SPI Master Design

Lewis Collum - Clarkson University

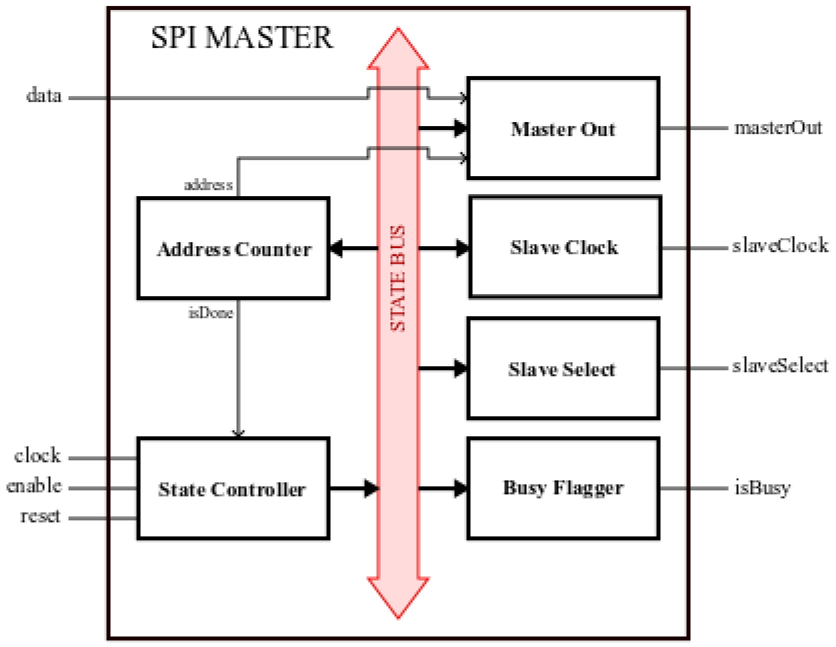

Figure 1. Conceptual model for the SPI master controller. This controller takes in a clock and the data to send, and outputs the slave clock, slave select and the master out, as well as a "busy" flag.

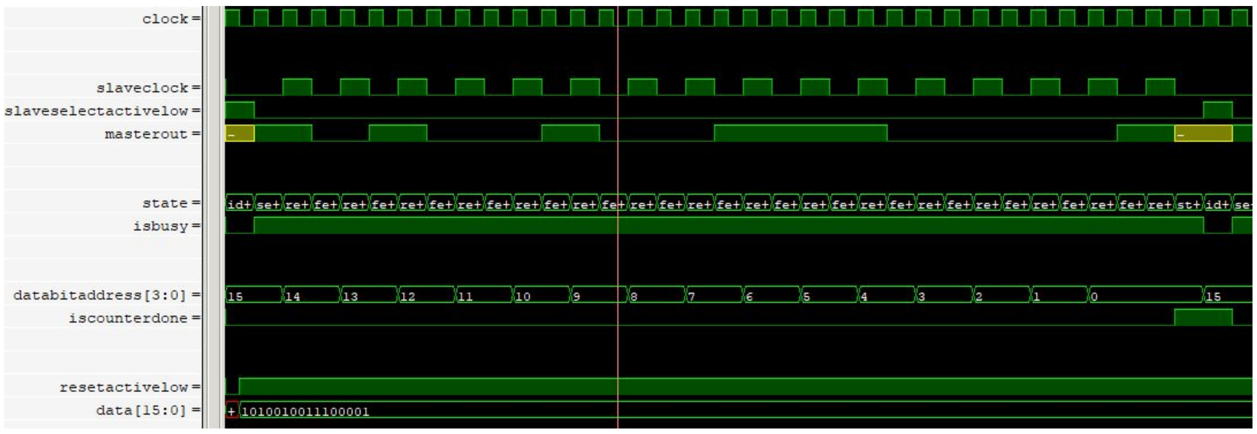

Figure 2. Illustrates *master out* correctly fetching data.

A. common.vhdl

This package creates types that are used throughout the project.

1. ***stateType*** is an enumeration for the states.
2. ***word*** is the length of the data to be sent.
3. ***counter*** is used in the *Address Counter* (figure 1) to create an unsigned with a max value that is the length of ***word***.
4. ***validFrequencyRange*** is the valid frequency range for the Sparkfun Seven-Segment Display.

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

package common is
    type stateType is (idle, selecting, fetching, receiving, stopping, disabled);
    subtype word is unsigned(15 downto 0);
    subtype counter is unsigned(integer(log((real(word'length)) / log(real(2)))-0.5) downto 0);
    subtype validFrequencyRange is integer range 1 to 500e3;
end package common;
```

B. SPIMaster.vhdl

Contains modules: *Busy Flagger, State Controller, Slave Clock, Slave Select,* and *Master Out* (figure 1). Each of these modules is a process. These processes should be extracted into their own VHDL entity to improve encapsulation and readability.

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

library work;
use work.common.all;

entity SPIMaster is
        generic(
         constant slaveClockPolarity: std_logic);
        port(
        clock: in std_logic;
        data: in word;
        resetActiveLow: in std_logic;
        enable: in std_logic;
        isBusy: out std_logic;
        masterOut: out std_logic;
        slaveClock: buffer std_logic;
        slaveSelectActiveLow: out std_logic);
end SPIMaster;

architecture CPHA0 of SPIMaster is
    component SPIAddressCounter is
         port(
                 state: inout stateType;
                 address: buffer counter;
                 isDone: out std_logic);
    end component SPIAddressCounter;

    signal dataBitAddress: counter;
    signal state: stateType;
    signal isCounterDone: std_logic;

begin
    stateControl : process(clock, isCounterDone)
         procedure handleCounterInterrupt;
         procedure updateAllStates;
         procedure updateCyclicalStates;
         impure function isReadyToSelect return boolean;
         impure function isReadyToFetch return boolean;
         impure function isReadyToReceive return boolean;
         impure function isReadyToIdle return boolean;
         impure function isLastBitOfData return boolean;

         procedure handleCounterInterrupt is begin
                 state <= stopping;
         end procedure handleCounterInterrupt;

         procedure updateAllStates is begin
                 if resetActiveLow = '0' then
                         state <= idle;
                 elsif enable = '1' then
                         updateCyclicalStates;
                 else
                         state <= disabled;
                 end if;
         end procedure updateAllStates;
```

```vhdl
	procedure updateCyclicalStates is begin
		if isReadyToIdle then
			state <= idle;
		elsif isReadyToSelect then
			state <= selecting;
		elsif isReadyToFetch then
			state <= fetching;
		elsif isReadyToReceive then
			state <= receiving;
		end if;
	end procedure updateCyclicalStates;

	impure function isReadyToSelect return boolean is begin
		return state = idle;
	end function isReadyToSelect;

	impure function isReadyToFetch return boolean is begin
		return state = receiving;
	end function isReadyToFetch;

	impure function isReadyToReceive return boolean is begin
		return state = fetching or state = selecting;
	end function isReadyToReceive;

	impure function isReadyToIdle return boolean is begin
		return state = stopping;
	end function isReadyToIdle;

	impure function isLastBitOfData return boolean is begin
		return dataBitAddress = dataBitAddress'low;
	end function isLastBitOfData;
begin
	if rising_edge(isCounterDone) then
		handleCounterInterrupt;
	elsif rising_edge(clock) then
		updateAllStates;
	end if;
end process stateControl;


slaveClock_StateMachine : process(state)
	procedure reset is begin
		slaveClock <= slaveClockPolarity;
	end procedure reset;

	procedure trailingEdge is begin
		slaveClock <= slaveClockPolarity;
	end procedure trailingEdge;

	procedure leadingEdge is begin
		slaveClock <= not slaveClockPolarity;
	end procedure leadingEdge;
begin
	case state is
		when idle | stopping =>
			reset;
		when fetching =>
			trailingEdge;
		when receiving =>
			leadingEdge;
		when others =>
			null;
	end case;
end process slaveClock_StateMachine;
```

```
    slaveSelect_StateMachine : process(state)
         procedure deselectSlave is begin
                 slaveSelectActiveLow <= '1';
         end procedure deselectSlave;

         procedure selectSlave is begin
                 slaveSelectActiveLow <= '0';
         end procedure selectSlave;
    begin
         case state is
                 when idle =>
                         deselectSlave;
                 when selecting =>
                         selectSlave;
                 when others =>
                         null;
         end case;
    end process slaveSelect_StateMachine;


    masterOut_StateMachine : process(state)
         procedure fetchBit is begin
                 masterOut <= data(to_integer(dataBitAddress));
         end procedure fetchBit;

         procedure releaseBit is begin
                 masterOut <= '-';
         end procedure releaseBit;
    begin
         case state is
                 when idle | stopping =>
                         releaseBit;
                 when fetching | selecting =>
                         fetchBit;
                 when others =>
                         null;
         end case;
    end process masterOut_StateMachine;

    isBusy_StateMachine: process(state)
    begin
         case state is
                 when idle =>
                         isBusy <= '0';
                 when selecting =>
                         isBusy <= '1';
                 when others =>
                         null;
         end case;
    end process isBusy_StateMachine;

    dataBitAddressCounter: SPIAddressCounter
         port map(
                 state => state,
                 address => dataBitAddress,
                 isDone => isCounterDone);

end CPHA0;
```

C. SPIAddressCounter.vhdl

Generates the address of the bit that is being read from the data. Also sends an interrupt when done.

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

library work;
use work.common.all;

entity SPIAddressCounter is
    port(
         state: in stateType;
         address: buffer counter;
         isDone: out std_logic);
end SPIAddressCounter;

architecture countDown of SPIAddressCounter is
    signal isLastFetch: std_logic;
    signal isLastReceive: std_logic;
begin
    stateControl: process(state)
         procedure resetCounter is begin
                 address <= to_unsigned(word'high, address'length);
         end procedure resetCounter;

         procedure resetFlags is begin
                 isLastFetch <= '0';
                 isLastReceive <= '0';
         end procedure resetFlags;

         procedure decrementCounter is begin
                 address <= address - 1;
         end procedure decrementCounter;

         procedure updateOnFetch is begin
                 if address = 0 then
                         isLastFetch <= '1';
                 end if;
         end procedure updateOnFetch;

         procedure updateOnReceive is begin
                 if isLastFetch = '1' then
                         isLastReceive <= '1';
                 else
                         decrementCounter;
                 end if;
         end procedure updateOnReceive;

    begin
         case state is
                 when idle =>
                         resetCounter;
                         resetFlags;
                 when fetching =>
                         updateOnFetch;
                 when receiving =>
                         updateOnReceive;
                 when others =>
                         null;
         end case;
    end process stateControl;

    setIsDoneInterrupt: process(state)
    begin
```

```
          if isLastReceive = '1' then
                  isDone <= '1';
          else
                  isDone <= '0';
          end if;
     end process setIsDoneInterrupt;

end architecture countDown;
```

D. timingUtil.vhdl

Helper package for testing.

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

package TimingUtil is
    function frequencyToPeriod(frequency: integer) return time;
    procedure generateClock(signal clock: out std_logic; frequency: integer);
    procedure highPulseForTime(signal output: out std_logic; duration: time);
    procedure lowPulseForTime(signal output: out std_logic; duration: time);
end package TimingUtil;

package body TimingUtil is

    function frequencyToPeriod(frequency: integer) return time is
    begin
         return 1 sec / frequency;
    end function frequencyToPeriod;

    procedure generateClock(signal clock: out std_logic; frequency: integer) is
         constant period: time := 1 sec / frequency;
         constant halfPeriod: time := period / 2;
    begin
         loop
                 clock <= '1';
                 wait for halfPeriod;
                 clock <= '0';
                 wait for halfPeriod;
         end loop;
    end procedure generateClock;

    procedure highPulseForTime(signal output: out std_logic; duration: time) is
    begin
         output <= '1';
         wait for duration;
         output <= '0';
    end procedure highPulseForTime;

    procedure lowPulseForTime(signal output: out std_logic; duration: time) is
    begin
         output <= '0';
         wait for duration;
         output <= '1';
    end procedure lowPulseForTime;

end package body TimingUtil;
```

E. Test_SPIMaster.vhdl

Test Entity for SPIMaster

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

library work;
use work.common.all;
use work.timingUtil.all;


entity Test_SPIMaster is
end entity Test_SPIMaster;

architecture simulation of Test_SPIMaster is
    component SPIMaster is
        generic(
                constant slaveClockPolarity: std_logic);
        port(
                clock: in std_logic;
                data: in word;
                resetActiveLow: in std_logic;
                enable: in std_logic;
                isBusy: out std_logic;
                masterOut: out std_logic;
                slaveClock: buffer std_logic;
                slaveSelectActiveLow: out std_logic);
    end component SPIMaster;

    constant clockFrequency: validFrequencyRange := 250e3;
    constant clockPeriod: time := frequencyToPeriod(clockFrequency);

    signal clock: std_logic;
    signal dataToSend: word;
    signal resetActiveLow: std_logic;
    signal enable: std_logic;
    signal isBusy: std_logic;
    signal masterOut, slaveClock, slaveSelectActiveLow: std_logic;

begin

    generateClock(clock, frequency => clockFrequency);

    uut: SPIMaster
        generic map(
                slaveClockPolarity => '0')
        port map(
                clock => clock,
                data => dataToSend,
                resetActiveLow => resetActiveLow,
                enable => enable,
                isBusy => isBusy,
                masterOut => masterOut,
                slaveClock => slaveClock,
                slaveSelectActiveLow => slaveSelectActiveLow);

    test : process is
        constant testWord: word := x"A4E1";

        procedure setup is
        begin
                enable <= '1';
                lowPulseForTime(resetActiveLow, clockPeriod/2);
        end procedure setup;
        procedure testFetchWord is
        begin
                setup;
                dataToSend <= testWord;
                wait for clockPeriod*2*19;
        end procedure testFetchWord;

        procedure testResetWhileFetching is
        begin
                setup;
                dataToSend <= testWord;
                wait for 6*clockPeriod;
                lowPulseForTime(resetActiveLow, clockPeriod);
                wait for clockPeriod*10;
        end procedure testResetWhileFetching;

        procedure testDisableWhileFetching is
        begin
                setup;
                dataToSend <= testWord;
                wait for 6*clockPeriod;
                lowPulseForTime(enable, 4*clockPeriod);
                wait for 8*clockPeriod;
        end procedure testDisableWhileFetching;

    begin
        testFetchWord;`
        report "END OF TEST" severity failure; --TODO figure out best practice to end a test properly.
    end process test;

end architecture simulation;
```