

Developing SPI drivers for FPGA

Last Updated Friday, 01 May 2009

Introduction

A common issue when programming FPGA's as part of a multi-processor system is how to transfer data into and out of the part. The Serial Peripheral Interface (SPI) is easy to implement on an FPGA, is supported by a large number of micro-processor parts and has the benefit of requiring a low pin count. In this article, the code to implement a SPI bus on an FPGA part is developed and used to illustrate two fundamentals of FPGA design – clocked processes and synchronising information across clock domains. The code is presented as VHDL for a Xilinx part, although the principles hold true regardless of language or specific manufacturer.

Clocked processes

For those from a software background, the idea of a clocked process may be a new one. Within a standard micro-processor part, the programmer's code is written and executed sequentially, with each basic assembler instruction taking one or more clock ticks. One of the major benefits of using an FPGA is that it allows for true parallel processing in the sense that different networks of gates can process information concurrently. However, the time each of these parallel networks take to complete their processing is dependent on the particular routing chosen for each network and the processing load of each section, and thus sharing information between these networks becomes a non-trivial problem. One approach would be to balance the timings so that data is valid when it is transferred. However this approach is poor since a slight change in routing, or a change to the instantiated algorithm could cause unforeseen data corruption. Fortunately the approach of using clocked processes offers a far better solution. It is an oft quoted maxim of FPGA programming that “Nothing should happen without a clock”. This means that processing only ever occurs in response to a rising or falling clock edge. This allows an algorithm to be broken down into discrete chunks, each of which is capable of being completed within a clock period. Often, each of these chunks is encapsulated in one of the states of a finite state machine. By comparing the state machines of parallel networks, it becomes straightforward to build combinations of states which will ensure valid data transfers.

SPI code

For the purposes of the current demonstration, the FPGA will be the SPI slave driven by an external master clock. This is, of course, only one of several possible solutions for transferring data over a SPI bus. The SPI slave could be driven by an internal clock and poll its inputs based on known timing constraints. The FPGA could also act as a SPI master and use a poll-for-new-information type of protocol. These other approaches have their advantages, but the SPI slave clocked by an external master is a common enough pattern to warrant demonstration. The code required for capturing incoming information is actually quite simple. If the chip select is high (inactive) no data is taken into the module. If it is low, then data is taken into the array `spi_val` on every falling edge of the input clock until a full complement of `MSG_WIDTH` bits have been obtained. Note that incrementing `bit_cnt` and using it to control processing implicitly sets up a finite state machine around the processing. Once a full message has been obtained, a signalling bit (`got_data`) is toggled and the state machine is trapped in an idle state until the chip select is de-asserted.

```
SPI_RX_PROC: process(spi_clk, spi_cs) begin
  if spi_cs = '1' then -- Asynchronous reset      bit_cnt <= 0;  elsif
  falling_edge(spi_clk) then -- Read some data in  if bit_cnt < (MSG_WIDTH - 1) then      spi_val <= spi_mosi &
  spi_val((MSG_WIDTH - 1) downto 1); -- LSB encoded input      bit_cnt <= bit_cnt + 1;  elsif bit_cnt < MSG_WIDTH
  then      spi_val <= spi_mosi & spi_val((MSG_WIDTH - 1) downto 1); -- LSB encoded input      bit_cnt <= bit_cnt + 1;
  got_data <= not got_data;  else      bit_cnt <= MSG_WIDTH;  end if;  end if; end process SPI_RX_PROC;
```

In the above code, `spi_val` is an array of bits. Adding the latest incoming data into the top of the array implies that the incoming data should be formatted as least significant bit first. Sampling data on the falling edge of the SPI clock implies that the clock is active high and incoming data is sampled on the trailing clock edge, or that the clock is active low and incoming data is sampled on the leading clock edge. Trapping the `bit_cnt` at `MSG_WIDTH` until the chip select is taken inactive (high) implies that the chip select must be de-asserted between each transaction. It is perfectly possible to change any and all of these assumptions, or to parameterise them to obtain a more general SPI slave module. This has not been done here since the emphasis is on simplicity of example.

Crossing clock domains

The SPI bus implemented above is clocked by the external master, whereas processes internal to the FPGA will be clocked by some other clock. The external master clock may be at a different frequency, may be asynchronous to the FPGA's clock, and in many implementations is active only during the period when data is being transferred. Thus the incoming information from the SPI bus needs to cross from one clock domain to another before it can usefully be used within the FPGA's internal processes. To do this, we set up a clocked process which polls the state of `got_data` and compares it to a local copy to note changes. The problem here is that transferring data from an asynchronous clock domain can lead to sampling of metastable states. This is not space to go into a discussion of metastable states here, suffice to say that they result in corrupt data and are surprisingly easy to provoke given enough samples of the asynchronous data. Fortunately, there is a simple method to deal with metastable states - we simply sample the `got_data` signal on two successive states of the finite state machine. If it is consistent over the two states then we process the data, otherwise we ignore it.

```
SYNC_CMD_PROC: process(fpga_clk, got_data) begin
  if rising_edge(fpga_clk) then      case sync_state is
  when SYNC_IDLE =>      if (not(got_data = local_get_data)) then      sync_state <= SYNC_UPDATE;
  end if;      -- Second check for new message, and any required processing      when SYNC_UPDATE =>
  sync_state <= SYNC_IDLE; -- Always go to idle, regardless of whether we process data      if (not(got_data =
  local_get_data)) then -- Smooth out metastable states      local_get_data <= got_data;      -- Process data
  here      if spi_val = X&rdquo;01&rdquo; then      -- Do something !!      end if;      end if;  end
```

case; end if; end process SYNC_CMD_PROC;

In order to process the data, we could process it directly, read it into a local signal to process at our leisure, read it into a FIFO for some other module to process, etc.

Extending the code

The code presented in this article is intended as a simple example illustrating how to set up clocked processes, based on either implicit or explicit finite state machines, and how to transfer data between asynchronous clock domains. There are several extensions to the code that could be used to improve it. The polarity and phase of the SPI bus could be parametrised, as could the endianness of the incoming data. The SPI bus could be made to accept multiple transactions within a single chip select assertion. A data FIFO could be added to store incoming data. A command processor could be added to interpret the incoming data to control the FPGA's behaviour. Most obviously, the other half of the SPI bus (SPI_TX_PROC, active on the spi_clk rising edge) could be implemented to allow the FPGA to send information to the SPI master. These extensions are all fairly straight-forward to add and allow for tailoring to fit specific system requirements.