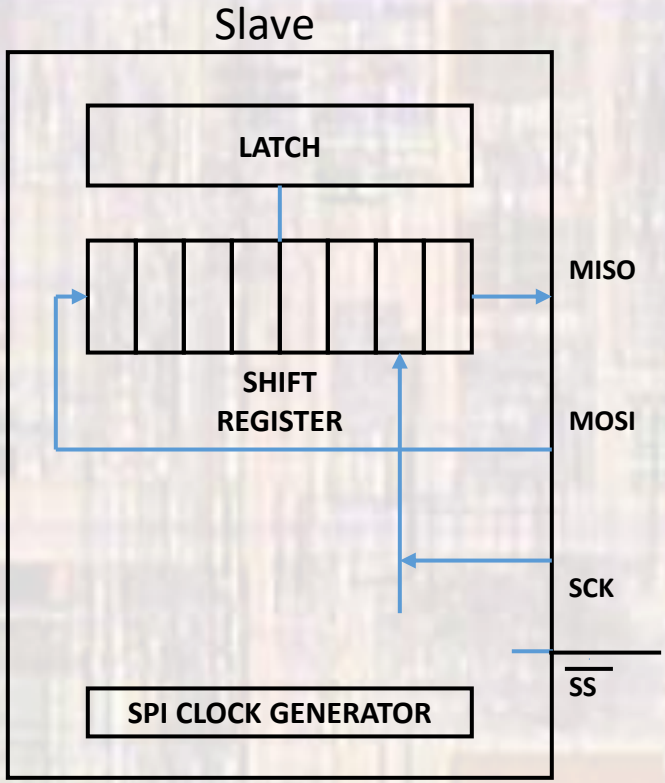
- Overview
  - N bit synchronous shift register used to communicate externally
  - Most often used to communicate with peripherals
    - displays, sensors, converters
  - Can be used for inter-processor communication
  - Two modes of operation
    - Master – responsible for providing the clock
    - Slave – receives clock from the master

© tj

SPI

- Overview





MISO – Master:IN or Slave:OUT

MOSI – Master:OUT or Slave:IN

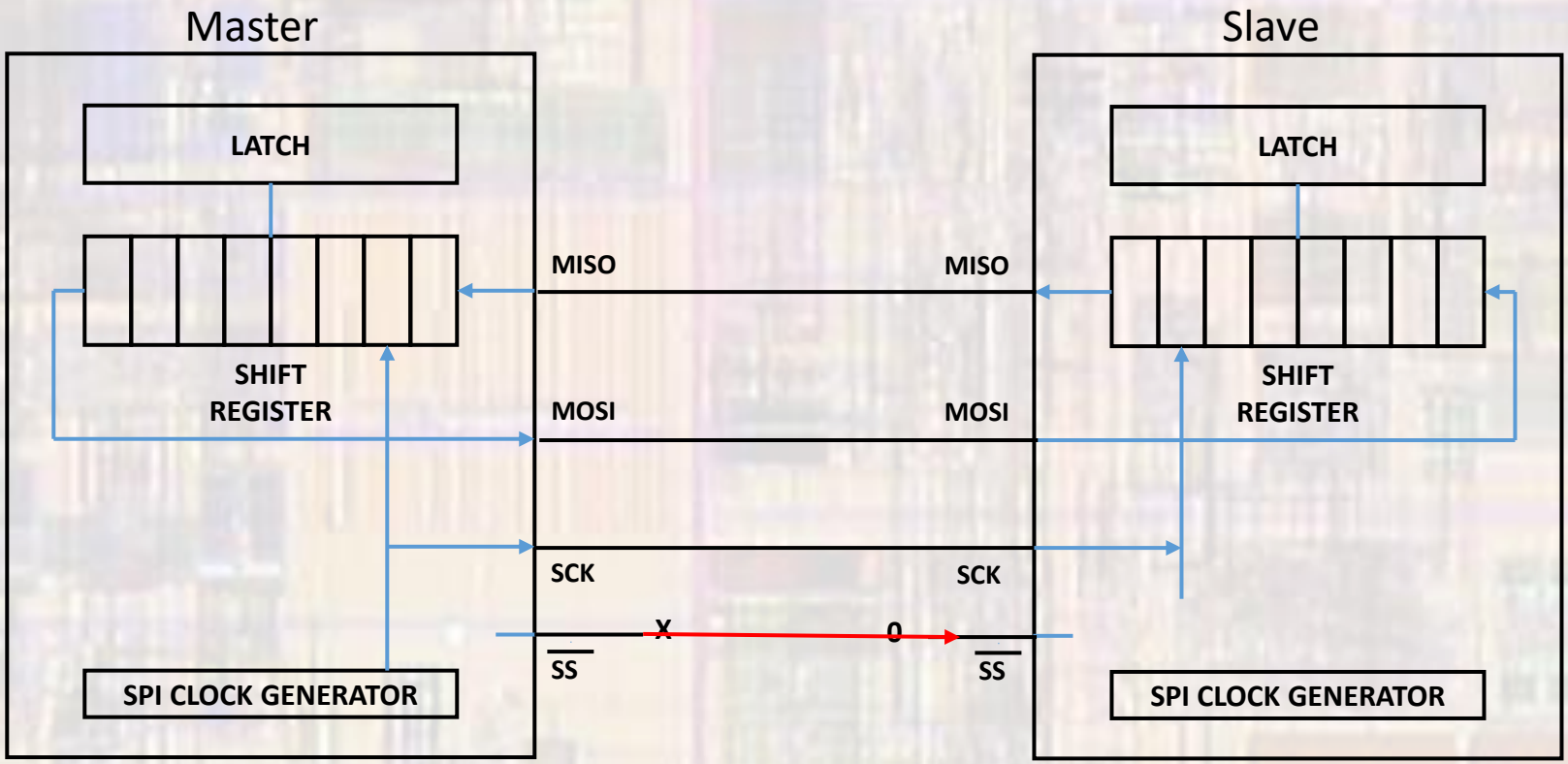
SCK – SPI CLK

$\overline{SS}$ – Slave Select Bar

© tj

SPI

- Operation

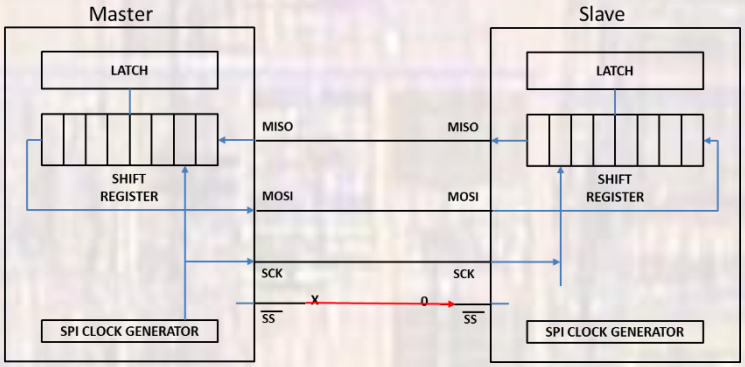


Latch → Shift Register in both master and slave
Master generates 8 clocks → shifts both registers (swaps content)
Shift Register → Latch in both master and slave

© tj

SPI

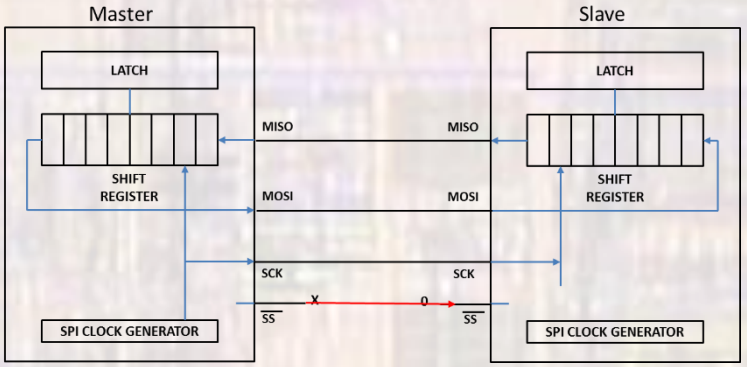
- Operation
  - Configure 1 device as master
  - Configure 1 or more devices as slaves
  - Load values into register(s)
  - Pull SSbar low on the desired slave device
  - Initiate transfer by pulsing Start
    - The master will generate the appropriate clocks

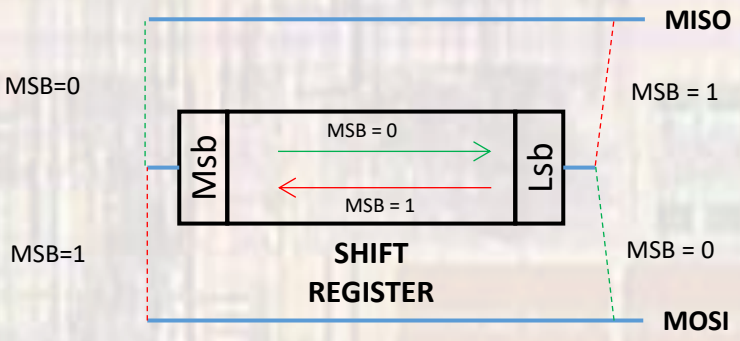


© tj

SPI

- Operation
  - 2 options for clock polarity
    - CKPL = 0 → rising edge triggered
    - CKPL = 1 → falling edge triggered
  - 2 options for clock phase
    - CKPH = 0 → leading edge triggered
    - CKPH = 1 → trailing edge triggered
  - 2 options on transfer direction
    - MSB = 0 → LSB transferred first
    - MSB = 1 → MSB transferred first

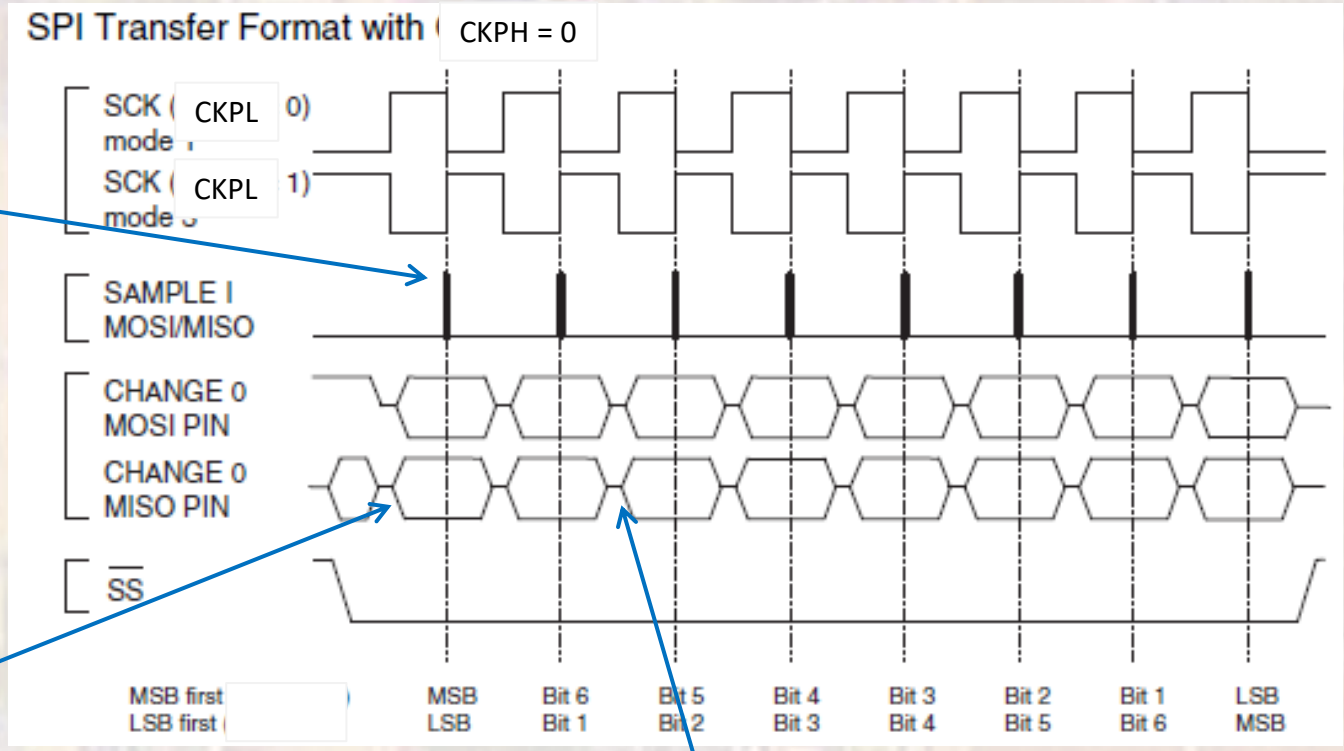




© tj

SPI

- Operation
  - CKPH = 0



Captured in register on trailing clock edge

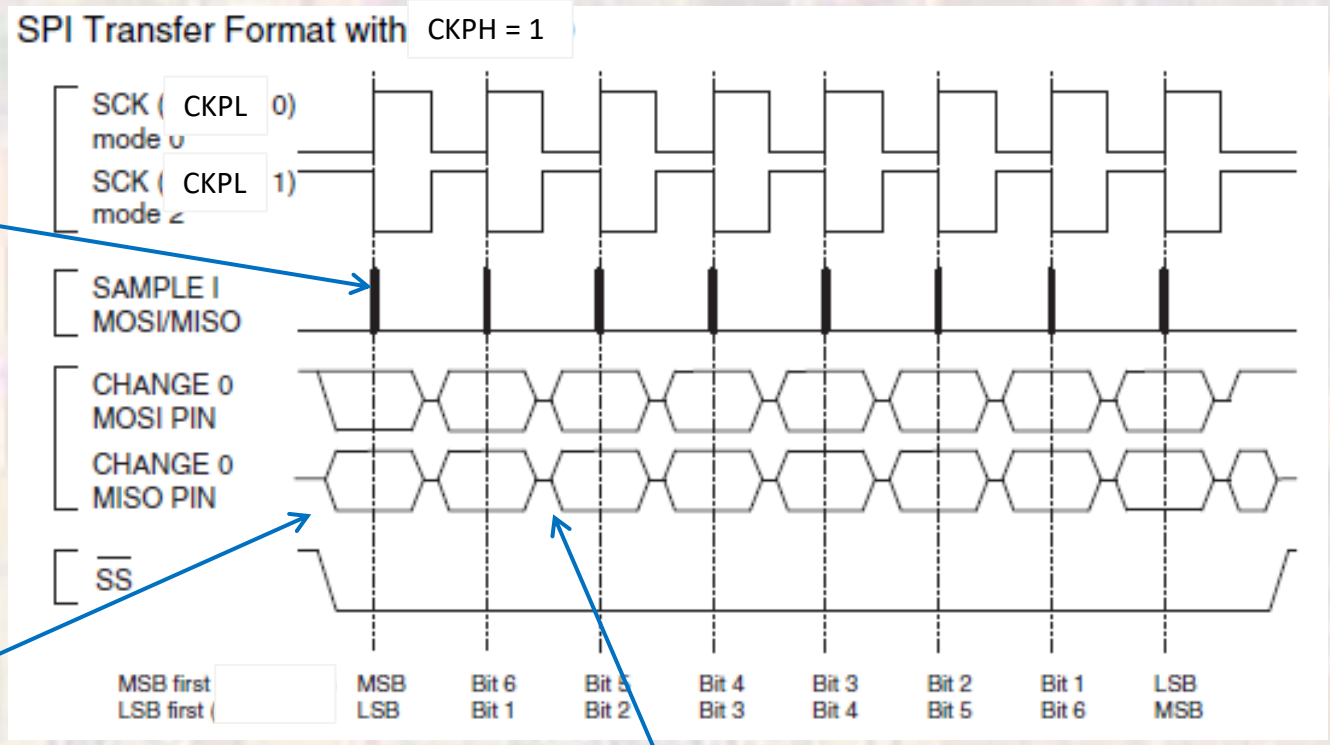
Values active on pins on first clock edge

New values placed on pins on leading clock edge

© tj

SPI

- Operation
  - CKPH = 1



Captured in register on leading clock edge

Values active on pins as soon as SSbar goes low

New values placed on pins on trailing clock edge

© tj

SPI

- Operation
  - Multiple Slave Configuration

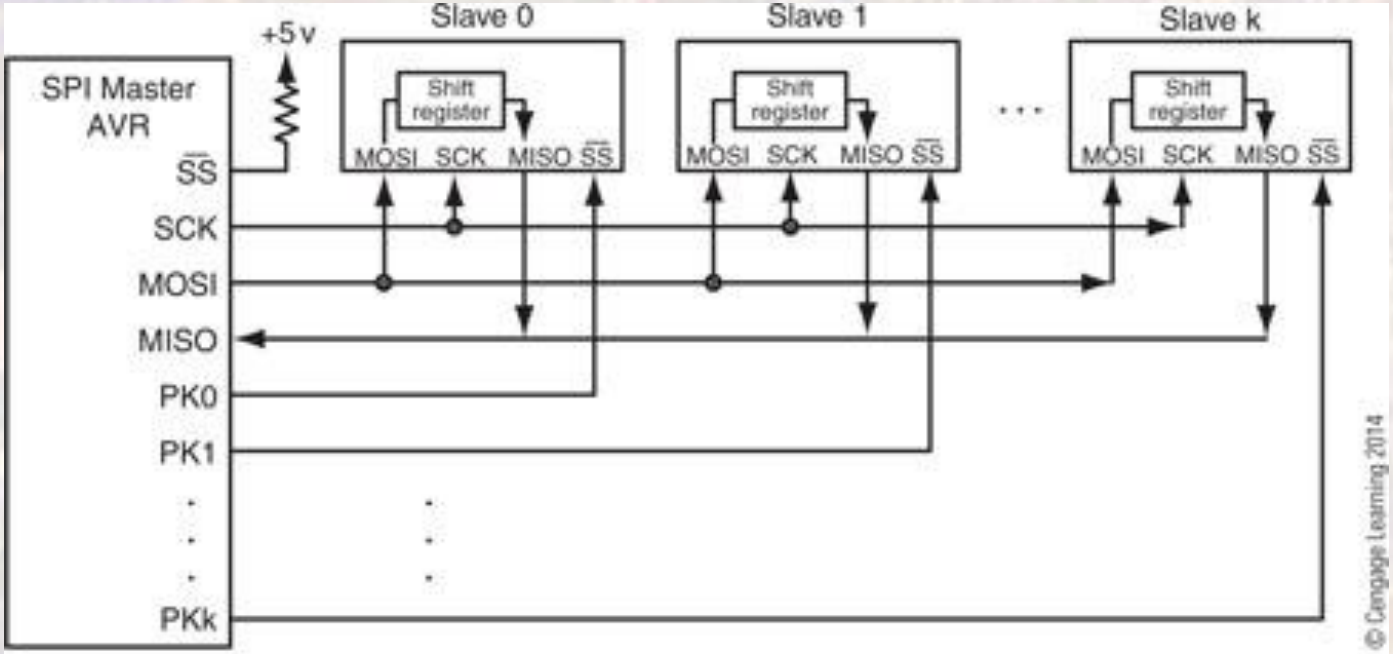


Figure 14.7 ■ Single-master and multiple-slave device connection (method 1)

SPI

- Operation
  - Multiple Slave – Extended Shift Configuration

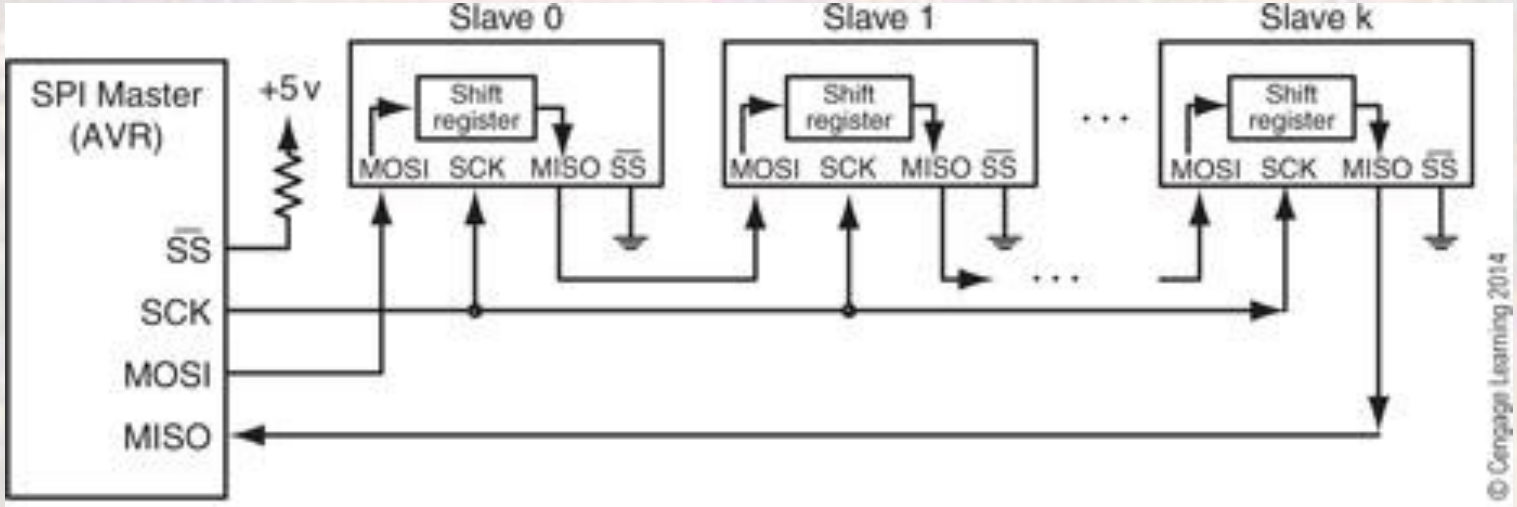
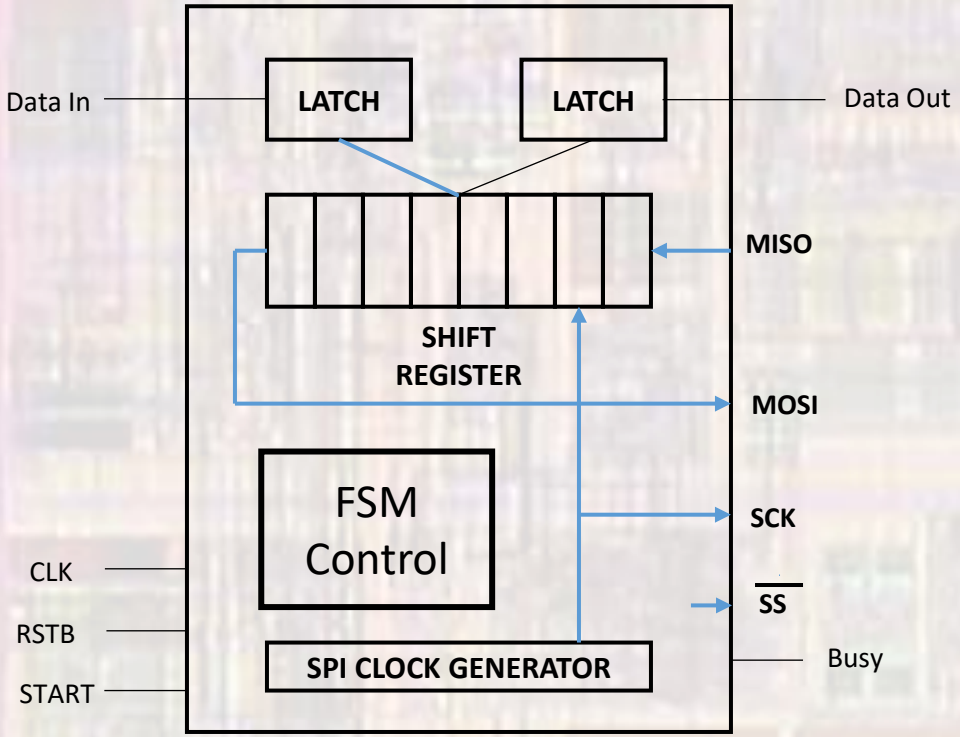


Figure 14.8 ■ Single-master and multiple-slave device connection (method 2)

SPI

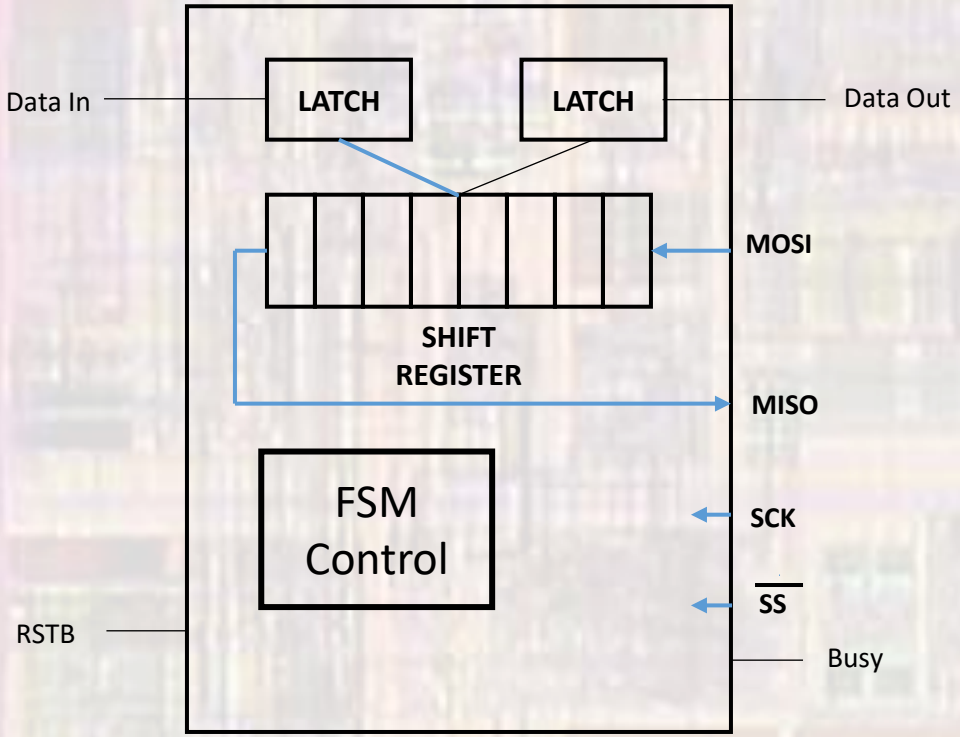
- Master
  - Separate RX/TX registers
  - State Machine
  - Clock divider
  - Status signals



© tj

SPI

- Slave
  - Separate RX/TX registers
  - State Machine
  - Status signals



© tj

SPI

- Master VHDL - description

```
------------------------------------------------
--
-- spi_master_example.vhdl
--
-- created 8/22/26 by tj
--
------------------------------------------------
-- FSM implementation
--
-- Configuration
--    SPI Mode 0
--       CPOL = 0    SCLK Idle low
--       CPHA = 0    Sampled on leading edge, transferred on trailing edge
--                   rising                      Falling
--    MSB first
--
-- SPI Interface
--    SCLK  - Serial Clock
--    MOSI  - Master Out / Slave In
--    MISO  - Master In / Slave Out
--    CS_N  - Active-Low Chip Select
--
-- Module Interface
--    clk         input: system clock - divided inside the block
--    reset       input: active high - not the system i_reset, i_reset for this block only
--    start       input: single pulse
--    tx_data     input: data to transmit
--    rx_data     output: received data
--    busy        output: transfer active
--    done        output: single pulse
--
-- Generics
--    CLK_DIVIDER    SCLK relative to system clock
--    DATA_WIDTH     SPI transfer size (typically 8 bits)
--
------------------------------------------------------------------------------
```


© tj

SPI

- Master VHDL - entity

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

entity spi_master_example is
    generic(
            CLK_DIVIDER:    positive := 4;
            DATA_WIDTH:     positive := 8
        );
    port(
        i_clk_in:           in  std_logic;
        i_reset_in:         in  std_logic;

        i_start_in:         in  std_logic;

        i_tx_data_in:       in  std_logic_vector((DATA_WIDTH - 1) downto 0);
        o_rx_data_out:      out std_logic_vector((DATA_WIDTH - 1) downto 0);

        o_busy_out:         out std_logic;
        o_done_out:         out std_logic;

        o_spi_cs_n_out:     out std_logic;
        o_spi_sclk_out:     out std_logic;
        o_spi_mosi_out:     out std_logic;
        i_spi_miso_in:      in std_logic
      );
end entity;
```

SPI

- Master VHDL - signals

```
architecture behavioral of spi_master_example is

   type STATE_TYPE is (Idle, Setup, Rise, Fall, Complete);
   signal state:        STATE_TYPE;
   signal state_next:   STATE_TYPE;

   signal clk_divider_count:  unsigned((integer(ceil(log2(real(CLK_DIVIDER)))) - 1) downto 0);
   signal spi_tick:           std_logic;

   signal tx_shift_reg:       std_logic_vector((DATA_WIDTH - 1) downto 0);
   signal rx_shift_reg:       std_logic_vector((DATA_WIDTH - 1) downto 0);

   signal transfer_bit_count: unsigned((integer(ceil(log2(real(DATA_WIDTH + 1)))) - 1) downto 0);

   signal cs_control:         std_logic;
   signal sclk_control:       std_logic;
   signal load_data:          std_logic;
   signal transfer_done:      std_logic;
```


© tj

SPI

- Master VHDL – clock generator

```
begin
   ------------------------------------------------------------------------
   --   spi_tick is a clock ENABLE, not a new clock.
   --   The entire design continues to use the system clock.
   ------------------------------------------------------------------------
   ckl_div: process(i_clk_in, i_reset_in)
   begin
      if (i_reset_in = '1') then
         clk_divider_count <= (others => '0');
         spi_tick <= '0';
      elsif rising_edge(i_clk_in) then
         if (clk_divider_count = to_unsigned(((CLK_DIVIDER / 2) - 1), clk_divider_count'length)) then
            clk_divider_count <= (others => '0');
            spi_tick <= '1';
         else
            clk_divider_count <= clk_divider_count + 1;
            spi_tick <= '0';
         end if;
      end if;
   end process;
```


© tj

SPI

- Master VHDL – data transfer

```
-------------------------------------------------
-- Transfer data into and out of the module
-------------------------------------------------
data_process: process(i_clk_in, i_reset_in)
begin
   if (i_reset_in = '1') then
      tx_shift_reg            <= (others => '0');
      rx_shift_reg            <= (others => '0');
      transfer_bit_count      <= (others => '0');
   elsif rising_edge(i_clk_in) then
      -- load data
      if (load_data = '1') then
         tx_shift_reg         <= i_tx_data_in;
         rx_shift_reg         <= (others => '0');

         transfer_bit_count   <= (others => '0');

      -- read MISO
      elsif ((state = Rise) and (spi_tick = '1')) then
         tx_shift_reg         <= tx_shift_reg;
         rx_shift_reg         <= rx_shift_reg((DATA_WIDTH - 2) downto 0) & i_spi_miso_in;

         transfer_bit_count   <= transfer_bit_count + 1;

      -- shift MOSI
      elsif ((state = Fall) and (spi_tick = '1')) then
         tx_shift_reg         <= tx_shift_reg((DATA_WIDTH - 2) downto 0) & '0';
         rx_shift_reg         <= rx_shift_reg;

         transfer_bit_count   <= transfer_bit_count;

      else
         tx_shift_reg         <= tx_shift_reg;
         rx_shift_reg         <= rx_shift_reg;

         transfer_bit_count   <= transfer_bit_count;
      end if;
   end if;
end process;
```


© tj

SPI

- Master VHDL – FSM – next state logic

```
fsm_next_state: process(state, i_start_in, spi_tick, transfer_bit_count)
begin
   case state is
      when Idle =>
         if (i_start_in = '1') then
            state_next <= Setup;
         else
            state_next <= state;
         end if;
      when Setup =>
         if (spi_tick = '1') then
            state_next <= Rise;
         else
            state_next <= state;
         end if;
      when Rise =>
         if (spi_tick = '1') then
            state_next <= Fall;
         else
            state_next <= state;
         end if;
      when Fall =>
         if (transfer_bit_count = to_unsigned(DATA_WIDTH, transfer_bit_count'length)) then
            state_next <= Complete;
         else
            if (spi_tick = '1') then
               state_next <= Rise;
            else
               state_next <= state;
            end if;
         end if;
      when Complete =>
         state_next <= Idle;
      when others =>
         state_next <= Idle;
   end case;
end process;
```


© tj

SPI

- Master VHDL – FSM – register logic

```
fsm_register: process(i_clk_in, i_reset_in)
begin
   if (i_reset_in = '1') then
      state <= Idle;
   elsif rising_edge(i_clk_in) then
      state <= state_next;
   end if;
end process;
```


© tj

SPI

- Master VHDL – FSM – output logic

```
fsm_output: process(state, i_start_in, tx_shift_reg)
begin
   case state is
      when Idle =>
         cs_control     <= '1';
         sclk_control   <= '0';

         if (i_start_in = '1') then
            load_data   <= '1';
         else
            load_data   <= '0';
         end if;

         o_done_out     <= '1';
         o_spi_mosi_out <= '0';
         o_busy_out     <= '0';
      when Setup =>
         cs_control     <= '0';
         sclk_control   <= '0';
         load_data      <= '0';
         o_done_out     <= '1';
         o_spi_mosi_out <= tx_shift_reg(DATA_WIDTH-1);
         o_busy_out     <= '1';
      when Rise =>
         cs_control     <= '0';
         sclk_control   <= '1';
         load_data      <= '0';
         o_done_out     <= '0';
         o_spi_mosi_out <= tx_shift_reg(DATA_WIDTH-1);
         o_busy_out     <= '1';
```

```
      when Fall =>
         cs_control     <= '0';
         sclk_control   <= '0';
         load_data      <= '0';
         o_done_out     <= '0';
         o_spi_mosi_out <= tx_shift_reg(DATA_WIDTH-1);
         o_busy_out     <= '1';
      when Complete =>
         cs_control     <= '1';
         sclk_control   <= '0';
         load_data      <= '0';
         o_done_out     <= '1';
         o_spi_mosi_out <= tx_shift_reg(DATA_WIDTH-1);
         o_busy_out     <= '1';
      when others =>
         cs_control     <= '1';
         sclk_control   <= '0';
         load_data      <= '0';
         o_done_out     <= '0';
         o_spi_mosi_out <= '0';
         o_busy_out     <= '0';
   end case;
  end process;
```

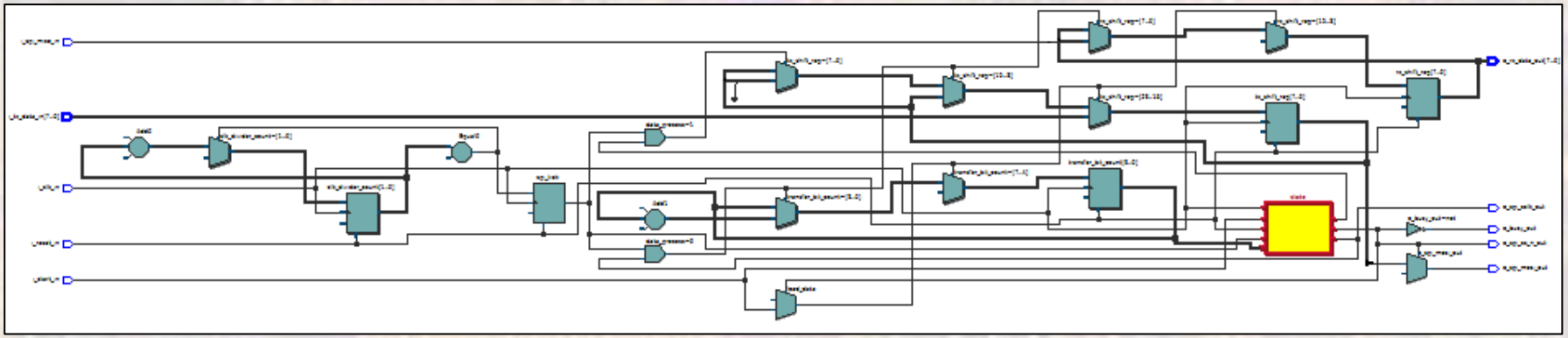
```
   -- outputs
     o_rx_data_out  <= rx_shift_reg;
     o_spi_cs_n_out <= cs_control;
     o_spi_sclk_out <= sclk_control;

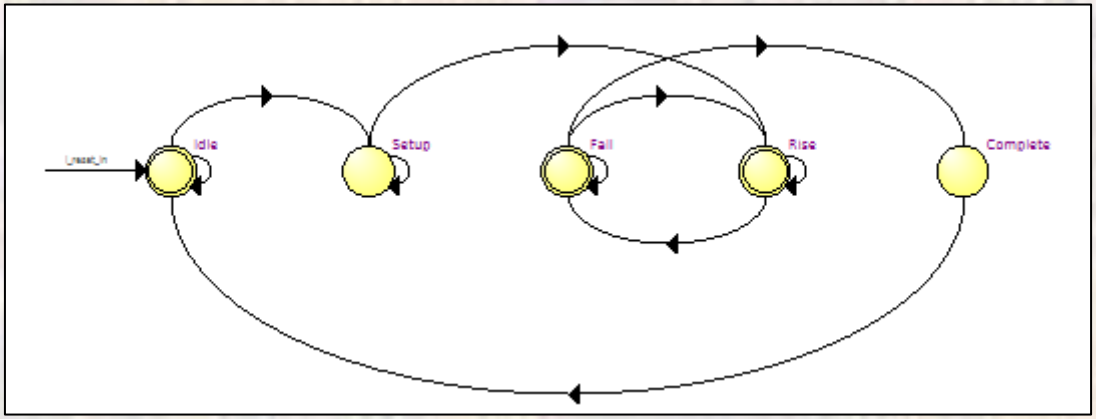
end architecture;
```


© tj

SPI

- Master RTL





 © tj

SPI

- Slave VHDL - description

```
------------------------------------------------
--
-- spi_slave_example.vhdl
--
-- created 9/21/26 by tj
--
------------------------------------------------------------------------------------------
-- FSM implementation
--
-- Configuration
--    SPI Mode 0
--       CPOL = 0    SCLK Idle low
--       CPHA = 0    Sampled on leading edge, transferred on trailing edge
--                   rising                       Falling
--    MSB first
--
-- SPI Interface
--    SCLK  - Serial Clock
--    MOSI  - Master Out / Slave In
--    MISO  - Master In / Slave Out
--    CS_N  - Active-Low Chip Select
--
-- Module Interface
--
--    reset       input: active high - not the system reset, reset for this block only
--
--    tx_data     input: data to transmit
--    rx_data     output: received data
--    busy        output: transfer active
--    done        output: single pulse
--
-- Generics
--
--    DATA_WIDTH     SPI transfer size (typically 8 bits)
--
------------------------------------------------------------------------------
```


© tj

SPI

- Slave VHDL - entity

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

entity spi_slave_example is
   generic(
            DATA_WIDTH: positive := 8
          );
   port(
         i_reset_in:       in    std_logic;

         i_tx_data_in:     in    std_logic_vector((DATA_WIDTH-1) downto 0);
         o_rx_data_out:    out   std_logic_vector((DATA_WIDTH-1) downto 0);

         o_busy_out:       out   std_logic;
         o_done_out:       out   std_logic;

         i_spi_cs_n_in:    in    std_logic;
         i_spi_sclk_in:    in    std_logic;
         i_spi_mosi_in:    in    std_logic;
         o_spi_miso_out:   out   std_logic
      );
end entity;
```


© tj

SPI

- Slave VHDL - signals

```
architecture behavioral of spi_slave_example is
    ---------------------------------------------------------------------------
    -- The SPI clock provides the rising and falling edges.
    ---------------------------------------------------------------------------

    type STATE_TYPE is (Idle, Transfer);
    signal state:        STATE_TYPE;
    signal state_next:   STATE_TYPE;

    ---------------------------------------------------------------------------
    -- The first TX bit is handled separately because, in SPI Mode 0, it must
    -- already be present on MISO BEFORE the first rising edge of SCLK.
    ---------------------------------------------------------------------------
    signal rx_shift_reg: std_logic_vector((DATA_WIDTH - 1) downto 0);
    signal tx_shift_reg: std_logic_vector((DATA_WIDTH - 1) downto 0);

    signal bit_count:    unsigned((integer(ceil(log2(real(DATA_WIDTH + 1)))) - 1) downto 0);
```


© tj

SPI

- Slave VHDL – data transfer

```
data_receive_process: process(i_spi_sclk_in, i_reset_in)
begin
   if (i_reset_in = '1') then
      rx_shift_reg   <= (others => '0');
      bit_count      <= (others => '0');
   elsif rising_edge(i_spi_sclk_in) then
      if (i_spi_cs_n_in = '0') then
         -- clear the receive register first and place MOSI into bit 0.
         if (state = Idle) then
            rx_shift_reg      <= (others => '0');
            rx_shift_reg(0)   <= i_spi_mosi_in;
            bit_count         <= to_unsigned(1, bit_count'length);
         else
            rx_shift_reg      <= rx_shift_reg((DATA_WIDTH - 2) downto 0) & i_spi_mosi_in;
            bit_count         <= bit_count + 1;
         end if;
      end if;
   end if;
end process;
```

```
data_transmit_process: process(i_spi_sclk_in, i_reset_in)
begin
   if (i_reset_in = '1') then
      tx_shift_reg <= (others => '0');
   elsif falling_edge(i_spi_sclk_in) then
      if (i_spi_cs_n_in = '0') then
         if (bit_count = to_unsigned(1, bit_count'length)) then
            tx_shift_reg <= i_tx_data_in((DATA_WIDTH - 2) downto 0) & '0';
         else
            tx_shift_reg <= tx_shift_reg((DATA_WIDTH - 2) downto 0) & '0';
         end if;
      end if;
   end if;
end process;
```


© tj

SPI

- Slave VHDL – FSM – next state logic

```
fsm_next_state: process(state, i_spi_cs_n_in, bit_count)
begin
   case state is
      when Idle =>
         if (i_spi_cs_n_in = '0') then
            state_next <= Transfer;
         else
            state_next <= Idle;
         end if;
      when Transfer =>
         if (i_spi_cs_n_in = '1') then
            state_next <= Idle;
         -- need to allow final bit full clock cycle - so overcount by 1
         elsif (bit_count = to_unsigned(((DATA_WIDTH - 1) + 1), bit_count'length)) then
            state_next <= Idle;
         else
            state_next <= Transfer;
         end if;
      when others =>
         state_next <= Idle;
   end case;
end process;
```


© tj

SPI

- Slave VHDL – FSM – register logic

```
fsm_register: process(i_spi_sclk_in, i_reset_in, i_spi_cs_n_in)
begin
   if (i_reset_in = '1') then
      state <= Idle;
   elsif (i_spi_cs_n_in = '1') then
      state <= Idle;
   elsif rising_edge(i_spi_sclk_in) then
      state <= state_next;
   end if;
end process;
```


© tj

SPI

- Slave VHDL – FSM – output logic

```
   fsm_output: process(state, i_spi_cs_n_in, i_spi_sclk_in, bit_count, tx_shift_reg, i_tx_data_in)
   begin
      case state is
         when Idle =>
            o_busy_out        <= '0';
            if (i_spi_cs_n_in = '0') then
               o_spi_miso_out <= i_tx_data_in(DATA_WIDTH - 1);
            else
               o_spi_miso_out <= '0';
            end if;
         when Transfer =>
            o_busy_out        <= '1';
            if ((bit_count = to_unsigned(1, bit_count'length)) and (i_spi_sclk_in = '1')) then
               o_spi_miso_out <= i_tx_data_in(DATA_WIDTH - 1);
            else
               o_spi_miso_out <= tx_shift_reg(DATA_WIDTH - 1);
            end if;
         when others =>
            o_busy_out        <= '0';
            o_spi_miso_out    <= '0';
      end case;
   end process;

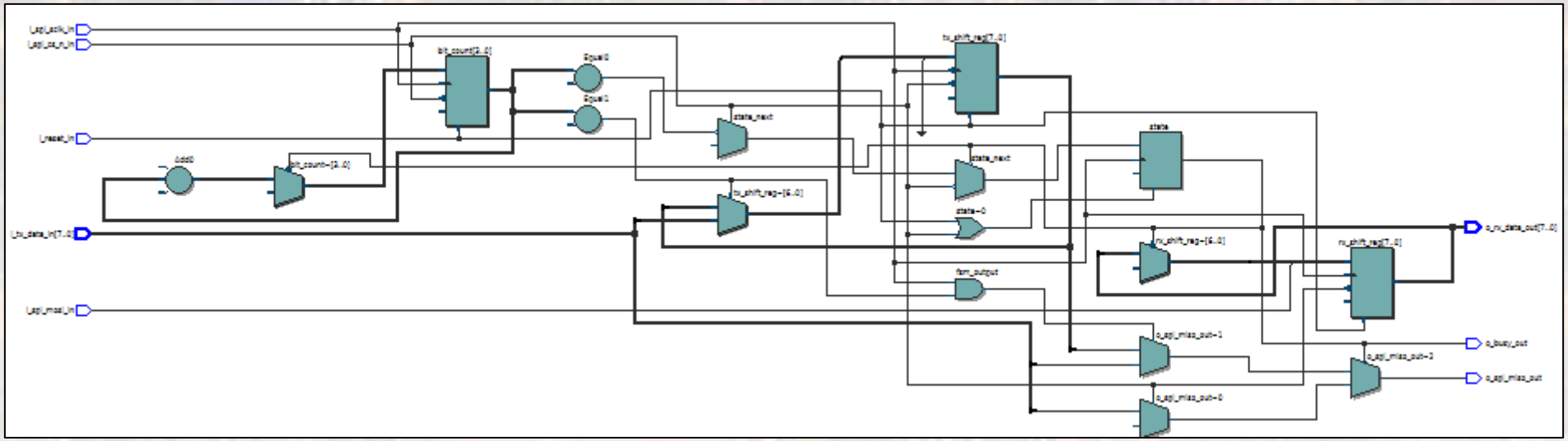
   -- outputs
   o_rx_data_out <= rx_shift_reg;

end architecture;
```


© tj

SPI

- Slave RTL
  - Why no state machine ??



© tj

SPI

- Master – Slave Simulation

```
# run -all
# ** Note: ========================================
#    Time: 14130 ns  Iteration: 0  Instance: /spi_master_slave_example_tb
# ** Note: ALL SPI TESTS COMPLETED
#    Time: 14130 ns  Iteration: 0  Instance: /spi_master_slave_example_tb
# ** Note: ========================================
#    Time: 14130 ns  Iteration: 0  Instance: /spi_master_slave_example_tb
# ** Failure: SPI TESTBENCH COMPLETE.
#    Time: 14210 ns  Iteration: 0  Process: /spi_master_slave_example_tb/test_process File: C:/Users/johnsontimoj/ELE3510/Quartus_Projects_25.1/SPI/spi_master_slave_example_tb.vh
dl
```

