ENGS 31/ CoSc 56

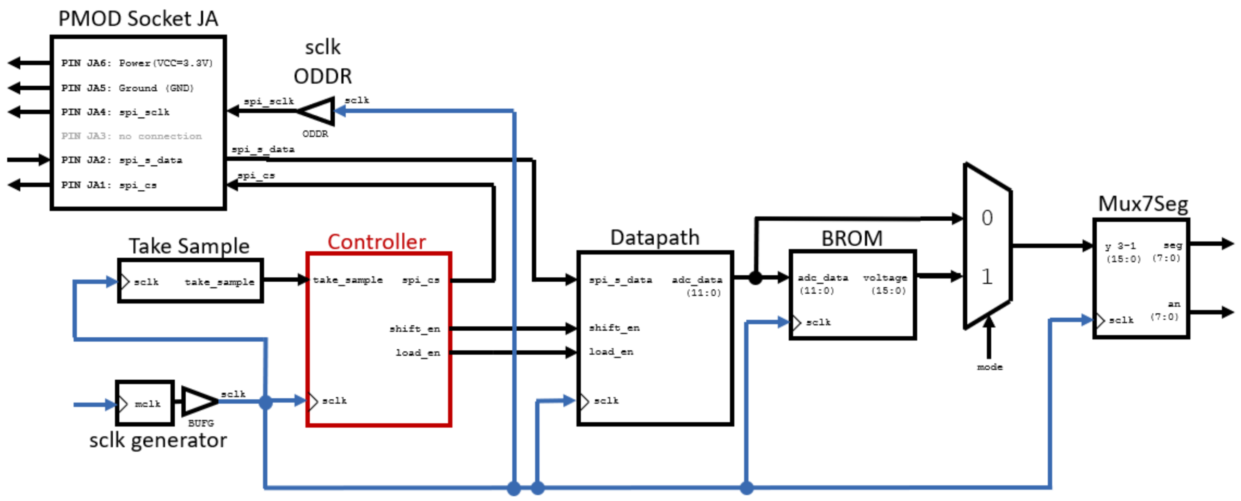
Lab 6: Serial Communication

The Pre-Lab Exercise for this Lab is due on Canvas at the start of your lab section
The Post-Lab write-up is due on Canvas by 11:59PM, May 12

Laboratory Objectives

1. Understand how the SPI bus works, implement a working SPI bus interface.
2. Observe and explain the behavior of an analog-to-digital converter.

Problem Statement



In this lab you will build a basic voltmeter, a device that measures and displays analog voltage. The design uses the Digilent Pmod AD1 analog-to-digital converter—a circuit board that you can plug into a Pmod socket on your Basys 3. The Pmod AD1 communicates with the FPGA via a Serial Peripheral Interface (SPI) bus. Processors (FPGAs, microprocessors) often use SPI to communicate with external devices like analog-to-digital and digital-to-analog converters (ADC and DAC, respectively).[1] The basic SPI bus definitions and timing were presented in class on Day 20 this week. The ADC chip on the Pmod AD1 is the ADCS7476. The datasheet for the ADCS7476 can be found on the assignment page, and the key details of the SPI interface and the operation of this device found in this datasheet are included in this handout.

The block diagram is shown above. The datapath and other supporting files (constraints, clock generation, take sample tick, seven-segment display driver, block read-only memory (BROM), and shell) are written for you. ***You do not need to write these blocks, though you should understand how they work.*** The lab reading gives an in-depth overview of each block. In the prelab, you will design the controller for the system, highlighted in red in the diagram above. In the lab, you will make some minor alterations to the shell to program the clock generator and tick generator by learning to use *generics*. ***As you read through the descriptions of the sub components, have the Vivado project open and read through the associated VHDL code.***

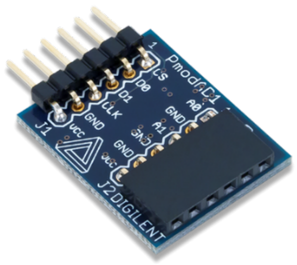
[1] For example, a list of SPI bus devices compatible with the Basys3 FPGA board may be found at http://store.digilentinc.com/by-communication-protocol/spi/.

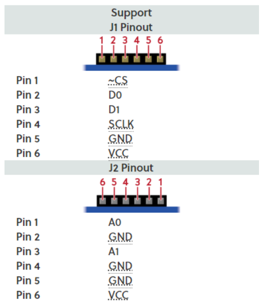
The Pmod AD1

In this lab you will be using two Pmod peripherals—the Pmod AD1, an analog-to-digital converter, and a pass-through board that will allow you to read the SPI-bus with the digital oscilloscope (logic analyzer): the Pmod TPH or the Pmod TPH2 (depending on what you received in lab). You can find all the information located in this handout at the resource center for the AD1, found here.

The resource center contains valuable information—sample code for the device, reference manuals, and schematics. The following images come from this source. The datasheet often includes vital information, like the electrical characteristics of the device, an application note, and the timing diagram required for using the device. You will look at some critical elements from the datasheets.

Pinout: The names of the pins are printed on the silk-screen (white lettering) of the Pmod AD1's. The male pins (the side that you plug into your Basys 3) are digital pins. The female pins are analog pins. Fig. 1 shows the Pmod AD1 board and the associated pinout.





Support

J1 Pinout

Pin 1	~CS
Pin 2	D0
Pin 3	D1
Pin 4	SCLK
Pin 5	GND
Pin 6	VCC

J2 Pinout

Pin 1	A0
Pin 2	GND
Pin 3	A1
Pin 4	GND
Pin 5	GND
Pin 6	VCC

Figure 1: The Pmod AD1 and associated pinout.

High Level Block Diagram:

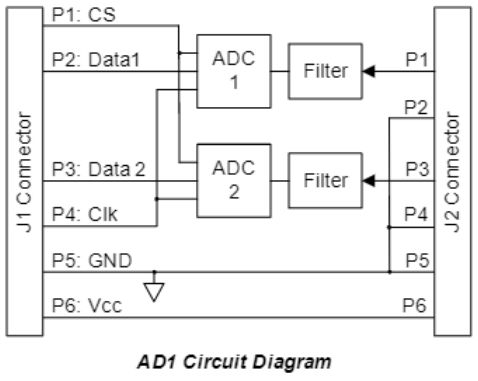


Figure 2: The Pmod AD1 block diagram.

The block diagram for the board is presented in Fig. 2. Notice that there are two analog-to-digital converter channels on this device. You will just be using ADC 1 in this lab, which maps analog input pin A0 to digital output pin D0. Nothing will be connected to the second channel, which maps analog input pin A1 to digital output pin D1. The chip select (CS in the diagram, spi_cs in our system) and the clock (Clk in the diagram and spi_sclk in our system) are shared between the two ADCs.

Analog to Digital Conversion and SPI Timing

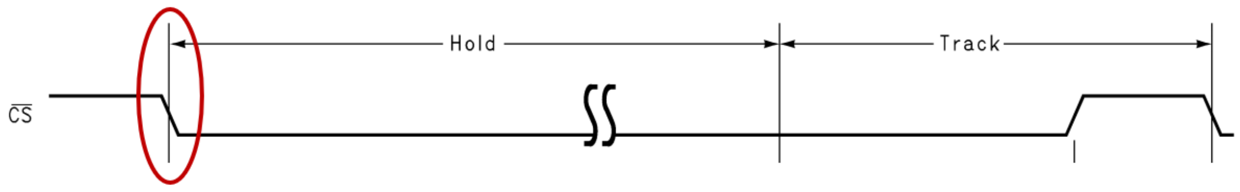


Figure 3: Chip Select timing diagram. Notice the falling edge of CS′.

The timing diagram for the low-true chip select (CS′) is presented in Fig. 3. Though CS′ is a 1-bit signal, it can send two distinct commands to the ADC by encoding the "take a sample" command in *an edge-triggered event* (the falling edge of CS′), and by encoding the "transmit the measured sample to the FPGA" command in *a level-triggered event* (CS′ is LOW).

When the chip select goes from HIGH to LOW (a falling edge is detected), the ADC takes a sample of the voltage on analog pin A0 (capturing a single datapoint) and digitizes it as a 12-bit number which is stored in a register. Each 12-bit number, which comprises values x"000" (decimal 0) to x"FFF" (decimal 4095), represents a unique "bin" (small range) of analog voltages. The measured voltage is rounded to the bin closest to the measured analog voltage and assigned the corresponding 12-bit number.

The quantization (rounding) process introduces an error dependent on the number of bits of the ADC. An 8-bit ADC has $2^8 = 256$ bins that a signal can be placed in, while a 12-bit ADC has 4096 bins. For a fixed input voltage range, the 12-bit ADC will have less quantization error than an 8-bit ADC because the size of each bin is smaller.

The input voltage range of the Pmod AD1's ADCS7476 is 3.3 V. The voltage bin size is, therefore, $\Delta V = 3.3\text{ V} / 4096 = 0.000806\text{ V} = 806\ \mu\text{V}$ (also known as "one least significant bit", or "1 LSB"). The voltage-to-binary conversion characteristic for the A/D is presented in Fig. 4. [2]

[2] National Semiconductor ADCS7476 data sheet (2003)

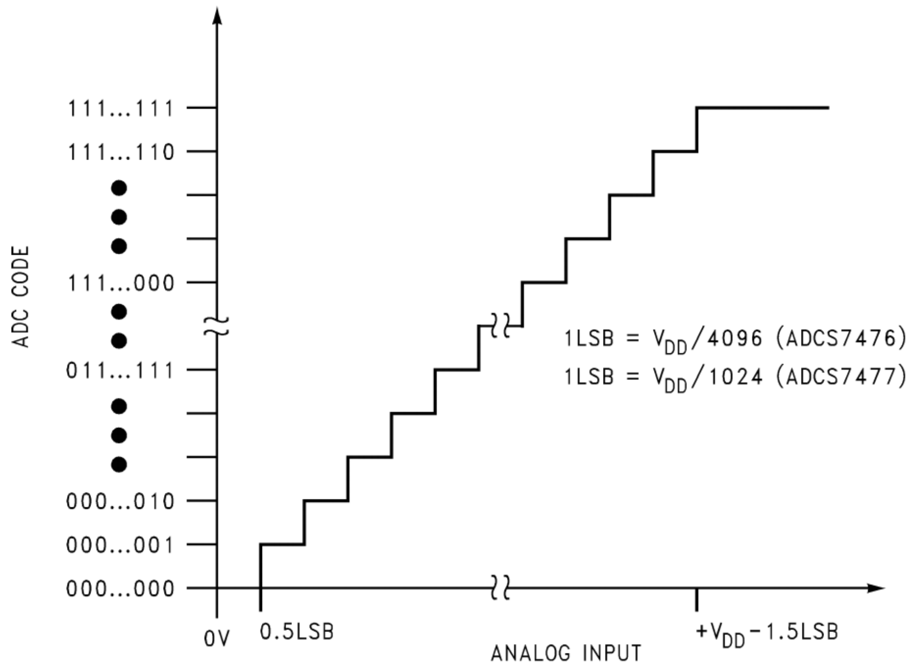


Figure 4: The mapping of measured analog voltages to digital sample bins.

Analog inputs between 0V and 0.5 LSB (403 μV) are assigned to the output bin x"000". The output bin is x"001" for inputs between 0.5 LSB and 1.5 LSB, etc. The output is x"FFF" for voltages at or above 3.3 V – 1.5 LSB (about 3.2988V).

When CS′ stays LOW after a falling-edge, the level-triggered command is sent to the ADC. The 12-bit sample is transferred serially (one bit at a time) on the *falling edge* (not rising edge!) of a serial clock called SCLK provided by the controlling device (in this case, our Basys 3 FPGA). A reciprocal shift register (receiver) reads in the bits as they are shifted out of the serial register (transmitter) in the Pmod AD1.

The timing diagram for the whole interface is presented in Fig. 5. Note that the ADC sends the data bits on the falling edge of the shared clock so the data can be read on the rising edge at the receiving device, enabling rapid data transfer. This behavior motivates the need for a 50% duty cycle on the clock—this duty cycle yields the maximum time for set-up and hold for both devices. The controlling device (your FPGA) sends the CS′ and SCLK signals to the ADC and receives the 1-bit line SDATA. Before transmission, the data is padded with three or four leading zeros. The data bits are contained in the last 12 bits (11 down to 0).

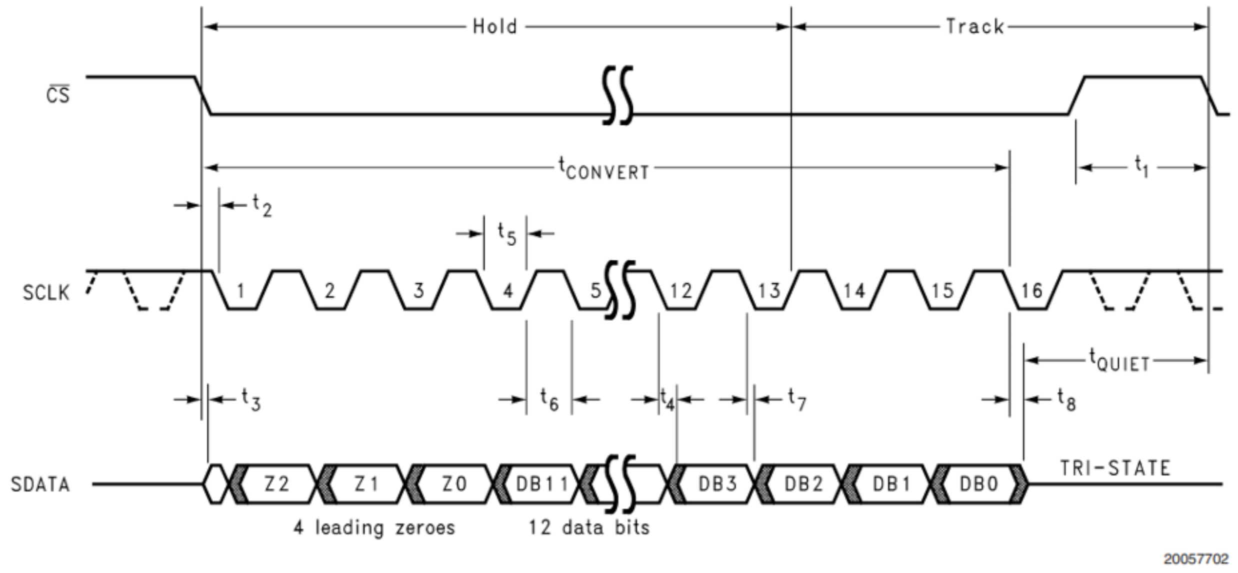


Figure 5: The complete timing diagram for the SPI interface.

Design and Operation of the SPI Receiver

Fig. 6 presents the system diagram for your voltmeter. The blocks highlighted in RED are blocks that you will design and implement in this lab. This section of the lab handout discusses key features of this design.

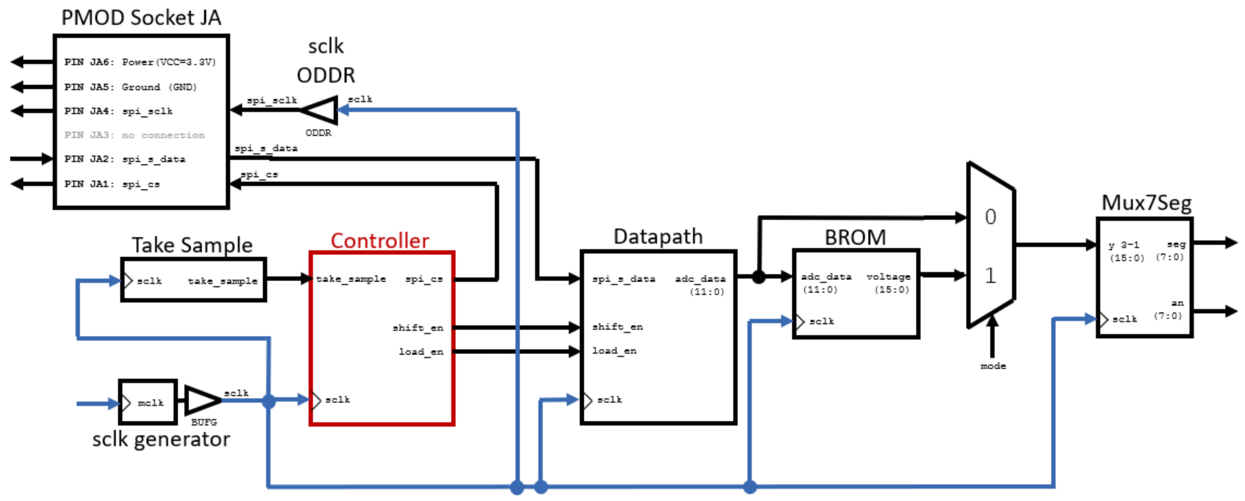


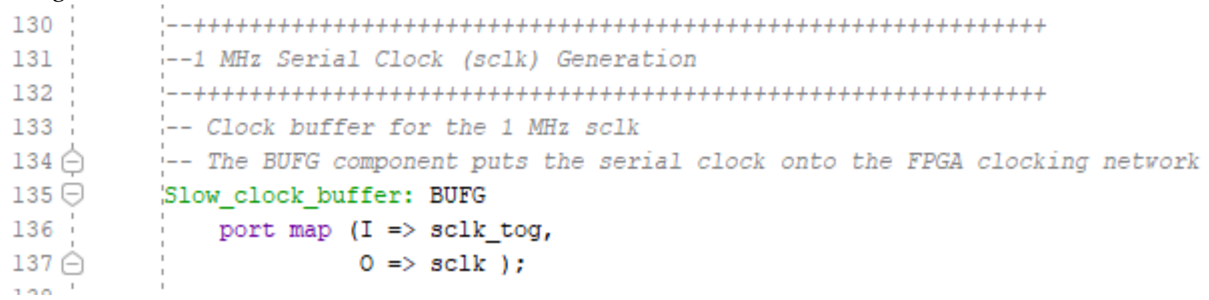
Figure 6: Block Diagram of the Lab 6 system.

System Timing

In Figure 6, wires that are black exist in the FPGA fabric (programmable logic). The wires highlighted in BLUE are on the clocking tree (an optimized highway for clock signals). Open the project shell file and examine the sclk divider within. At this point, this circuit should be familiar to you. The system takes in the 100 MHz `clk_ext_port` and uses a clock (frequency) divider circuit to generate the 1 MHz serial clock (`system_clk`, which is the same as `sclk` (serial clock) in the timing diagram) that will be used in the design. Notice that while `clk_ext_port` starts on the clocking tree, the frequency divider used to generate `sclk` exists in programmable logic, and, just like last week, you need to use a special buffer (BUFG) to port your serial clock signal onto the clocking tree. The clocking tree ensures the clock arrives at all the flip-flops at the same time, which enables the design to meet timing requirements and be well synchronized. Signals on the clocking tree can be used in `if rising_edge(clk) then` statements.

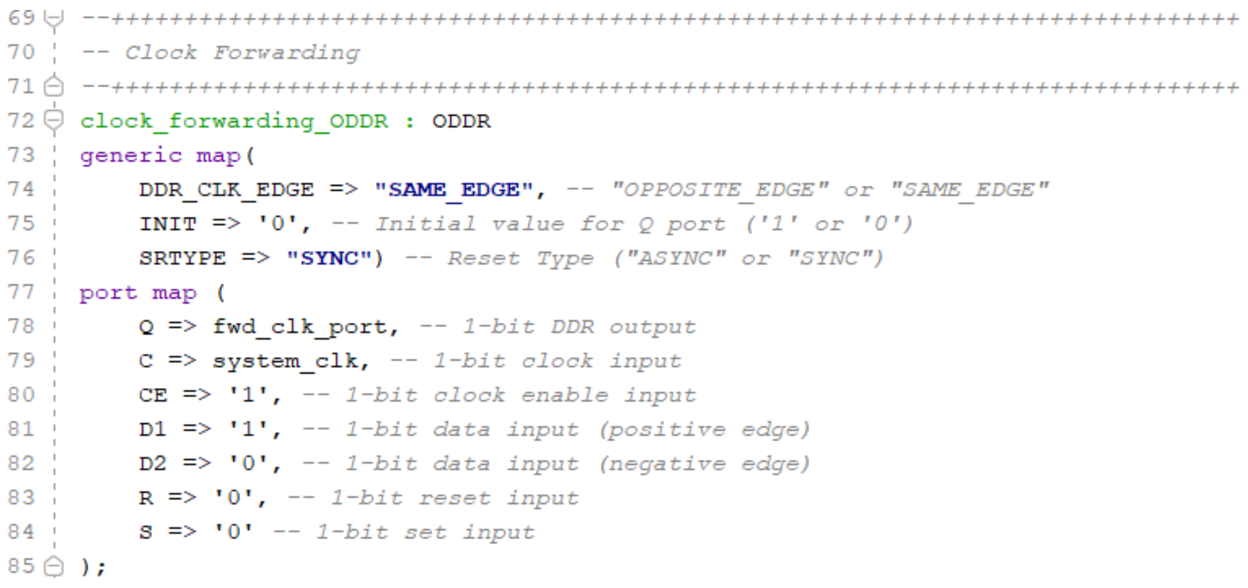
This week, you want to pass the serial clock on to the peripheral device (the Pmod AD1) to synchronize data transfer. This process is known as "clock forwarding." To reduce the delay from sclk to the output, you need to use a component called an ODDR (output double data rate) register, which is a special kind of flip-flop that exists right on the edge of the FPGA. The ODDR, like the BUFG, is given to you in the clock generation VHDL model. You can read more about the ODDR functionality and how to use it on page 127 of the Xilinx SelectIO Resources User Guide (UG741) located in the Lab 6 folder.

An excerpt from the manual is shown in Fig. 9. Configuring the ODDR is a little more complicated than the BUFG. Compare the instantiation of the BUFG shown in Fig. 7 with the instantiation of the ODDR shown in Fig. 8:



```
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--1 MHz Serial Clock (sclk) Generation
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- Clock buffer for the 1 MHz sclk
-- The BUFG component puts the serial clock onto the FPGA clocking network
Slow_clock_buffer: BUFG
    port map (I => sclk_tog,
              O => sclk );
```

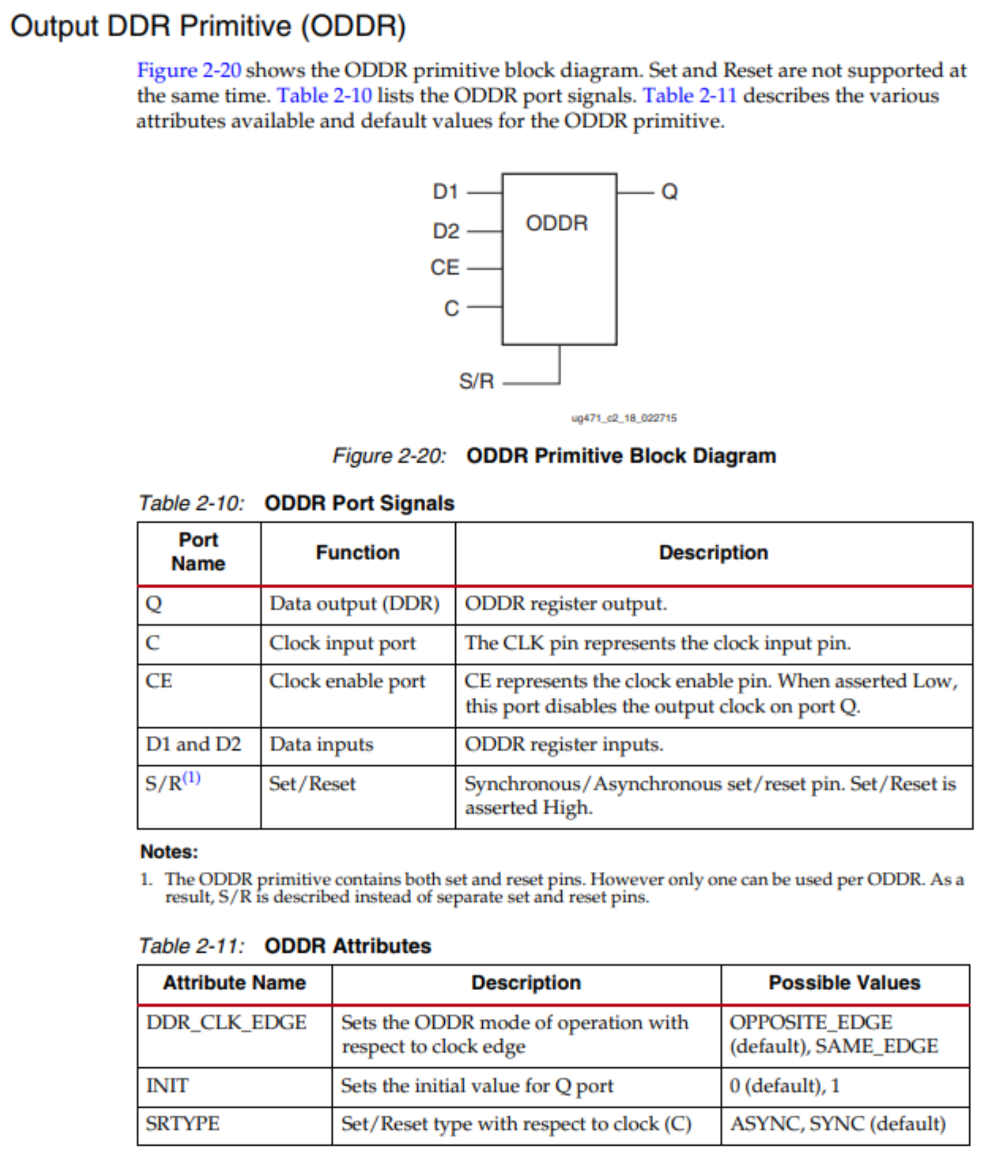
Figure 7: BUFG, a buffer which takes a signal from programmable logic and puts it on the clocking tree.



```
--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- Clock Forwarding
--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
clock_forwarding_ODDR : ODDR
generic map(
    DDR_CLK_EDGE => "SAME_EDGE", -- "OPPOSITE_EDGE" or "SAME_EDGE"
    INIT => '0', -- Initial value for Q port ('1' or '0')
    SRTYPE => "SYNC") -- Reset Type ("ASYNC" or "SYNC")
port map (
    Q => fwd_clk_port, -- 1-bit DDR output
    C => system_clk, -- 1-bit clock input
    CE => '1', -- 1-bit clock enable input
    D1 => '1', -- 1-bit data input (positive edge)
    D2 => '0', -- 1-bit data input (negative edge)
    R => '0', -- 1-bit reset input
    S => '0' -- 1-bit set input
);
```

Figure 8: ODDR, a special flip-flop which takes a signal from the clocking tree and safely passes it to a Pmod pin.

On line 78 of the code shown in Fig. 8, the forwarded clock (spi_sclk) that is routed to an external Pmod pin is defined as the output of this register, which is clocked by the system clock on line 79. The ODDR is enabled on line 80. Lines 81 and 82 respectively set the output of the register relative to the direction of the clock edge (flipping the bits here would output the peripheral clock on the falling edge of the internal clock, which can sometimes be advantageous). Lastly, on lines 83 and 84 you ensure that the set and reset pins of the ODDR are low, to enable normal behavior. The signal names used in the provided project are a little different, but the underlying form of the ODDR instantiation is the same.



Output DDR Primitive (ODDR)

Figure 2-20 shows the ODDR primitive block diagram. Set and Reset are not supported at the same time. Table 2-10 lists the ODDR port signals. Table 2-11 describes the various attributes available and default values for the ODDR primitive.

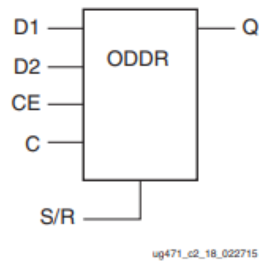


Figure 2-20: **ODDR Primitive Block Diagram**

Table 2-10: **ODDR Port Signals**

Port Name	Function	Description
Q	Data output (DDR)	ODDR register output.
C	Clock input port	The CLK pin represents the clock input pin.
CE	Clock enable port	CE represents the clock enable pin. When asserted Low, this port disables the output clock on port Q.
D1 and D2	Data inputs	ODDR register inputs.
S/R[1]	Set/Reset	Synchronous/Asynchronous set/reset pin. Set/Reset is asserted High.

Notes:

1. The ODDR primitive contains both set and reset pins. However only one can be used per ODDR. As a result, S/R is described instead of separate set and reset pins.

Table 2-11: **ODDR Attributes**

Attribute Name	Description	Possible Values
DDR_CLK_EDGE	Sets the ODDR mode of operation with respect to clock edge	OPPOSITE_EDGE (default), SAME_EDGE
INIT	Sets the initial value for Q port	0 (default), 1
SRTYPE	Set/Reset type with respect to clock (C)	ASYNC, SYNC (default)

Figure 9: Datasheet page regarding the ODDR.

Take Sample

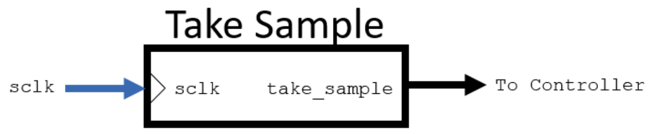


Figure 10: Take Sample functional block diagram (from Fig. 6).

You want to collect a new sample of the voltage measured by the ADC every tenth of a second (T = 100 ms, f = 10 Hz). The `take_sample` signal keeps time to let your system know when a tenth of a second has passed and it is time to collect a new sample. This signal is a "tick" which means that it is high for one clock cycle once a period—so in this case, `take_sample` should high for one cycle every 100 ms, and be low otherwise. This signal has a very low duty cycle. If you are having trouble visualizing what this signal should look like, draw it out on paper: high for 1μs, low for 99.999ms.

It is important to recognize that the `take_sample` tick drives no clock inputs on any flip-flops, and its rising edge is not an event-triggering condition for any logic. It is never passed onto the clocking tree and will never have a BUFG associated with it. It is, rather, a periodic input to the SPI bus controller that, when high, initiates acquisition of a data value from the A/D converter. This method of taking data at a regular interval is called "polling." In the future, if you use an A/D or D/A converter in your project, **be careful not to confuse the SPI bus serial clock or the system clock with the `take_sample` signal.**

It is also important to recognize that the `take_sample` signal **does not** directly tell the Pmod AD1 to take a sample—it is not connected to Pmod socket JA at all. Rather, it tells the controller when the SPI bus should wake up (every 100 ms) and generate a signal called the chip select (`spi_cs`) that **is** wired to Pmod socket JA, that tells the ADC to take and transmit a new sample (data point) when the Pmod AD1 detects that `spi_cs` has gone low.

This circuit is another variation on a frequency divider (counter that counts clock cycles). In contrast to the clock divider (count half the total number of outputting LOW, count half the total number of steps outputting HIGH, generating a 50% duty cycle), you do not have the same duty cycle constraint here that necessitates a toggle flip-flop (frequency division by 2)—you can divide directly to the frequency you want to generate.

Datapath and Controller

Datapath: the shift register element of the datapath is shown below. The serial data coming from the A/D converter, denoted spi_sdata, is *left*-shifted into a 12-bit shift register, called shift_reg, which is clocked by the 1 MHz clock, sclk, and enabled by a signal shift_en from the controller. The lower 12 bits (11 downto 0) of the shift register are parallel-loaded into the 12-bit output register, which is clocked by sclk and enabled by a signal load_en from the controller. The contents of the output register are called adc_data. Though the transmitted signal from the A/D converter is typically 15 or 16 bits (more on this below), the register needs only account for the 12 data bits, as the leading bits are always 0 and contain no information about the measured voltage itself.

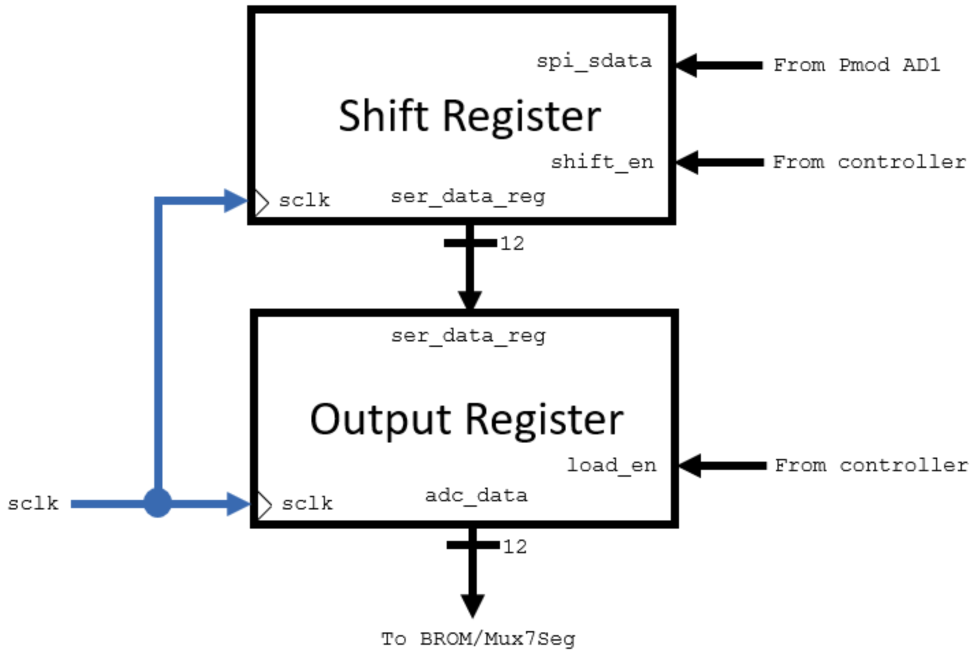


Figure 11: The core datapath that you will implement (taken from Fig. 6).

Controller: the controller is also clocked by the 1 MHz system_clk. Each time the controller sees a take_sample tick, it leaves its idle state and goes through a sequence of states, generating the SPI bus control signal spi_cs and datapath control signals shift_en and load_en in the appropriate sequence so that the serial A/D data is clocked into the shift register and then transferred to the output register. It then returns to its idle state to await the next take_sample tick.

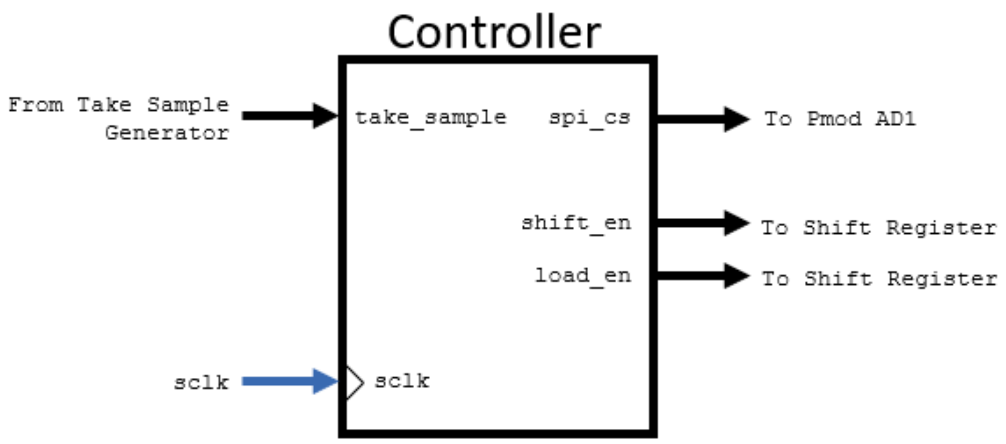


Figure 12: Controller functional block diagram (taken from Fig. 6).

Make sure that you understand the purpose and timing characteristics of each signal in the controller. Refer to the timing diagram for the A/D converter, Fig. 5 of the lab handout. Study the diagram along with the following description and make sure you understand it. The ability to interpret a timing diagram is an important digital design skill. If you do not understand the timing diagram, ask your TA, Ben, or Professor Luke about it.

- The CS′ signal (our spi_cs) is asserted low by the controller. It is convenient to assert CS′ on the rising edge of the serial clock, SCLK (spi_sclk). Although the diagram does not explicitly show this timing, it is permitted by the timing spec t2.
- The timing diagram shows that the first bit coming out is shorter than the rest. The data sheet explains, on pp. 17-18. Depending upon the application, the first edge on SCLK after CS′ goes low may be either a falling edge or a rising edge. If the first SCLK edge after CS′ goes low is a rising edge, all four leading zeroes will be valid on the first four falling edges of SCLK. If instead the first SCLK edge after CS′ goes low is a falling edge, the first leading zero may not be set up in time to be read correctly. The remaining data bits are still clocked out on the falling edges of SCLK. Since you will generate the CS′ signal from your controller state machine, which changes states on the rising clock edge, the CS′ signal will change shortly after the rising edge, and the first clock edge after CS′ goes low will be a falling edge. This is the second case described by the data sheet, in which the first zero bit will not be captured. ***As such there will only be 3 leading 0's (15 bits transferred).***
- On the first falling edge of SCLK after CS′ goes low, the first of the three preamble bits, Z2, is shifted out of the A/D onto the serial data line, SDATA. This bit will be shifted into your datapath shift register on the next rising edge of SCLK (labelled 1 in the figure). After two more preamble bits, Z1 and Z0, the actual data will begin to appear on the bus. The first data bit is DB11, the most significant bit. Because this bit appears first, your serial register must shift left and bring the spi_sdata stream in through the least significant bit.
- The last data bit, DB0, is shifted into the datapath shift register on the rising edge labelled 15 in the timing diagram. Finally, on the next rising edge (#16), the lower 12 bits of the serial data register are loaded into the output register and the CS′ signal is returned to high.

Block Read Only Memory (BROM) and 7-Segment Multiplexing

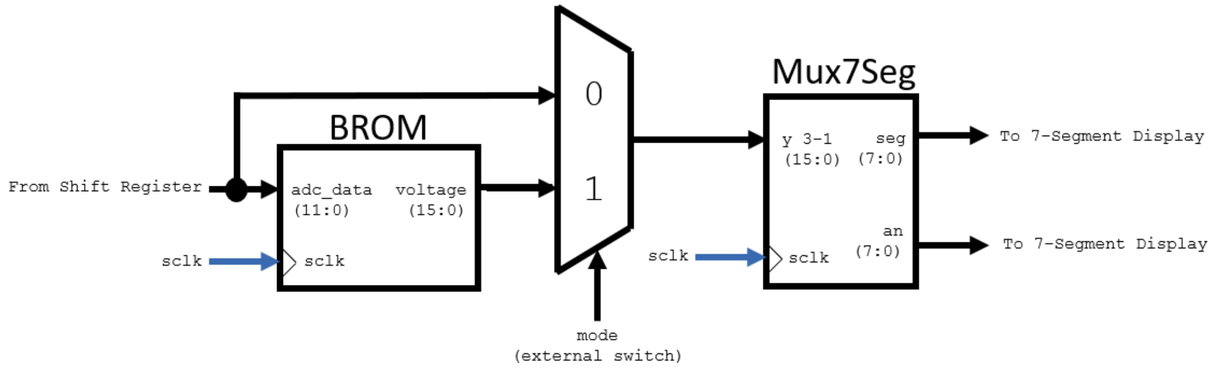


Figure 14: The rest of the datapath (taken from Fig. 6).

While you examined the shift register in the previous section, the shift register was not the totality of the datapath. This is built for you in the project, but it is important that you understand how it works, as this will make validating your design easier. The 12-bit sample recovered from the Pmod AD1 by your shift register is passed into an element known as a Block Read Only Memory (BROM).

Block memories are elements that are commonly used by digital designers—so common that Vivado gives us tools for generating them in the Intellectual Property (IP) Manager. If you are interested in learning how block memory can be generated in Vivado (you might want to use a BROM in your final project, for example!), look at slides 6-17 of the associated slide deck Memory 2 in Files > Laboratory Assignments > Lab6 on Canvas. These IPs are designed to be easy to use and implement. Instantiated IPs are specific to a Vivado project, which is why you provided a project for you in this lab—the block memory is already instantiated.

This block ROM acts as a look-up table that implements the inverse function of the ADC mapping seen in Fig. 4. While the Pmod AD1 measures an analog voltage and places it in one of 4096 digital bins, this look-up table takes the bin (output of our SPI receiver) as an address and sends a 16-bit binary coded decimal number (4 decimal digits) to the 7-segment display to read out the voltage. This mapping is stored in a coefficients file, known as a COE file. There is one coefficient for each address (possible bin). You can generate COE files with Matlab and a Matlab script to do so is in Files > Laboratory Assignments > Lab6 on Canvas. The COE file used in this project is also located in this folder.

Recall that the 12-bit output of the A/D converter goes from x"000" (0) to x"FFF" (4095). There are 4096 entries in the COE file, corresponding to the 4096 memory locations you use in the block ROM. The BCD representations of the actual voltages, from 0V to 3.3V, are stored in these memory locations. The ROM maps the A/D output value to the voltage represented by that value.

Following the graph in Fig. 4, the formula to convert from the A/D output value, adc_data, to the voltage stored in the ROM is $V = \text{round}(1000 \times \text{adc\_data} \times \Delta V)$. This produces a four-digit result, in millivolts (mV). Activating the decimal point between digits 3 and 2 will display these values in volts. For example, suppose that adc_data is x"1A7". In decimal, x"1A7" = 423, and $423 \times \Delta V = 423 \times 0.000806 = 0.340938$ V. Multiply by 1000 and round, the result is 341 mV. 341 is stored, in BCD (0000 0011 0100 0001 = x"0341") at location x"1A7" in the ROM. When the A/D outputs x"1A7", the 7-segment display shows 0.341 V.

Looking at the voltage and the hexadecimal value can be useful in validating the design, so an external switch (SW0/V17) is used to multiplex the input of the 7-segment display. When mode is HIGH the display shows the 4-digit analog voltage (your voltmeter). When mode is LOW the display shows the 3-digit hexadecimal value.

Before Coming to Lab

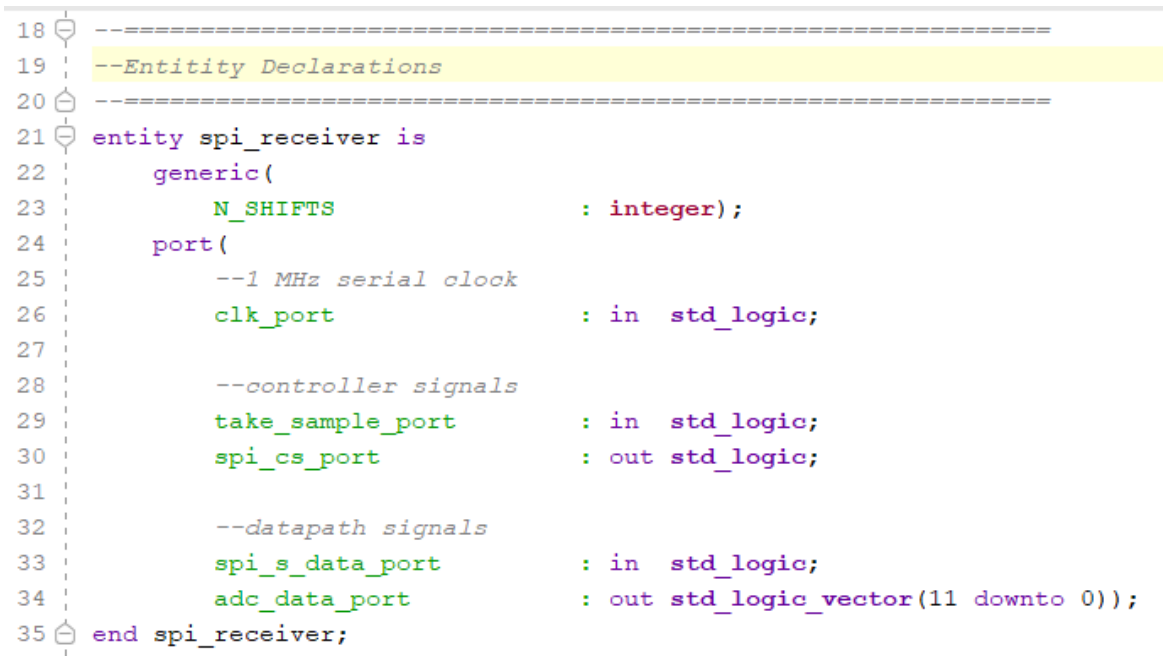
Make sure that you have read pages 1-12 of the lab handout.

Your task in this prelab assignment is to design the controller for the datapath on paper, then translate that design to a VHDL model. The `take_sample` signal is a sequence of pulses that are one clock cycle in duration, repeating every 100ms. This periodic signal alerts the controller to sequence the steps for a conversion, according to the A/D converter's timing diagram shown in the lab handout:

- Asserting the `spi_cs` signal to start the A/D conversion.
- Asserting `shift_en` to enable the shift register to capture the `spi_s_data` bits on the rising edges of `sclk`.
- After the required number of shifts, asserting `load_en` to transfer the contents of the shift register to the output register `adc_data`.
- When the conversion is complete, the controller returns to an idle state to await the next `take_sample` tick.

1. *State diagram*
Design a state diagram for the controller.

2. *VHDL model*
Write a VHDL model for the controller, including the state machine (two or three processes) and shift counter (one process). Use the template code provided in spi_receiver.vhd to get started. The spi receiver contains both the controller and the datapath, as the datapath is sufficiently simple. This means that the template file contains the shift register, so you will need to match signal and port names as needed. The entity of the spi receiver is shown below:



```
--==============================================================
--Entitity Declarations
--==============================================================
entity spi_receiver is
    generic(
        N_SHIFTS                : integer);
    port(
        --1 MHz serial clock
        clk_port                : in  std_logic;

        --controller signals
        take_sample_port        : in  std_logic;
        spi_cs_port             : out std_logic;

        --datapath signals
        spi_s_data_port         : in  std_logic;
        adc_data_port           : out std_logic_vector(11 downto 0));
end spi_receiver;
```

In Lab

For each deliverable, read through the appropriate procedure.

Deliverable 1: Design system timing (5 pts):

CLOCK_DIVIDER_RATIO (not tc): ________________________________

FREQUENCY_DIVIDER_RATIO (not tc): ________________________________

Deliverable 2: Simulation of the Controller and Datapath (10 pts).

Staff signature that x"abc" ends up in the `adc_data` register: ________________________________

Deliverable 3: Hardware Validation + Screenshot of one sample (15 pts).

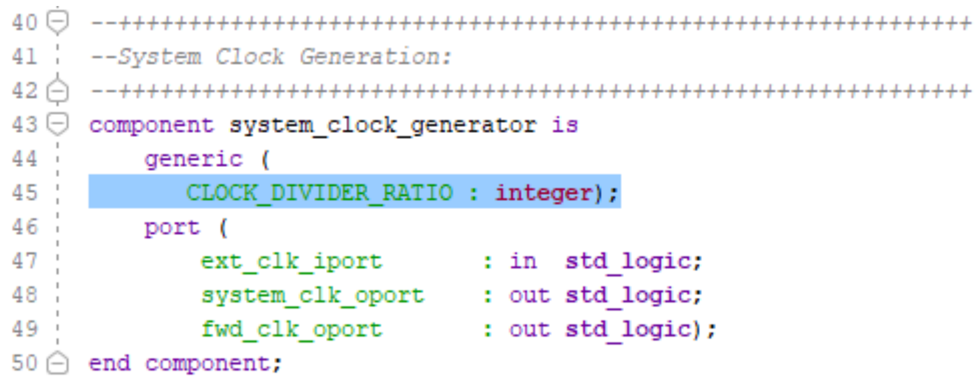
Staff signature that design is working correctly on the benchtop: ________________________________

Staff signature that the screen shot is correct: ________________________________

Deliverable 4: Final Draft of Paper Design + VHDL Model (20 pts).

Deliverable 1: Timing

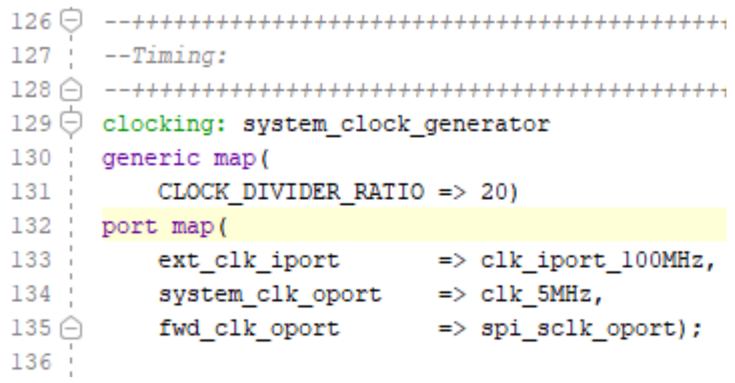
Open lab6_shell.vhd, system_clock_generator.vhd, and tick_generator.vhd and note the entity declaration of each. Notice that in addition to the port declaration, there is also a declaration for the component called a "generic."



```
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--System Clock Generation:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
component system_clock_generator is
    generic (
        CLOCK_DIVIDER_RATIO : integer);
    port (
        ext_clk_iport       : in  std_logic;
        system_clk_oport    : out std_logic;
        fwd_clk_oport       : out std_logic);
end component;
```

Generics let you pass a constant into the block, which you will use to set terminal counts. Along with the port map, you can specify a "generic map" when you instantiate the component into your design. Using generics can make modular blocks easier to recycle from project to project. Instead of editing the architecture of a component to change the value of a parameter, you can specify the parameter value at the top level and pass it into the component.

Suppose you wanted to clock the design at 5 MHz. You would use this generic/port map in this project to do so:



```
--++++++++++++++++++++++++++++++++++++++++++
--Timing:
--++++++++++++++++++++++++++++++++++++++++++
clocking: system_clock_generator
generic map(
    CLOCK_DIVIDER_RATIO => 20)
port map(
    ext_clk_iport       => clk_iport_100MHz,
    system_clk_oport    => clk_5MHz,
    fwd_clk_oport       => spi_sclk_oport);

```

This would divide down the 100 MHz clock by a factor of 20. Notice that here you are specifying the clock divider RATIO, not the actual terminal count for the counter. The system clock generator creates the terminal count internally by dividing the ratio value specified by 2.

Calculate the clock divider ratio needed for a 1 MHz clock and a 10 Hz take_sample (polling) signal and enter them into the appropriate locations in the generic maps in the shell.

Deliverable 2: Simulation

Open the testbench spi_receiver_tb.vhd and read through the file. You will need to update the generics in this testbench. This testbench is different than many testbenches you have previously written and used, in that it is waiting from inputs from the FPGA in the stimulus process and will not run the process until spi_cs goes LOW. Until this event occurs, the spi_s_data line will remain undefined—so depending on when your take_sample signal occurs, you may need to run the simulation for a while to verify that your design is working properly.

This testbench pretends to be the ADC and implements the complementary shift register that listens to the SPI bus you have created. A data sample, 1010 1011 1100, or ABC, has been collected by the ADC and stored in the internal register TxData. This register is shifted 1 bit at a time when spi_cs is LOW. If your shift register is working properly, you should recover ABC in your adc_data register. Below is a screenshot of what the testbench should look like if it runs successfully. Notice that SPI_CS is low for exactly 15 clock cycles.

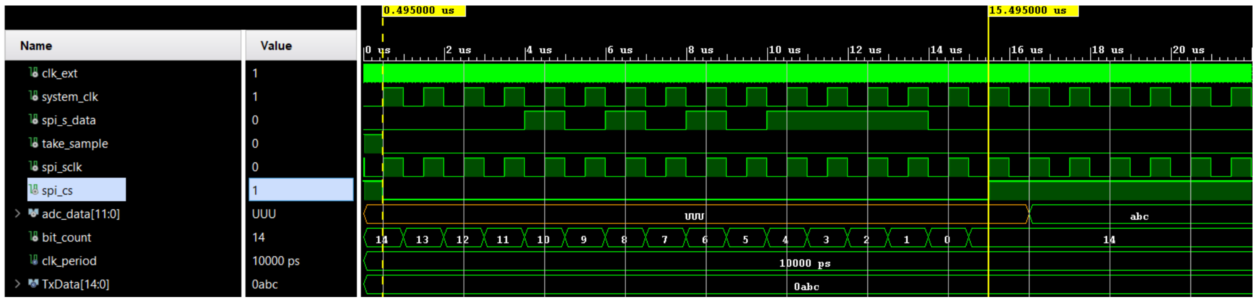


Figure 13: x"abc" being transmitted into the adc_data register over a SPI bus.

Show a member of the lab staff a screenshot of your testbench waveforms. In your submission for the lab, include a well annotated figure of your simulation. Note that you need to run the simulation long enough to get a take_sample tick to come along!

Deliverable 3: Design Validation

At your lab bench you will have also received a test-point header (TPH) Pmod board, either the Pmod TPH or a Pmod TPH2. Test-point headers are connection points for logic analyzer pins. These circuit boards will perform the same function for you in this lab—the TPH2 just has twice the pins. The resource center for these two devices can be found at:

Pmod TPH: https://reference.digilentinc.com/reference/pmod/pmodtph/start
Pmod TPH2: https://reference.digilentinc.com/reference/pmod/pmodtph2/start

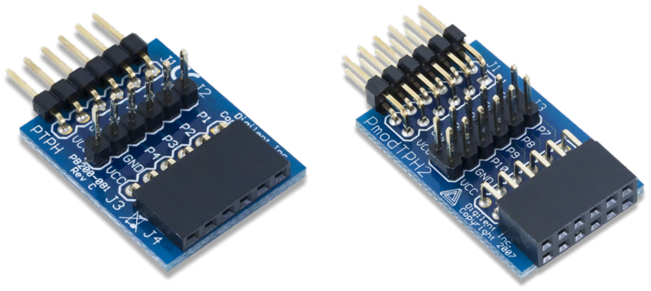


Figure 15: Pmod test point headers.

In the picture of the boards above, the Pmod TPH is on the left and the Pmod TPH2 is on the right. These boards are inserted between two circuits that want to talk to each other (like our ADC and our FPGA) and offer a tap into the wired connection between the circuits. You can attach logic analyzer leads to the pins sticking up to "sniff" the bus—the logic analyzer measures the signals of the bus as they travel between components. You will be inserting this device between the Pmod AD1's digital side and the Basys 3 Pmod socket to look at the signals on the SPI bus, like the clock, the chip select, and the serial data line. In the circuit schematics below (TPH left, TPH2 right), you can see that the PMOD merely offers a tap into the data lines:

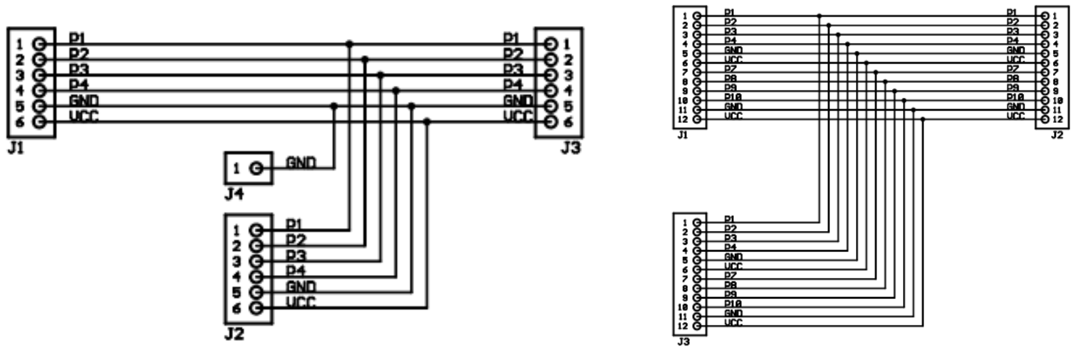


Figure 16: Schematics for the Pmod test point headers.

Test-point headers are incredibly useful tools for debugging and validating designs.

Generate the bitstream. To validate the design, plug the test point header Pmod TPH/2 into the JA socket of the Basys3, and the digital side of the Pmod AD1 into the Pmod TPH/2. Connect the digital scope to the data lines using the Pmod TPH/2. Wire DIO 0, 1, and 2 to the specified pins (spi_cs (JA1), spi_sdata (JA2), and spi_sclk (JA4) respectively) and remember to connect the GND. Wire the function generator signal and GND to pins A0 and GND. Double check that the function generator is in High-Z mode. Consult a member of the Lab Staff before energizing the system to ensure that everything is wired correctly.

Start with the DC voltage at 0 V. Slowly ramp up the voltage. Make sure that the DC voltage does not exceed 3.4 V as you test your design by varying the voltage. You can switch the 7-segment display between displaying the voltage and the hex bin by flipping sw0. Test a few points and see if the voltage reading on the 7-segment display matches the voltage you are inputting. **Set the voltage to 1 V and take a screenshot of the waveforms on the 7-segment display.** Trigger the device on JB1.

Congratulations! You have finished Lab 6 and have built a digital voltmeter. Now onto the project!