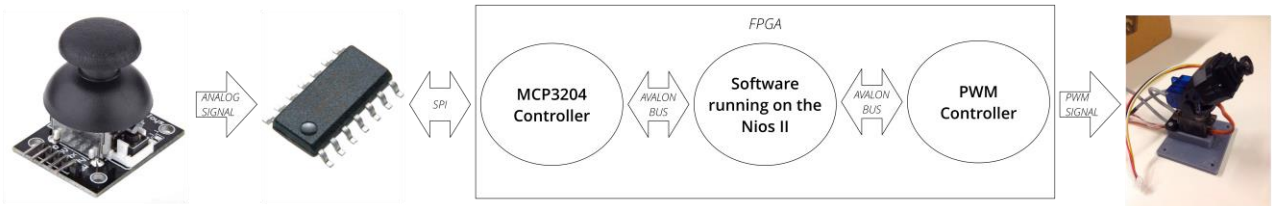
Lab 1.2: ADC Interface

Hardware

Recall the goal of the *Lab 1* series: to control a pan-tilt module with a joystick. The figure below shows the overall hardware system we have built for this project.



Prior work

Up until now, you have designed the hardware and software components needed to generate and control the PWM signal needed by the pan-tilt module.

- Lab 1.0: You wrote the software for a pre-compiled PWM generator that we provided you.
- Lab 1.1: You designed your own hardware module for the PWM generator.

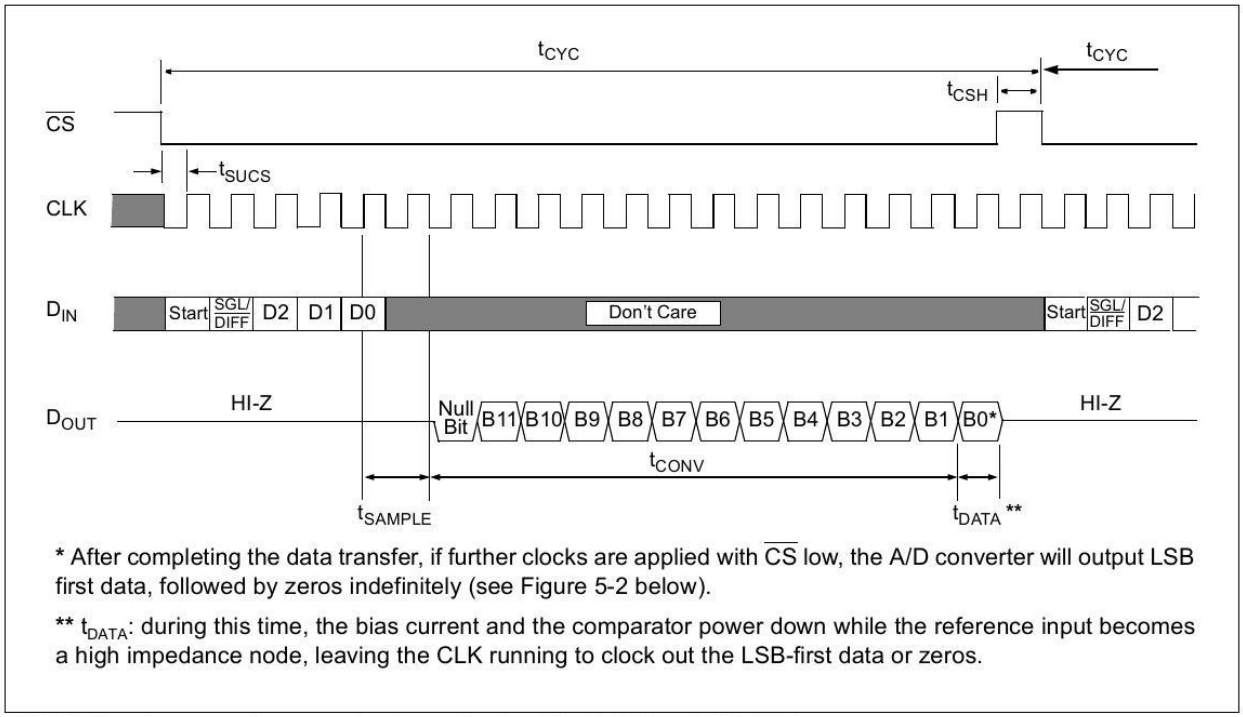
The pan-tilt part of the project is now complete, and we can move on towards the next topic.

What needs to be done

We now need to handle the joystick part of the design. The joystick we use in this lab has 2 ***analog*** outputs, one for each axis (horizontal & vertical). We need to convert these analog signals into ***digital*** values before we can do anything with them, so we need to use an *Analog to Digital Converter (ADC)*.

The ADC used in our design is the MCP3204. This ADC can take up to 4 ***analog*** input channels and provides the converted ***digital*** values to a host system through a standard communication interface. The MCP3204 uses the *Serial Peripheral Interface (SPI)* bus for commands and data.

Figure 5-1 below (c.f. MCP3204 datasheet) shows the communication between the ADC (SPI *slave*) and the host system (SPI *master*).



* After completing the data transfer, if further clocks are applied with $\overline{CS}$ low, the A/D converter will output LSB first data, followed by zeros indefinitely (see Figure 5-2 below).

** t_{DATA}: during this time, the bias current and the comparator power down while the reference input becomes a high impedance node, leaving the CLK running to clock out the LSB-first data or zeros.

FIGURE 5-1: *Communication with the MCP3204 or MCP3208.*

To obtain the digital value corresponding to one of the analog channels, you need to send a command to the device. Table 5-1 below (c.f. MCP3204 datasheet) shows the configuration bits for the MCP3204.

TABLE 5-1: CONFIGURATION BITS FOR THE MCP3204

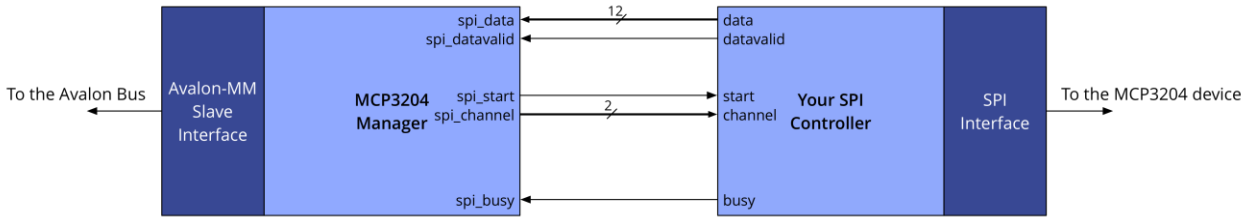
Control Bit Selections				Input Configuration	Channel Selection
Single/Diff	D2*	D1	D0		
1	X	0	0	single-ended	CH0
1	X	0	1	single-ended	CH1
1	X	1	0	single-ended	CH2
1	X	1	1	single-ended	CH3
0	X	0	0	differential	CH0 = IN+ CH1 = IN-
0	X	0	1	differential	CH0 = IN- CH1 = IN+
0	X	1	0	differential	CH2 = IN+ CH3 = IN-
0	X	1	1	differential	CH2 = IN- CH3 = IN+

* D2 is a “don’t care” for MCP3204

Controlling the ADC

We need to design a VHDL module that can supply commands to the ADC, and that can read the converted values back from the SPI bus. The digital values must then be written in 4 registers so a host processor can read them through the Avalon bus.

The figure below shows the structure of the MCP3204 VHDL module that we have designed for this.



The module consists of 2 parts:

- *MCP3204 Manager*: this unit interfaces with the host processor through an *Avalon-MM Slave* interface and provides 4 32-bit registers that hold the resulting digital values returned by the ADC. The unit is also responsible for supplying commands to a custom SPI controller that you will have to implement. The *Manager* is clocked at ***50 MHz***.

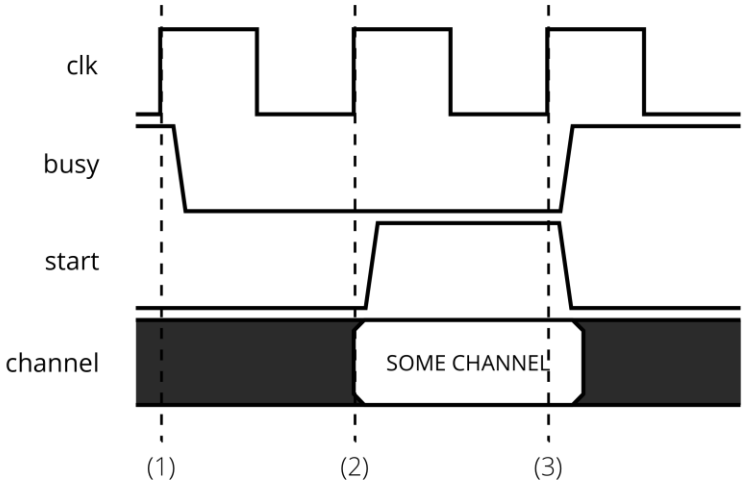
- *SPI controller*: this unit is responsible for performing the actual SPI communication to retrieve the converted values from the ADC. SPI communication needs to be done at ***1 MHz***.

Your task: designing the SPI controller

The *SPI Controller* works in two phases:

1. The *Manager* instructs it which channel to convert.
2. The SPI communication is performed and the converted data is sent back to the *Manager*.

Phase 1

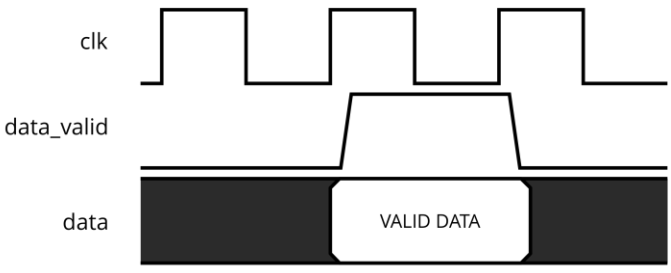


As long as the *SPI Controller* isn't ready, i.e. is busy converting, it asserts ***busy***. As soon as it finishes, it deasserts ***busy*** (see point 1 in the figure above).

The *Manager* waits for the *SPI Controller* to be ready. The *SPI Controller*'s 2-bit ***channel*** input is used by the *Manager* to instruct which channel must be requested from the ADC. Therefore, when the *Manager* detects that the *SPI Controller* is ready, it sets the appropriate ***channel*** and asserts the ***start*** strobe (see point 2 in the figure above).

Finally, the *SPI Controller* captures the new instruction and becomes busy again since it starts converting the appropriate channel (see point 3 in the figure above).

Phase 2



Once your *SPI Controller* has successfully converted a value. It passes it to the manager simply by asserting ***data_valid*** and putting the value on the ***data*** bus during one cycle.

Methods

We suggest that you follow the steps below:

- We provide the clock divider used for the SPI communication (and all associated signals) in *mcp3204_spi.*vhd. We suggest you read and make sense of the existing design before continuing.
- Draw the finite state machine (FSM) of your SPI controller on PAPER.
- Implement your FSM in the *STATE_LOGIC* process in *mcp3204_spi.vhd*. Using ModelSim is a non-negligible advantage. We provide the stimulus generation file *tb_mcp3204_spi.vhd* to generate instructions for the controller.

Pay attention to the fact that some signals need to be sent out on the ***falling*** edge of the *SPI clock (SCLK)*, while others on the ***rising*** edge of *SCLK*. This can lead to difficult to debug problems! Use the *reg_rising_edge_sclk* and *reg_falling_edge_sclk* flags provided in *mcp3204_spi.vhd* to detect the correct edge!

Software

The *Avalon-MM Slave* interface to the *MCP3204 Manager* is simple. There are no control registers to start/stop the unit. The only registers available are the 4 32-bit data registers that contain the converted digital values. There are 4 data registers available to support two joysticks since each joystick exports two analog signal (one for the X coordinate and one for the Y coordinate).

In *mcp3204.c*, complete the *mcp3204_read()* function. This function takes the channel number as input and returns the content of the appropriate register.

Note that the ADC returns 12-bit values for each channel, so the values returned by the function are bounded to [0..4095].

Write a short *while* loop where you continuously read and print out the values of the various registers. Move the joystick to verify that the values are coherent. (Be sure to print with a certain granularity, otherwise you will flood the Nios II console, which is already slow enough for printing without a flood).

Finally, once this works, you might interpolate those values to control the pan-tilt module. You will have to copy your *pwm.c* and *pwm.vhd* from previous lab sessions and recompile the current design.

Testing

Last week, you earned the title of ***soldering iron warrior*** by soldering an MCP3204 to a small PCB. We will now your work of art to test your implementation (obviously, only if it works in simulation first. You would be surprised how many people ignore this advice ...).

Connect the system appropriately, i.e joystick -> MCP3204 PCB -> DE0-Nano-SoC. The table below shows the GPIO pins that correspond to the various SPI signals.

SPI pin	DE0-Nano-SoC pin
CS_N	GPIO_0(2)
MOSI	GPIO_0(3)
MISO	GPIO_0(4)
SCLK	GPIO_0(5)

The figure below (c.f. MCP3204 datasheet) shows the pins on the MCP3204.