# TP SPI


## Elio KHAZAAL – Abel DIDOUH

## ESIEE Paris – Université Gustave Eiffel


*October 24, 2023*

## Lab Objective

The objective of this lab is to communicate via SPI with the analog-to-digital converters on the PmodAD1 module developed by Digilent, which can be directly connected to the Pmod connectors on their Basys3 boards.

## Overview of the PmodAD1

The Digilent PmodAD1 is a two-channel 12-bit analog-to-digital converter (ADC) that incorporates the Analog Devices AD7476A circuit. It has a sampling rate of up to one million samples per second.

The PmodAD1 communicates with the host board via a communication protocol similar to SPI. The difference between the standard SPI protocol and this protocol is evident in the pin layout of this Pmod. A typical SPI interface would expect a Chip Select pin, a Master-Out-Slave-In pin, a Master-In-Slave-Out pin, and a serial clock signal. However, with the two ADCs on this chip, both data lines (MOSI and MISO) are designed to function solely as outputs, effectively turning them both into MISO lines.

The PmodAD1 provides its 12 bits of information to the system board in 16 clock cycles, with the first four bits consisting of four initial zeros, and the remaining 12 bits representing the actual data with the most significant bit first. The first initial zero is transmitted at the falling edge of the CS (Chip Select) signal, with all subsequent bits transmitted at the falling edge of the serial clock signal.

# 1. Architecture and VHDL

1) In this part, we will explain the RTL structure and state machine that we have designed in order to communicate with the ADCs on the PmodAD1 using SPI communication. In order to do so, we need to reference our design on the Serial Interface Timing Diagram given in the AD7476A's datasheet as shown in Figure 1.

## A. AD7476A Serial Interface Timing Diagram Explanation

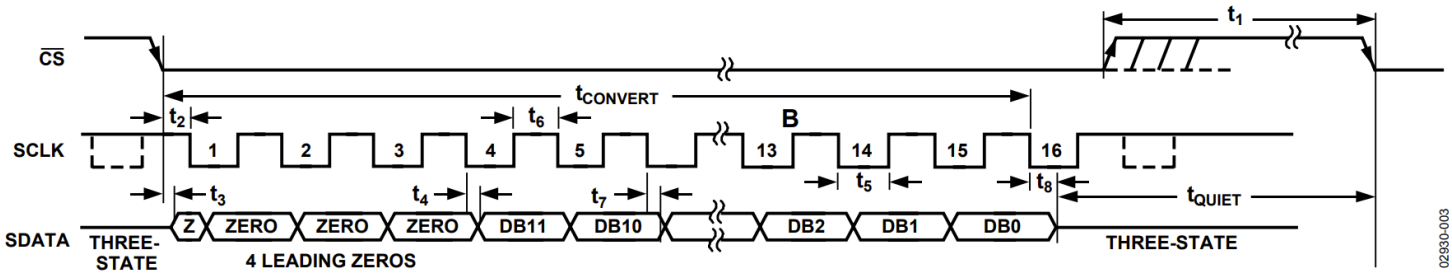



*Figure 1: Serial Interface Timing Diagram*

- **CS:** The CS signal in SPI is a control signal used to select and activate a specific slave device on the SPI bus. When the CS signal is asserted, it indicates to the selected slave device that it should pay attention to the incoming data and clock signals on the SPI bus. When the CS signal is deasserted, it signals the end of communication with that particular slave device.

  From the timing diagram in Figure 1, we can see that the CS is an active Low signal, so it is initially deasserted (set to 1) and then gets asserted (set to 0) to signal the start of conversion. It should stay asserted until the end of transmission.

  When the transmission is done, the CS gets deasserted, it should stay deasserted for a minimum duration of t1 = 10 ns before starting the next transmission.

- **SCLK:** The SCLK signal in SPI is a synchronized clock signal that controls the timing of data transmission between the master and slave devices on the SPI bus. Both the master and the slave devices use the rising and falling edges of the SCLK signal to sample and synchronize the data being sent or received.

  From the timing diagram in Figure 1, we can see that SCLK is initially set to 1, and then starts clocking after a minimum duration of t2 = 10 ns of when CS is asserted. After 16 rising edges, CS will be deasserted and SCLK will be set back to 1, signalling the end of

transmission. SCLK provides the serial clock for accessing data from the part. This clock input is also used as the clock source for the conversion process of AD7476A.

- **SDATA:** The SDATA signal in SPI the actual data being transmitted between the master and slave devices on the SPI bus. It shifts data bits in a serial fashion, typically one bit at a time, synchronized by the SCLK signal's rising or falling edges.

  From the timing diagram in Figure 1, we can see that the conversion result from AD7476A is provided on this output as a serial data stream. The first zero is clocked out on the falling edge of CS and the rest of the bits are clocked out on the falling edge of the SCLK input. The data stream from the AD7476A consists of four leading zeros followed by 12 bits of conversion data that are provided MSB first.

  The data will be retrieved on the FPGA on the rising edge of SCLK because we have to take into consideration the delay introduced by the transmission line giving the FPGA a window during which the data should be stable around the clock edge to be reliably captured.

## B. RTL structure

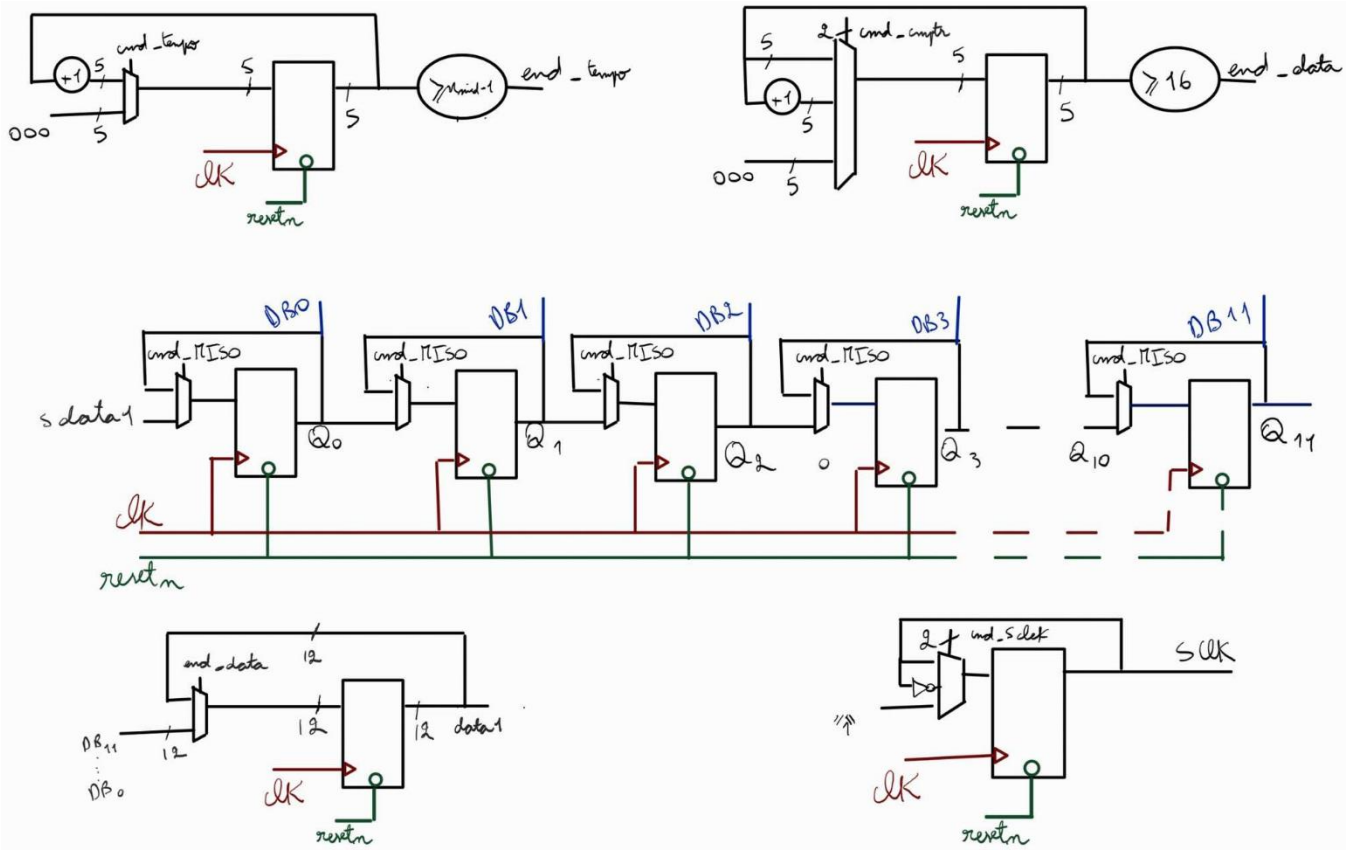



*Figure 2: System RTL Structure*

In the following Figure, we will show the complete RTL structure of our project containing the deserializer, the Sclk generator, the temporization counter, the data counter and the data saver, and then we will explain in detail each of these components.

## ❖ Deserializer

FPGAs are often designed to process data in parallel, meaning they work with multiple bits of data simultaneously. SPI is inherently a serial protocol, which means it transmits data one bit at a time. A deserializer must be used to convert the serial data from the ADC into a parallel format as shown in Figure 3. We will use 12 flip-flops in order to realize the deserializer because we have to read 12 bits of data without reading the first 4 zeroes sent by the ADC at the beginning of the transmission. This means that we have to count 16 clock edges before being able to read the data on Q0 to Q11. It is important to note that we need to implement 2 deserializers since we have 2 ADCs on the PmodAD1

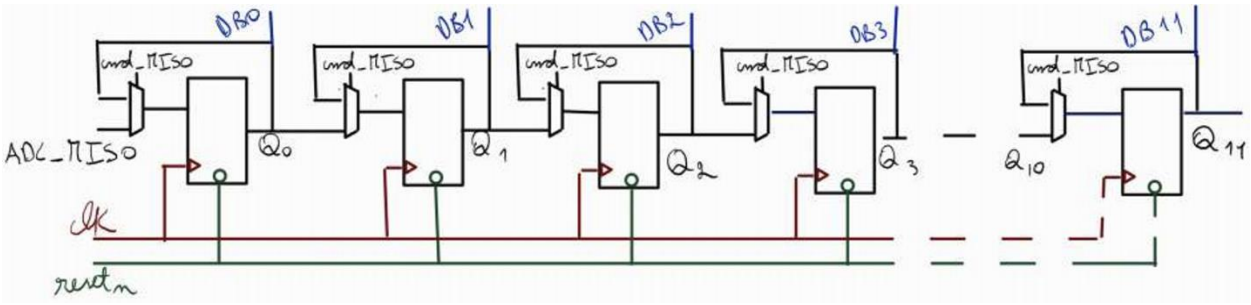



*Figure 3: Deserializer*

The Deserializer that we decided to use needs to have 2 functions:

- The first function is shifting the data to the left since we are receiving the MSB first and because we need to shift the data on each rising edge of SCLK until we can read the result of the transmission on the outputs of the flipflops Q0 to Q11 after 16 clock cycles.

- The second function is memorization because the SPI operates at the frequency SCLK which is lower than the frequency of clk which means that we have to memorize the values of the outputs of the flipflops until the next rising edge of SCLK arrives.

In order to implement both functions, we need to use a multiplexer in order to switch between the first function and the second function. This multiplexer will be controlled by the command ‘’cmd_MISO‘’ on 1 bit since we only have 2 possibilities.

## ❖ SCLK Generator

In order for the PmodAD1 to function properly, we need to provide it with an SCLK so that the ADC can position its data on the falling edges of SCLK. Hence, we created the RTL diagram of the SCLK generator as we can see in Figure 4.

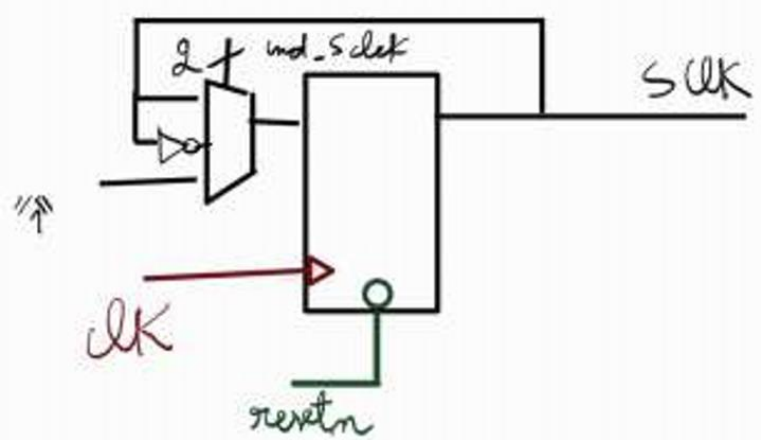



*Figure 4: SCLK Generator*

This generator has 3 functions:

- The first function is setting SCLK to ''1'' since in the timing diagram, before CS is asserted and after the transmission is done, SCLK needs to be set to 1.

- The second function is to toggle the output using the NOT inverter every half period of SCLK in order to create our serial clock.

- The third function is memorization because the SPI operates at the frequency SCLK which is lower than the frequency of clk which means that we have to memorize the output ''SCLK'' for half a period of SCLK before toggling it again.

In order to implement the three functions, we need to use a multiplexer in order to switch between the first, second and third function. This multiplexer will be controlled by the command ''cmd_Sclk'' on 2 bits since we have 3 possibilities.

## ❖ Temporization Counter

A temporization counter is a digital electronic component used to generate timing signals or delays in a circuit. It counts clock cycles and produces an output or action when a predetermined count is reached.

As we have mentioned before, we need to toggle the output of our Sclk generator every half period of Sclk in order to create the serial clock. Therefore, we will need the temporization counter shown in Figure 5 to indicate that half a period had passed.

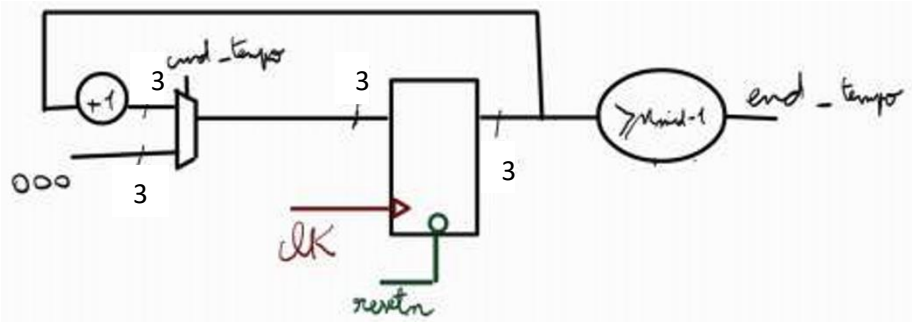



*Figure 5: Temporization Counter*

The ratio between the period of Sclk and clk is represented by the variable $x$ as follows: $T_{Spi} = x.T_{clk}$ . This means that for each $x$ periods of clk, we will get one period of Sclk. Since we want to toggle Sclk every half period, we will compare the counter value to $x_{mid} - 1$ with $x_{mid} = \frac{x}{2}$. When the value of the counter exceeds the value of $x_{mid} - 1$, the value of the output ''end_tempo'' will be set to 1, indicating that half a period of Sclk has passed and resetting the counter back to 0. The output ''end_tempo'''will be used to time the commands ''cmd_MISO'', ''cmd_Sclk'', ''cmd_tempo'' and ''cmd_ctr'' that we will see in the following paragraph. The number of flip flops required to achieve this circuit is $n = \log_2(x_{mid})$. In our case, we will set $f_{spi} = 10\ MHz\ and\ f_{clk} = 100\ MHz$ which will give us $x_{mid} = 5$ and $n = 3$ flip flops after rounding up.

The temporization counter has 2 functions:

- The first function is incrementing the counter with each clk rising edge.
- The second function is to reset the counter to 0 each time end tempo passes to 1.

In order to implement the two functions, we need to use a multiplexer in order to switch between the first and second functions. This multiplexer will be controlled by the command ''cmd_tempo'' on 1 bit since we have 2 possibilities.

## ❖ Data Counter

We need a data counter to count the bits that are being transmitted so that we know when the end of transmission is in order to pull sclk and CS back to 1 and reinitialize the system to prepare it for the next transmission. The data counter is shown in the following Figure:

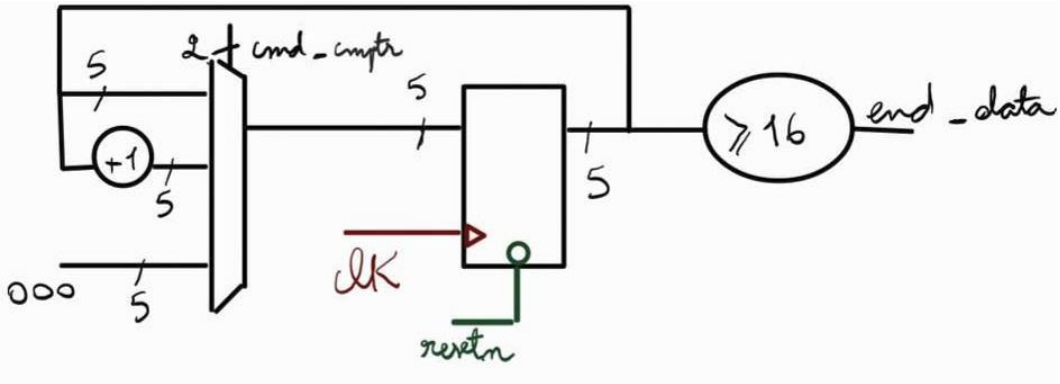



*Figure 6: Data Counter*

We can see in Figure 1 that we have to count 15 Sclk rising edges in order for the data to be transmitted, and an additional rising edge to signal the end of transmission to reinitialize the system. Therefore, we add a comparator that compares the value of the counter to the value ''16'', above which the signal ''end_data'' will be set to 1, reinitializing the system and the data counter back to 0.

The number of flip flops required to achieve this circuit is $n = \log_2(x_{cntr})$ with $x_{cntr} = 17$, which gives us after rounding up n = 5.

The data counter has 3 functions:

- The first function is incrementing the counter with each Sclk rising edge.
- The second function is to reset the counter to 0 each time end_data passes to 1.
- The third function is to memorize the value of the counter until the next rising edge of Sclk.

In order to implement the three functions, we need to use a multiplexer in order to switch between the first,second and third functions. This multiplexer will be controlled by the command ''cmd_ctr'' on 2 bits since we have 3 possibilities.

## ❖ Data Saver

At each rising edge of sclk, the data will be shifted left on the deserializer. However, we only want to read the value on the outputs of the deserializer after 16 rising edges of sclk have passed, which is the final result. This is especially important for the ''continuous mode'' in the ''implementation'' part because we will be reading the output of the deserializer all the time, and if the data being read is constantly shifting, we will not know which is the right value. Hence, we add the following RTL circuit which we called Data Saver:

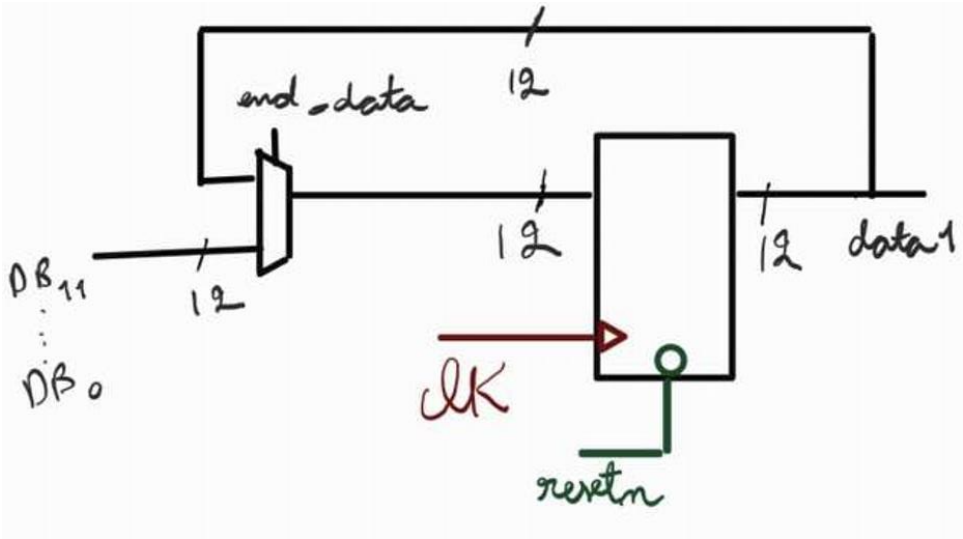



*Figure 7: Data Saver*

The data saver has 2 functions:

- The first function is reading the data from the outputs of the deserializer (DB11….DB0).
- The second function is to memorize the output.

In order to implement the two functions, we need to use a multiplexer in order to switch between the first and second functions. This multiplexer will be controlled by the command ''end_data'' which indicates that the data has been transmitted and 16 sclk rising edges have passed, allowing us to read the correct transmitted message.

It is important to note that we need to implement 2 data savers since we have two deserializers for the two ADCs on the PmodAD1.

## C. SPI Entity Ports

The following Figure shows the inputs and outputs of our entity “spi_pmod_ad1”:

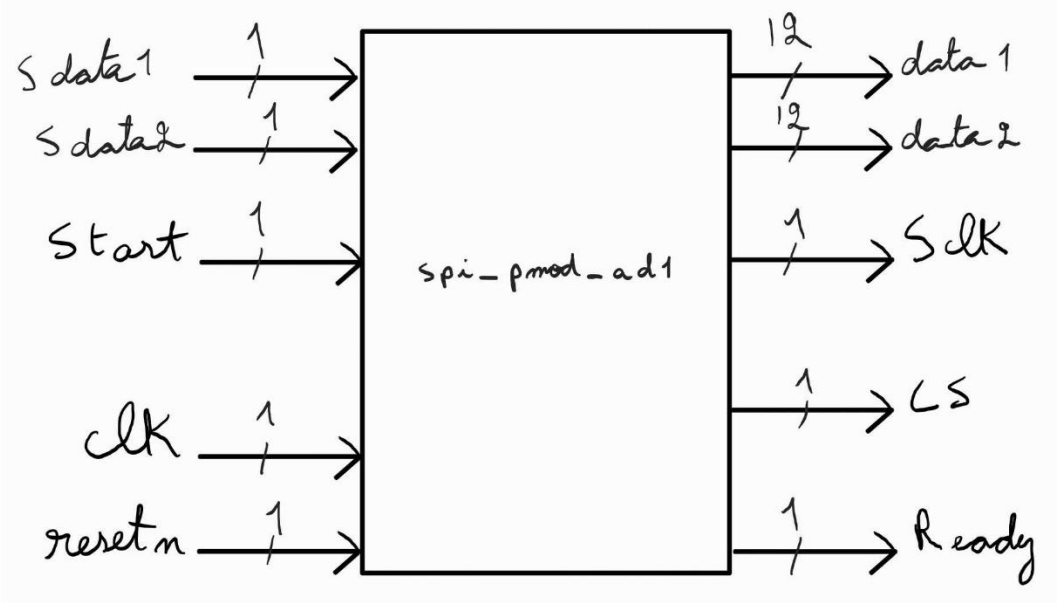



*Figure 8: spi_pmod_ad1 Entity*

The inputs of our entity are:

- sdata1: Serial data coming from the ADC1 on the pmodAd1 module.
- sdata2: Serial data coming from the ADC2 on the pmodAd1 module.
- start : When it is set to ‘’1’’, it will signal the beginning of the operation.
- clk : System clock.
- resetn : Asynchronus reset signal.

The outputs of our entity are:

- data1: Parallel data from the deserializer of ADC1.
- data2: Parallel data from the deserializer of ADC1.
- Sclk : SPI clock used for timing of the transfer of the data bits.
- cs : Chip Select, used to select the slave and start the data transfer.
- ready: Signal to indicate if the system is ready for a new transfer or not.

## D. State Machine

A state machine in VHDL is used to model and control the behavior of digital systems. It represents a finite set of states and transitions between these states based on input conditions. It provides a structured way to describe the sequence of operations a system should perform in response to various inputs. The following Figure shows the state machine that will pilot our RTL architecture.

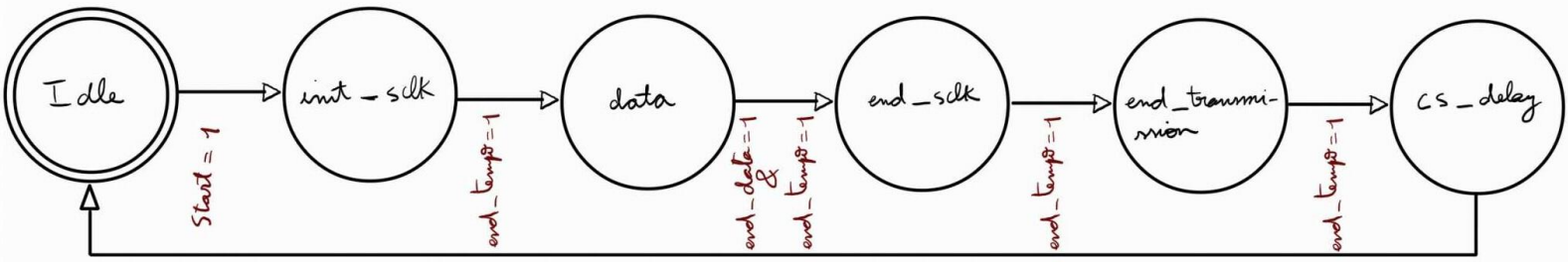



*Figure 9: System's State Machine*

In the following parts, we will explain the commands that we will control with our state machine as well as their functions. We will also explain how the state machine works at each stage.

### ❖ Table of Commands

The following table will showcase the commands that control our RTL structure along with which RTL components they control, the Functions that they realize and the values of the commands.

| Command Name | RTL Component | Functions | Values |
|---|---|---|---|
| **cmd_MISO** | Deserializer | Memorization | ''0'' |
| | | Shift Left | ''1'' |
| **Cmd_sclk** | Clock Generator | Memorization | ''00'' |
| | | Toggle | ''01'' |
| | | Set to ''1'' | ''11'' |
| **Cmd_tempo** | Temporization Counter | Set to ''0'' | ''0'' |
| | | Increment | ''1'' |
| **Cmd_ctr** | Data Counter | Set to ''0'' | ''00'' |
| | | Increment | ''01'' |
| | | Memorization | ''11'' |

## ❖ State Machine Sequence

At ''Idle'', the system is at its initial state, ''cs'' & ''sclk'' are set to 1, ''ready'' will be set to 1 indicating that the system is ready for a new transmission and the counters will be reinitialized to ''0''. When ''start'' is set to 1, the state machine will move to the next stage which is ''init_sclk''.

In the state ''init_sclk'', ''cs'' will be set to 0, signaling the beginning of the transmission, ''ready'' will be set to 0 meaning that the system is not currently ready for a new transmission and the sclk should start clocking. Before starting to toggle sclk, we should introduce a minimum delay ''t2'' as shown in Figure 1. In order to do so, we start incrementing the temporization counter, and when half a period of sclk (which is > t2) has passed, ''end_tempo'' will be set to 1 which will allow us to move to the next state ''data''.

In the state ''data'', ''sclk'' will toggle at each half period to form our serial clock. At each rising edge of ''sclk'', the data will be shifted left in our deserializer and the data counter will increment. On the $16^{th}$ rising edge, we will pull ''sclk'' back to 1, stop the clocking and move to the next state ''end_sclk''.

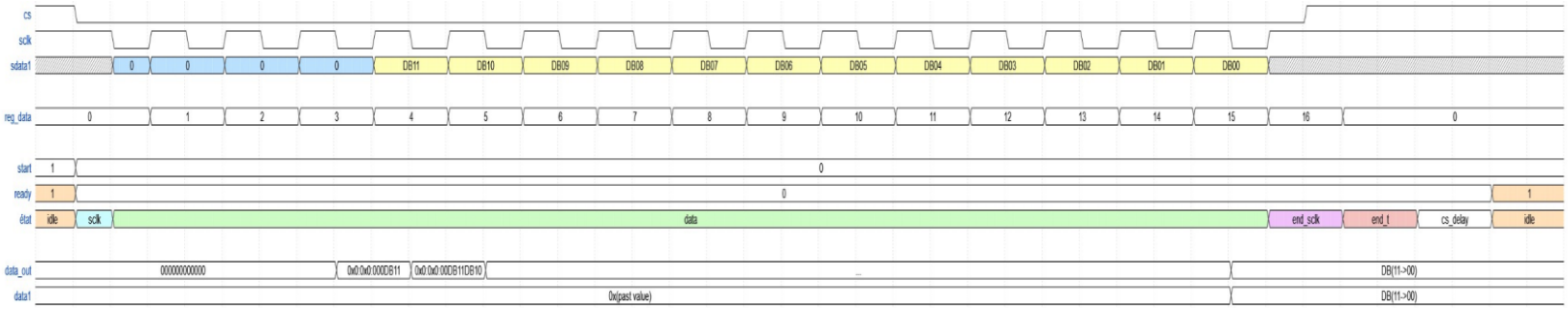
In the state ''end_sclk'', we wait for a duration of half a period of ''sclk'' to respect the delay that we can see in the serial interface timing diagram in Figure 1 right after the $16^{th}$ ''sclk'' rising edge, and then we move to the next state ''cs_delay''.

''cs_delay'' is the last state before going back to Idle. In this state, we pull ''cs'' back to 1. We then wait for half a period of ''sclk'' which is larger than the time ''t1 = 10 ns'' required in the serial interface timing diagram in Figure 1. Finally, we go back to the ''Idle'' state where the system is reinitialized and waiting for ''start'' to be set to 1 to start the new transfer.

| State | Action |
|---|---|
| **idle** | cs set to 1<br>cmd_tempo set to 0<br>cmd_MISO set to 0<br>cmd_sclk set to 11<br>cmd_cptr set to 00<br>Ready set to 1 |
| | **If start = 1**<br>next_state <= init_sclk |

| State | Action |
|---|---|
| **init_sclk** | cs set to 0<br>cmd_tempo set to 1<br>cmd_MISO set to 0<br>cmd_sclk set to 00<br>cmd_cptr set to 00<br>Ready set to 0 |
| | **If end_tempo = 1**<br>cmd_tempo set to 0<br>cmd_sclk set to 01<br>next_state <= data |
| **data** | cmd_sclk set to 00<br>cmd_tempo set to 1<br>cmd_cptr set to 11<br>cmd_MISO set to 0 |
| | **If end_tempo = 1 & sclk_wire = 1 & ctr_data = 16**<br>cmd_sclk set to 11<br>cmd_tempo set to 0<br>cmd_cptr set to 11<br>cmd_MISO set to 0 |
| | **If end_tempo = 1 & sclk_wire = 0 & ctr_data = 15**<br>cmd_sclk set to 11<br>cmd_tempo set to 0<br>cmd_cptr set to 01<br>cmd_MISO set to 0 |
| | **If end_tempo = 1 & sclk_wire = 1**<br>cmd_sclk set to 01<br>cmd_tempo set to 0<br>cmd_cptr set to 11<br>cmd_MISO set to 0 |
| | **If end_tempo = 1 & sclk_wire = 0**<br>cmd_sclk set to 01<br>cmd_tempo set to 0<br>cmd_cptr set to 01<br>cmd_MISO set to 1 |
| | **If end_tempo = 1 & end_data = 1**<br>cmd_tempo set to 0<br>cmd_cptr set to 00<br>next_state <= end_sclk |

| State | Action |
|---|---|
| **end_sclk** | cmd_tempo set to 1<br>cmd_MISO set to 0<br>cmd_sclk set to 11<br>cmd_cptr set to 00 |
| | **If end_tempo = 1**<br>cmd_tempo set to 0<br>next_state <= end_transmission |
| **end_transmission** | cs set to 1<br>cmd_tempo set to 1<br>cmd_MISO set to 0<br>cmd_sclk set to 11<br>cmd_cptr set to 00 |
| | **If end_tempo = 1**<br>cmd_tempo set to 0<br>next_state <= cs_delay |
| **cs_delay** | cmd_tempo set to 1<br>cmd_MISO set to 0<br>cmd_sclk set to 11<br>cmd_cptr set to 00 |
| | **If end_tempo = 1**<br>cmd_tempo set to 0<br>next_state <= idle |




Finally, we draw up a timeline for sending data. This chronogram shows the data counter (reg_data), the evolution of the state machine and the data register (data_out). Note that the data (data_out) is shifted to the left with each clock stroke of the SCLK signal. For ease of reading, we've only shown two values of data_out. In the *media* folder you'll find the chronogram.

## 2. Simulation

In this section, we will simulate our RTL architecture. We will use the following stimulus vectors:

| | |
|---|---|
| Vector 1 | 0000100000000000 |
| Vector 2 | 0000000000000001 |
| Vector 3 | 0000100100000001 |
| Vector 4 | 0000100000111000 |

The first vector allows us to test if DB11 is correctly sent.
The second vector allows us to test if DB00 is correctly sent.
The third and fourth vectors allow us to test if all the data bits (DB) are correctly sent.

These stimulus vectors are stored in an array.

We create automated simulation scripts. To do this, we draw inspiration from CRC (Cyclic Redundancy Check) scripts. Initially, we create an array (message) consisting of 16-bit test vectors. Using a loop, we iterate through each message in the array. A nested loop within the first one allows us to send each bit of the 16-bit test vector one by one. Two .tcl files are created to automate the simulation. The first file is called simu.tcl. The simulation duration is specified in this file. A second file, chrono.tcl, is used to specify which signals will be visualized during the simulation. To initiate the automated simulation, we execute the following command in the Questasim terminal:
`source simu.tcl`.

We aim to automatically verify the correctness of the simulation. Following the second loop, we wait for the rising edge of the 'ready' signal. After this action, we compare the output with the first 12 bits, starting from the LSB, of message number 'i.' If the message is correct, we display the following message:

```
Canal 1 SPI OK
Canal 1 SPI resultat obtenu : (valeur en hexadécimal de data1)
```

If the data does not correspond to the first 12 bits of the message, we display:

```
Canal 1 SPI not OK
Canal 1 SPI resultat attendu : (valeur en hexadécimal du message)
Canal 1 SPI resultat obtenu : (valeur en hexadécimal de data1)
```

The Pmod ADC1 consists of two channels. The two channels are sdata1 and sdata2. So, we are testing both channels. The previous mechanisms implemented for the first channel are used for the second.

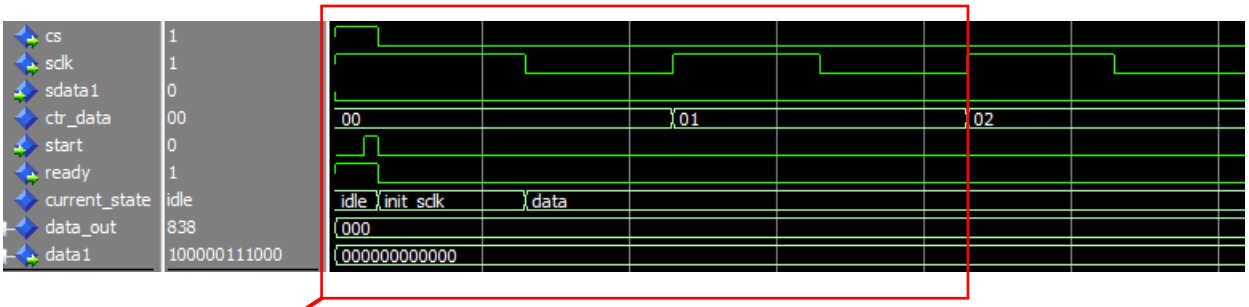
When we run simu.tcl, we get these results:

```
# ** Note: Test 1
#    Time: 10 ns  Iteration: 0  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 1 SPI OK
#    Time: 1815 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 1 resultat obtenu : 0x800
#    Time: 1815 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 2 SPI OK
#    Time: 1815 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 2 resultat obtenu : 0x838
#    Time: 1815 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Test 2
#    Time: 1815 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 1 SPI OK
#    Time: 3625 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 1 resultat obtenu : 0x001
#    Time: 3625 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 2 SPI OK
#    Time: 3625 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 2 resultat obtenu : 0x901
#    Time: 3625 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Test 3
#    Time: 3625 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 1 SPI OK
#    Time: 5435 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 1 resultat obtenu : 0x901
#    Time: 5435 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 2 SPI OK
#    Time: 5435 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 2 resultat obtenu : 0x001
#    Time: 5435 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Test 4
#    Time: 5435 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 1 SPI OK
#    Time: 7245 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 1 resultat obtenu : 0x838
#    Time: 7245 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 2 SPI OK
#    Time: 7245 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 2 resultat obtenu : 0x800
#    Time: 7245 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
```

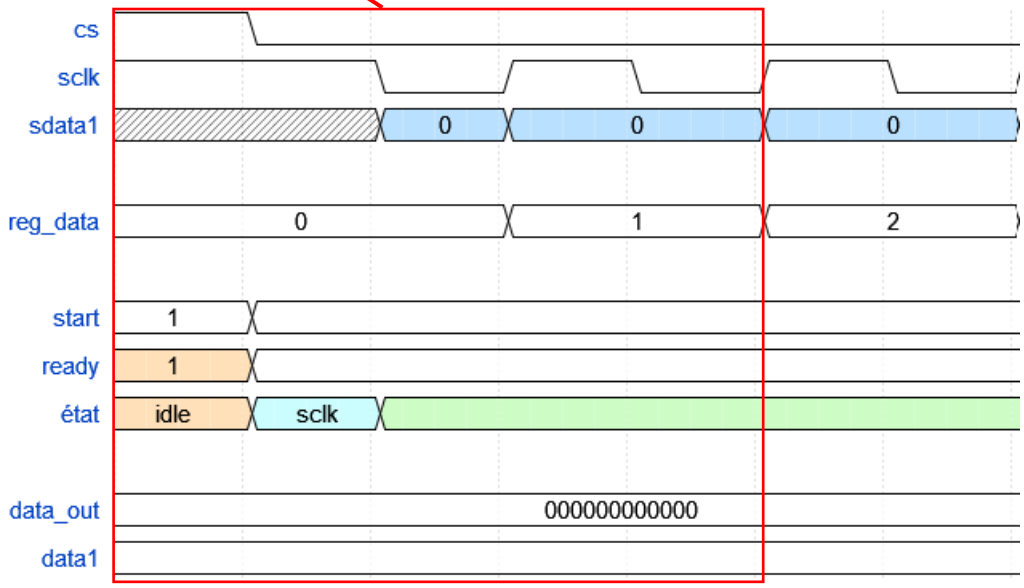
We analyze the results. Channel 1 (canal 1) and Channel 2 (canal 2) are functional. Sending the data and retrieving it works correctly.

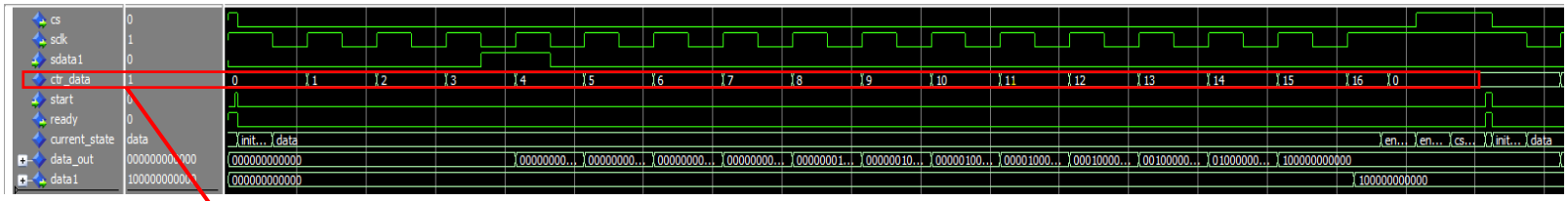
We will take a closer look at a few frames.

Let's take the case of channel 1 with test vector number 1: 0000100000000000.

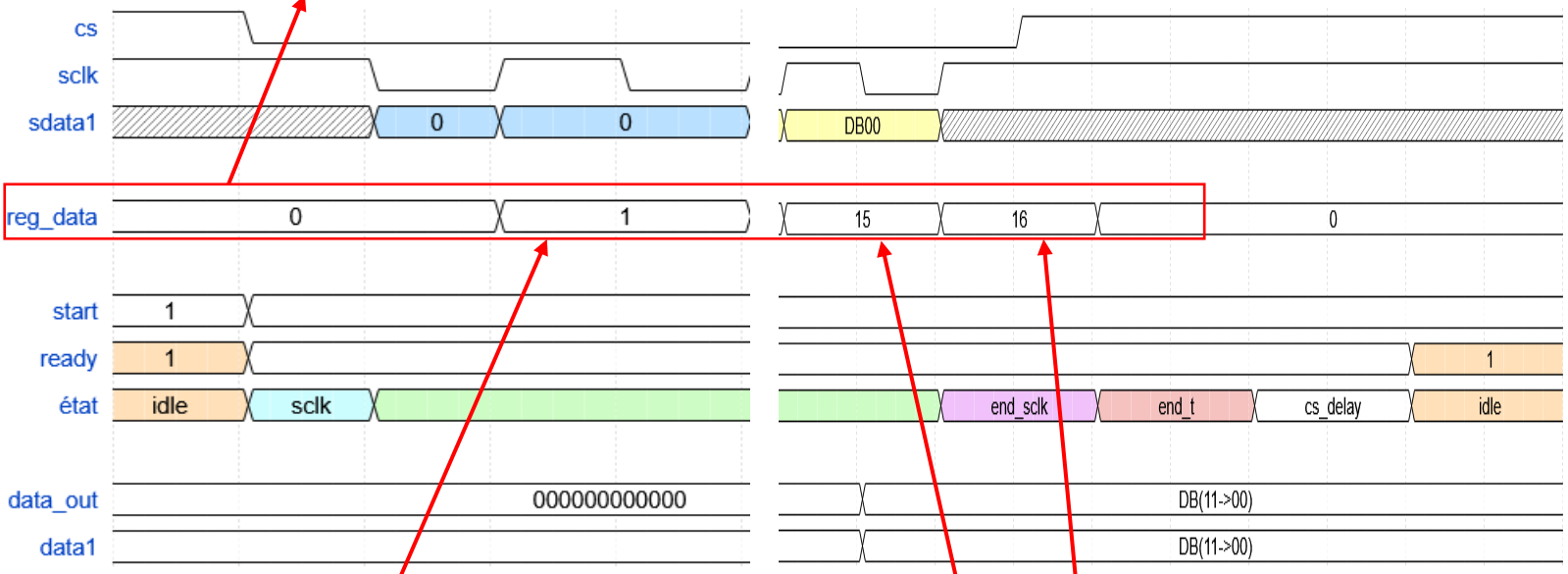



Initially, `data1` and `data_out` are initialized to zero. Indeed, the current state is idle. The component is ready. When the start signal is activated, the `cs` signal drops and the component enters the `init_sclk` state. At the end of this state, the signal `sclk` drop. The component enters the `data` state. In this state, the ready signal is low. The data counter begins at zero and then the counter increments each period of sclk.






During the transmission, the ctr_data increments. This counter starts at zero and stops at sixteen.



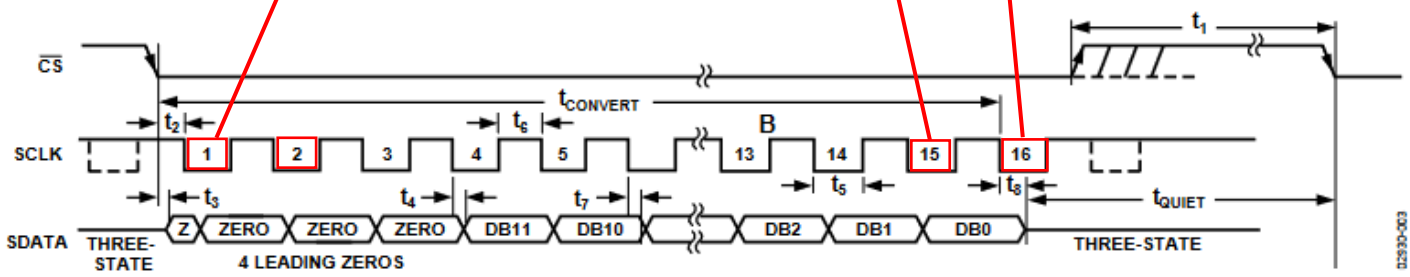
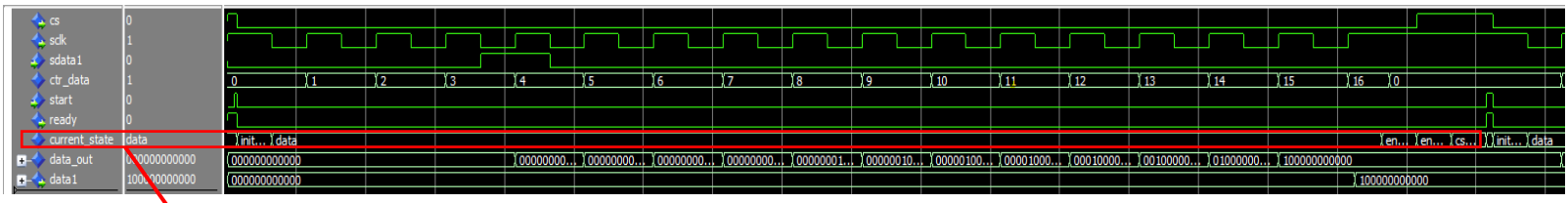



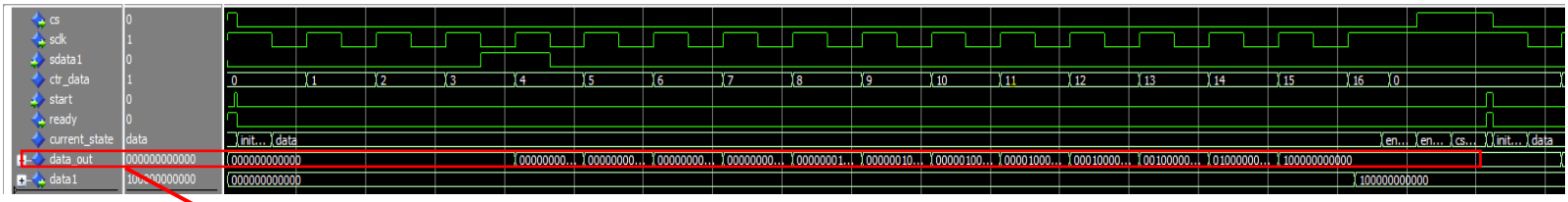

*Figure 3. AD7476A Serial Interface Timing Diagram*

When the counter reaches the value 16, it sets the SCLK signal to one.



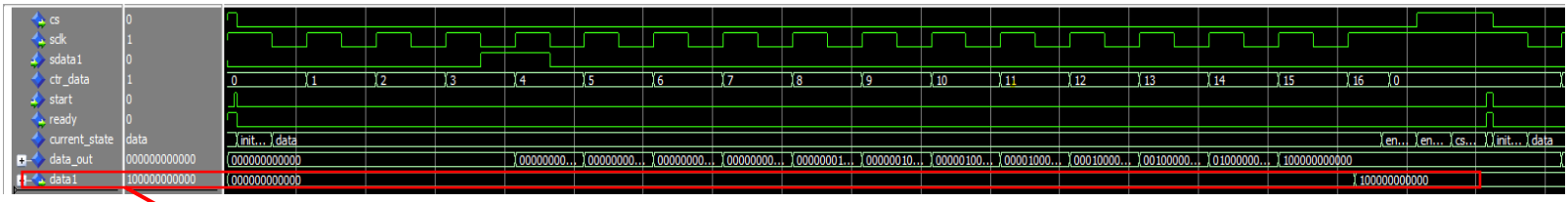

During transmission, the component passes through all the FSM states. Which are:

- idle
- init_sclk
- data
- end_sclk
- end_transmission
- cs_delay



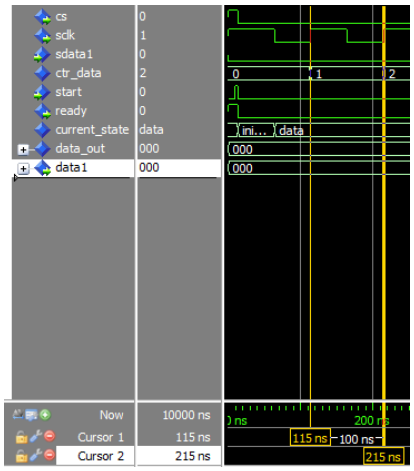

When the value of ctr_data is less than 3, the signal is always equal to the vector 000000000000. When the value of ctr_data becomes 4, we have this vector 0000 0000 000DB11. The component works and when the data counter (ctr_data) reaches the number sixteen, we obtain this vector:

DB11DB10DB09DB08DB07DB06DB05DB04DB03DB02DB01DB00

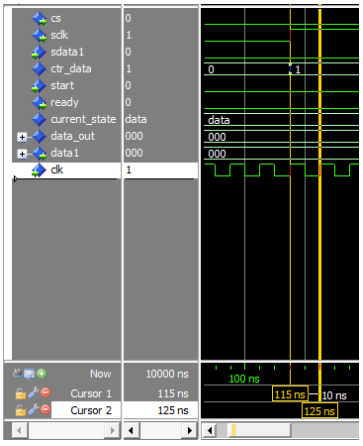



When we want to display the value in continuous mode, we do not want to see the shift of each bit to the left. Instead, we want to see the final value of the transmission.


When the data counter (ctr_data) reaches 16, the SCLK signal changes to 1 - the component is still in the data state. When the component is in the end_sclk state, the SCLK signal is always set to 1. Then the component goes into the end_transmission state, and the CS signal is set to 1. Finally, in the cs_delay signal, we wait for the time required by the protocol.




The SCLK period is 100 ns which corresponds to a frequency of 10 MHz. This is the SPI frequency we used in the testbench.




The clk period is 10 ns which corresponds to a frequency of 100 MHz.

```
dut : entity work.spi_pmod_adl
  generic map(
    f_clk => 100_000_000.0,
    f_spi => 10_000_000.0
  )
```

The clock runs at the correct frequency in relation to the generic frequency settings used in the testbench.

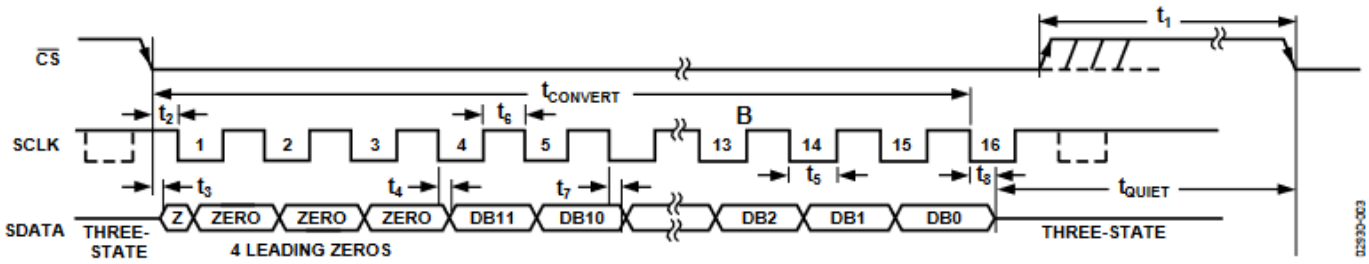
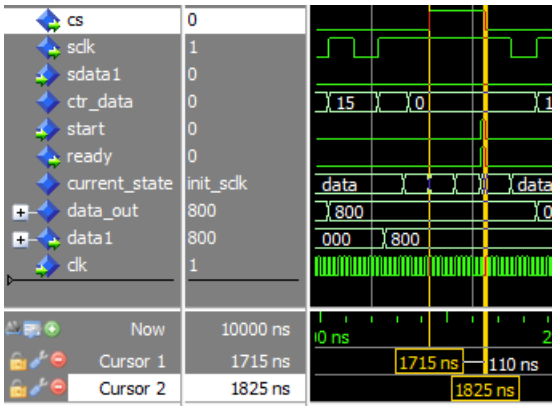



*Figure 3. AD7476A Serial Interface Timing Diagram*

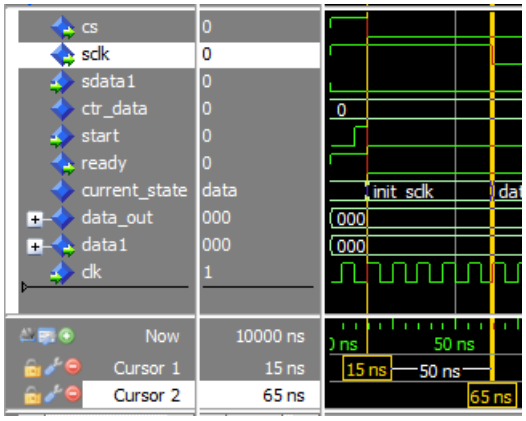
Let's check the times :

- t1 : In simulation we have 110 ns.



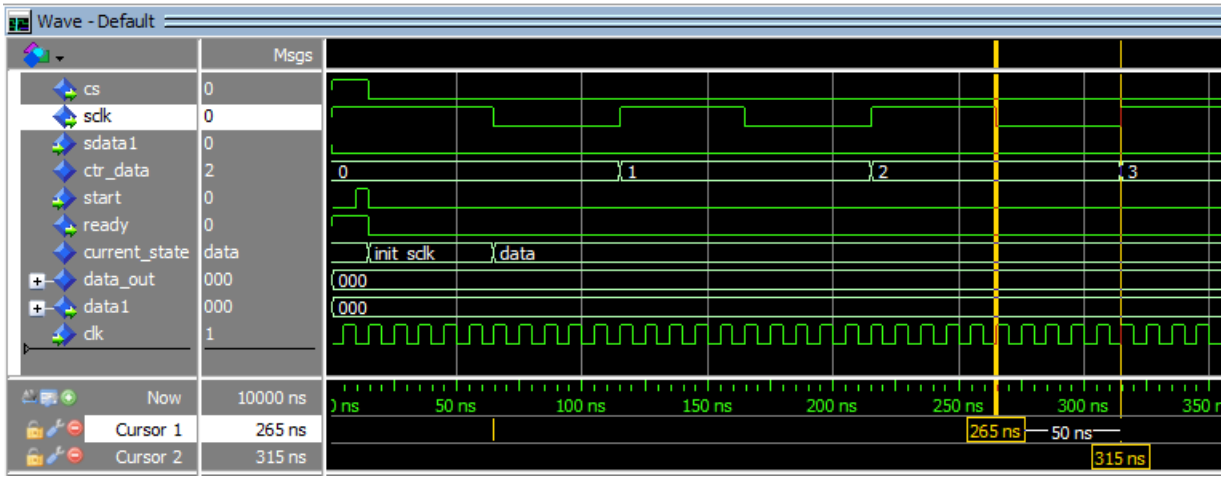

Timing specifications of the datasheet impose t1: 10 ns min for minimum CS pulse width.

- t2: In simulation we have 50 ns.



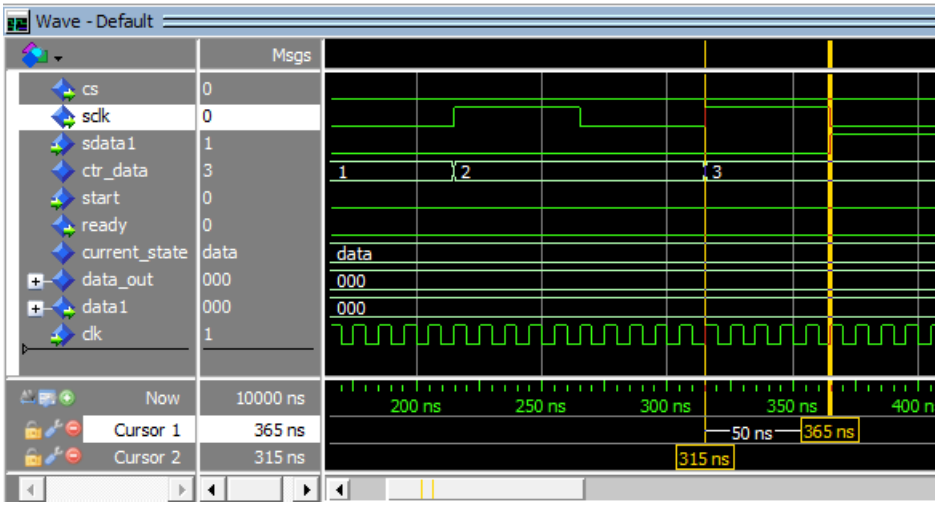

Timing specifications of the datasheet impose t2: 10 ns min for CS to SCLK setup time.

- t3: In simulation we cannot calculate this timing because we don't use three-state for the sdata signal.
- t4: In simulation we cannot calculate this timing.
- t5: In simulation we obtain 50 ns.



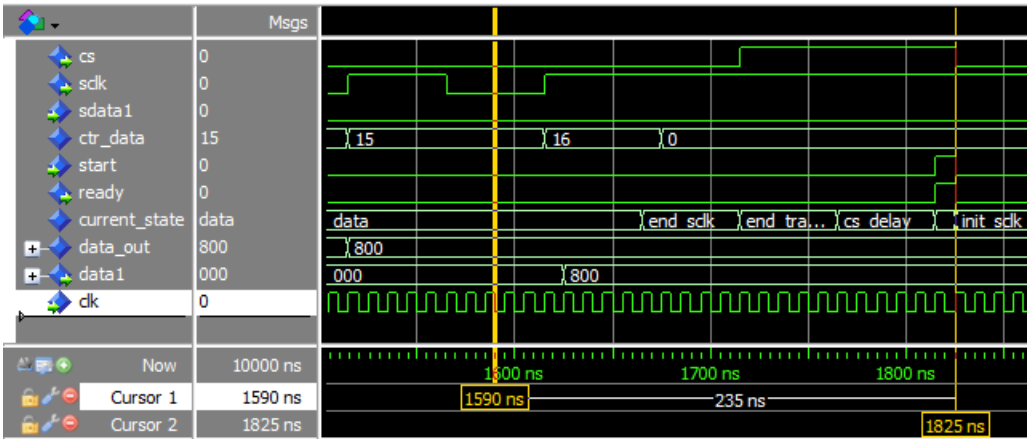

Timing specifications of the datasheet impose t5: 0.4 * t_sclk = 0.4*100 = 40 ns min for SCLK low pulse width.

- t6: In simulation we obtain 50 ns.



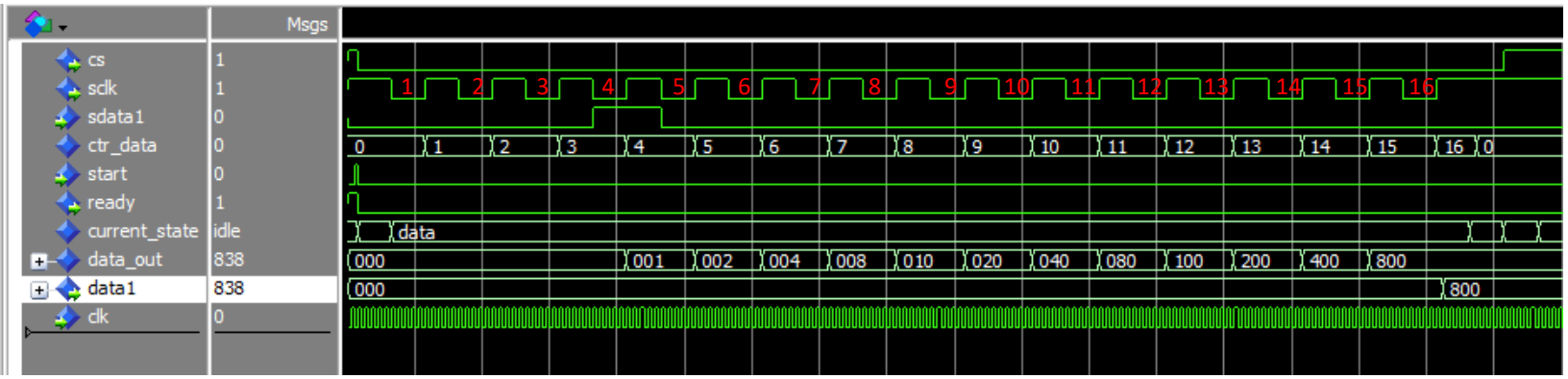

Timing specifications of the datasheet impose t6 : 0.4 * t_sclk = 0.4*100 = 40 ns min for SCLK high pulse width.

- t7: In simulation we cannot obtain this timing.
- t8: In simulation we cannot obtain this timing.
- tQuiet: In simulation we obtain 235 ns.

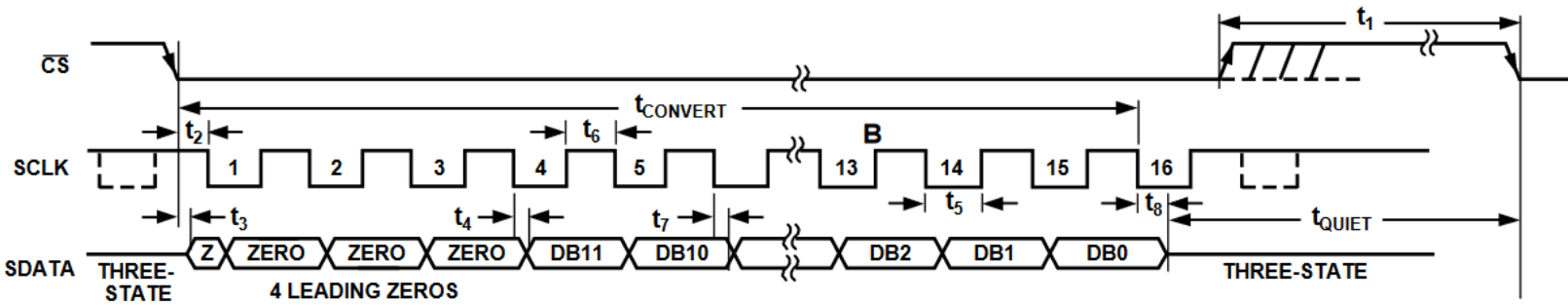



Timing specifications of the datasheet impose tQuiet: 50 ns min for minimum quiet time required between bus relinquish and start of next conversion.

We will count all the rising edges from the SCLK.

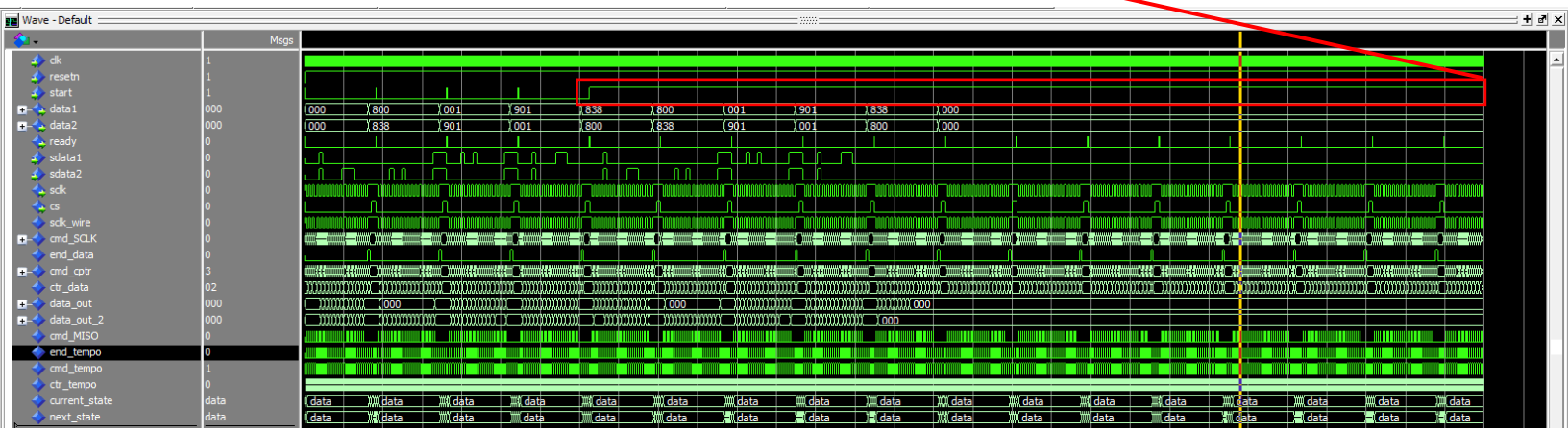



Finally, we count sixteen rising edges from the SCLK. We also find the same result in the datasheet.




We have analyzed the first test vector in detail. For the first vector, our component is functional. We use an automatic test. Finally, our component works for the four test vectors in the table.

We can ask how it works when the signal start is non-zero.



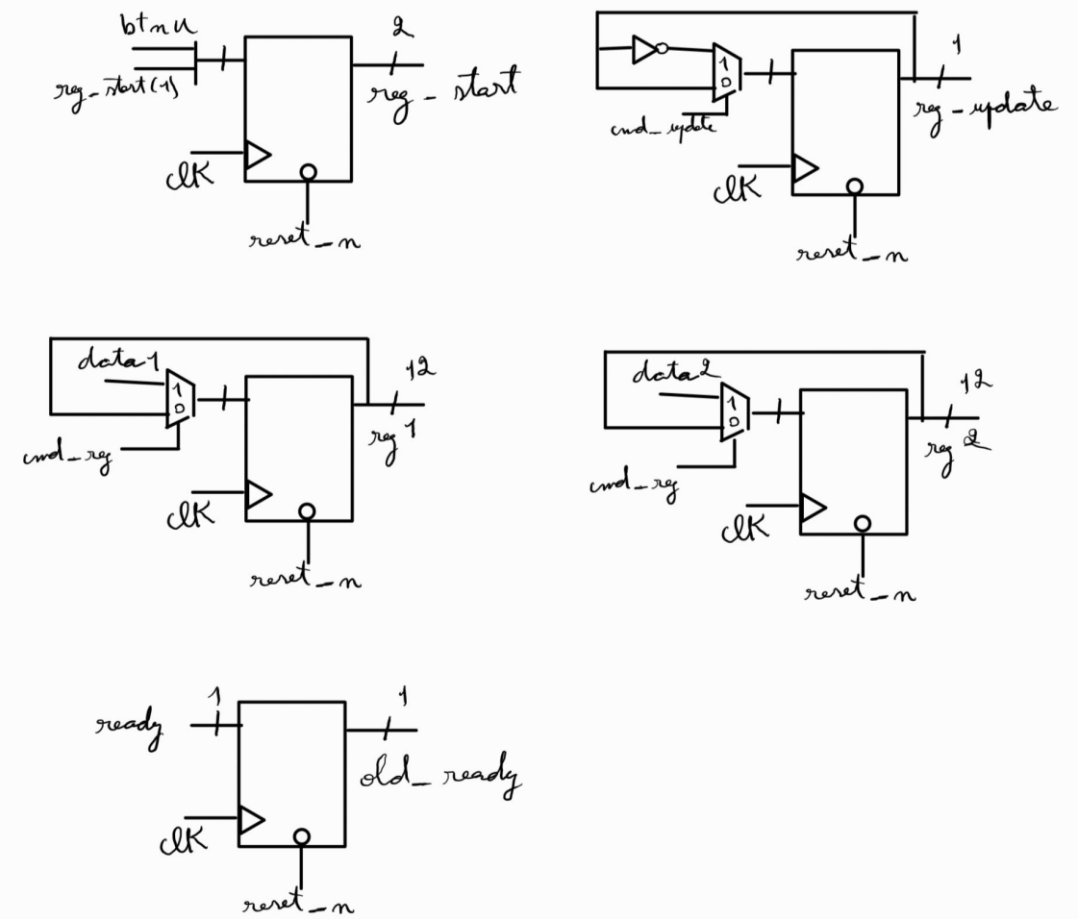

Firstly, we check our automatic test:

```
# ** Note: Test 5
#    Time: 7245 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 1 SPI OK
#    Time: 9055 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 1 resultat obtenu : 0x800
#    Time: 9055 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 2 SPI OK
#    Time: 9055 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 2 resultat obtenu : 0x838
#    Time: 9055 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Test 6
#    Time: 9055 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 1 SPI OK
#    Time: 10865 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 1 resultat obtenu : 0x001
#    Time: 10865 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 2 SPI OK
#    Time: 10865 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 2 resultat obtenu : 0x901
#    Time: 10865 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Test 7
#    Time: 10865 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 1 SPI OK
#    Time: 12675 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 1 resultat obtenu : 0x901
#    Time: 12675 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 2 SPI OK
#    Time: 12675 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 2 resultat obtenu : 0x001
#    Time: 12675 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Test 8
#    Time: 12675 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 1 SPI OK
#    Time: 14485 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
```

```
# ** Note: Canal 1 resultat obtenu : 0x838
#    Time: 14485 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Warning: Canal 2 SPI OK
#    Time: 14485 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
# ** Note: Canal 2 resultat obtenu : 0x800
#    Time: 14485 ns  Iteration: 2  Instance: /tb_spi_pmod_ad1
```

# 3. Implementation

1) From the demo VHDL code, we can draw the following RTL structure:



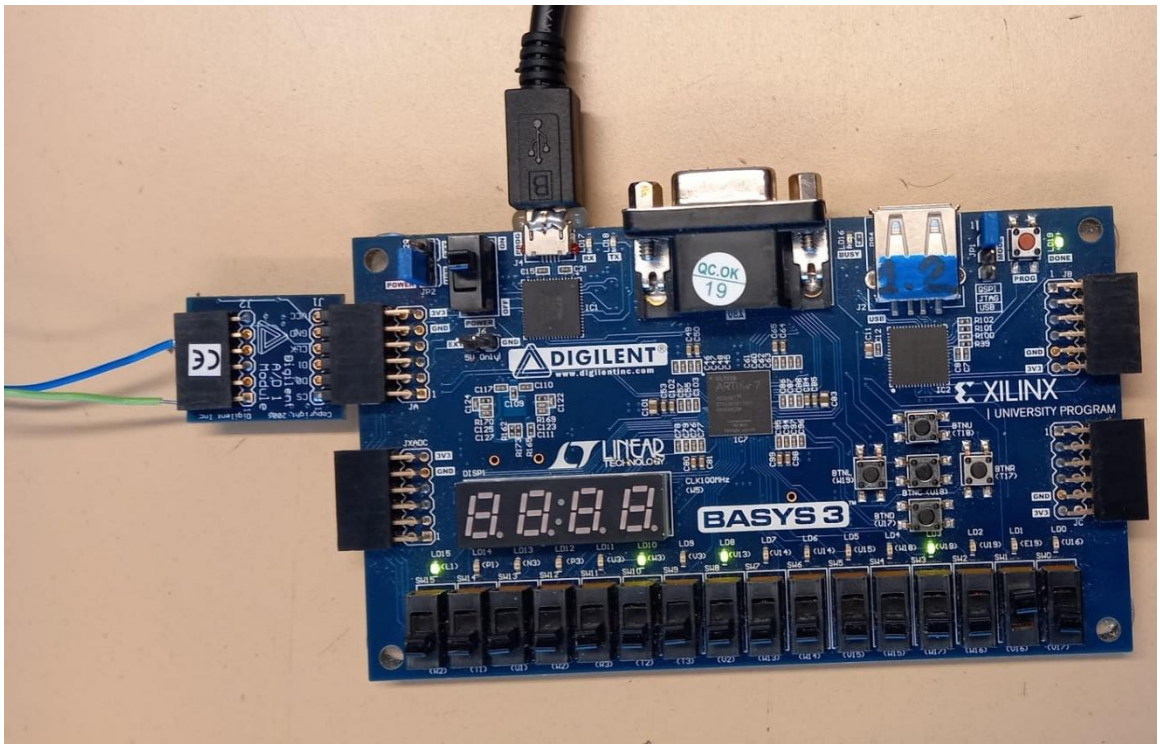

- ''reg_start'' which will be responsible for starting the transmission depends on the previous value of ''reg_start(1)'' and the value of the start push button ''btnu''.
- When ''ready'' = 1 and ''old_ready'' = 0, ''reg_update'' will toggle indicating that we have new data.
- When ''ready'' = 1 and ''old_ready'' = 0, signaling the end of a transmission, ''reg1'' and ''reg2'' will save the values of ''data1'' and ''data2'' respectively until the next rising edge of ''ready''. We did the same thing in our RTL structure as shown in Figure 7 in order to only visualize the final value of the transmission in continuous mode before knowing that it was implemented in the demo file. The simulation still works perfectly, however, this means that we have a redundancy in the components and we're using unnecessary resources.
- At each rising edge of ''clk'', ''old_ready'' will save the previous value of ''ready''.

2) We realized the implementation on the basys3 and validated the functioning of the system in continuous data acquisition mode and in data acquisition at pression of btnu by the teacher.

   In order to do so, we used a DC power supply and we set the voltage to values between 0V and 3.3V which is the voltage range of the ADC. We connected the power supply to ground and A0 of the PmodAd1.

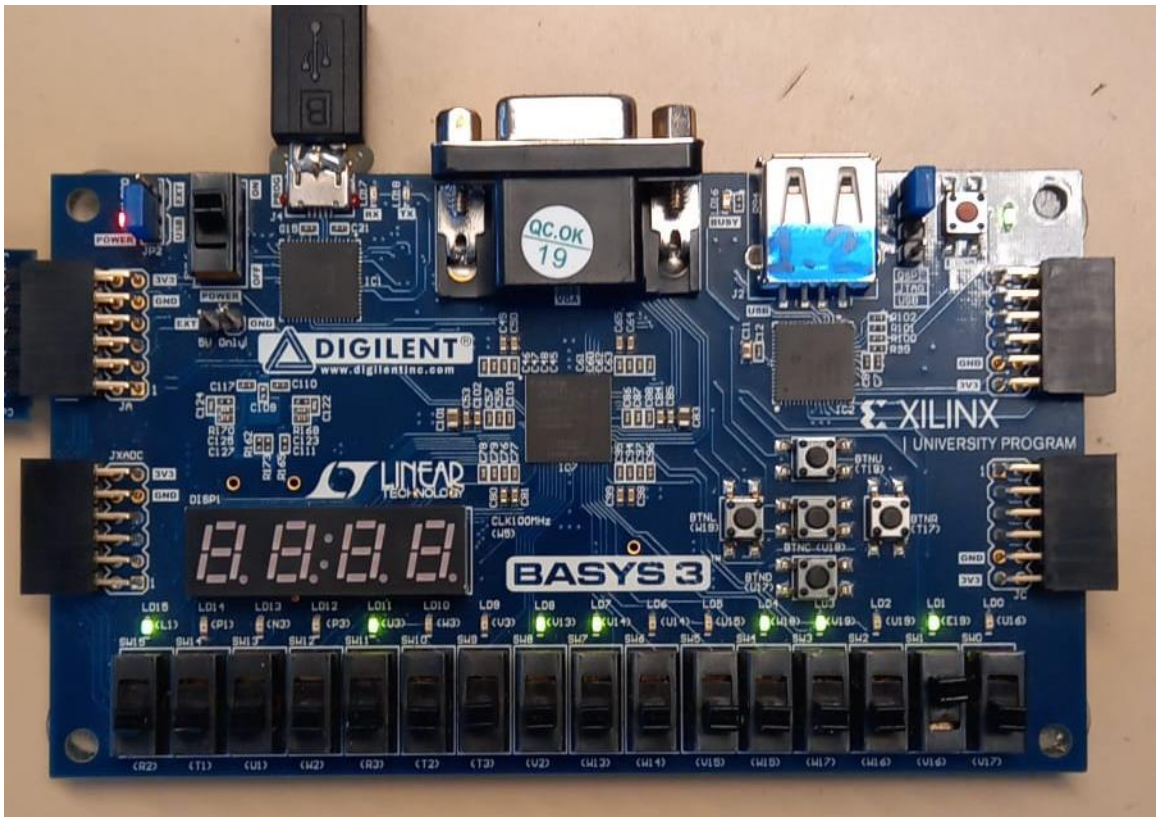
   Since the AD7476A is a 12-bit ADC, the number of voltage steps of the ADC are $N = 2^{12} = 4096$. The full scale of the ADC is $FS = 3.3\,V$. The output digital value of the ADC will be indicated by the leds LD11 to LD0 with LD11 indicating the most significant bit. The formula that gives the analog input value being measured is: $V_{analog} = \frac{D_{out}}{4096} * FS$ with $D_{out}$ being the output digital value of the ADC in decimal.

   In the following Image, we set the DC voltage to 1V, we got the following result on the basys3:




   The leds LD11 to LD0 show a value of ''10100001000'' which is equal to 1288 in decimal. This gives us $V_{analog} = \frac{1288}{4096} * 3.3 = 1.0377\,V$ which is correct because it is the value that we have set.

In the following Image, we set the DC voltage to 2V, we got the following result on the basys3:




The leds LD11 to LD0 show a value of '' 100110011010'' which is equal to 2458 in decimal. This gives us $V_{analog} = \frac{2458}{4096} * 3.3 = 1.98\ V$ which is correct because it is the value that we have set.

The small errors in the values are due to the inaccuracies introduced by the DC power supply.

3) Synthesis

If we count the number of flip-flops on our RTL structure, we have:

- 3 flip_flops for the temporization counter
- 5 flip_flops for the data counter
- 24 flip-flops for the deserializers of data1 and data2
- 24 flip-flops for the data saver registers of data1 and data2
- 1 flip-flop for the SCLK generator

If we add the number of flip-flops in our RTL structure, we find that we have a total of 57 flip-flops.

If we count the number of flip-flops in the demo_pmod_ad1, we have :

- 2 flip-flops for reg_sart
- 1 flip-flop for reg_update
- 1 flip-flop for old_ready
- 24 flip-flops to memorize the values of data1 and data2.

If we add the number of flip-flops in our demo_pmod_ad1 we get 28 flip-flops.

We can also, see that in our state machine we have 6 states, so we need a minimum of 3 flip-flops in order to code the state machine. In total, we have 88 flip-flops in our system.

We will analyze the synthesis report.

```
Report Check Netlist:


+------+------------------+-------+---------+-------+------------------+
|      |Item              |Errors |Warnings |Status |Description       |
+------+------------------+-------+---------+-------+------------------+
|1     |multi_driven_nets |      0|        0|Passed |Multi driven nets |
+------+------------------+-------+---------+-------+------------------+
```

Vivado reports that there are no short circuits.

```
-----------------------------------------------------------------------------
State             |        New Encoding      |          Previous Encoding
-----------------------------------------------------------------------------
             idle |                      000 |                             000
       init_sclk |                      001 |                             001
            data |                      010 |                             010
         end_sclk |                      011 |                             011
end_transmission |                      100 |                             100
         cs_delay |                      101 |                             101
-----------------------------------------------------------------------------
```

Vivado has recognized all our states in the state machine. Vivado coded the state machine on 3 bits as we expected.

```
---------------------------------------------------------------------------------
Start RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
Hierarchical RTL Component report
Module demo_pmod_ad1
Detailed RTL Component Info :
+---Registers :
               12 Bit    Registers := 2
                2 Bit    Registers := 1
                1 Bit    Registers := 2
+---Muxes :
  2 Input   12 Bit        Muxes := 1
  2 Input    1 Bit        Muxes := 1
Module spi_pmod_ad1
Detailed RTL Component Info :
+---Adders :
  2 Input    5 Bit       Adders := 1
  2 Input    3 Bit       Adders := 1
+---Registers :
               12 Bit    Registers := 4
                5 Bit    Registers := 1
                3 Bit    Registers := 1
                1 Bit    Registers := 1
+---Muxes :
  2 Input    5 Bit        Muxes := 2
  2 Input    3 Bit        Muxes := 2
  6 Input    3 Bit        Muxes := 1
  2 Input    2 Bit        Muxes := 6
  6 Input    2 Bit        Muxes := 2
  2 Input    1 Bit        Muxes := 8
  6 Input    1 Bit        Muxes := 5
---------------------------------------------------------------------------------
Finished RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
```

Vivado informs that it has recognized in the RTL component of spi_pmod_ad1 :

- Four registers of 12 flip-flops
- One register of 5 flip-flops
- Three registers of 1 flip-flop
- One register of 1 flip-flop

If we add the number of flip-flops in our spi_pmod_ad1 we get 57 flip-flops.

Vivado informs that it has recognized in the RTL component of demo_pmod_ad1 :

- Two registers of 12 flip-flops
- One register of 2 flip-flops
- Two registers of 1 flip-flop

If we add the number of flip-flops in our demo_pmod_ad1 we get 28 flip-flops.

By adding the number of flip-flops that we have from spi_pmod_ad1 and demo_pmod_ad1 (85) with the number of flip-flops from our state machine (3), we will get a total of 88 flip-flops for our whole system, which is the same number from when we counted the flip-flops by hand in our RTL structure.

We check that there is no latch.

Report Cell Usage:

| | Cell | Count |
|---|---|---|
| 1 | BUFG | 1 |
| 2 | LUT1 | 1 |
| 3 | LUT3 | 15 |
| 4 | LUT4 | 3 |
| 5 | LUT5 | 7 |
| 6 | LUT6 | 8 |
| 7 | FDCE | 84 |
| 8 | FDPE | 4 |
| 9 | IBUF | 7 |
| 10 | OBUF | 18 |

There is no latch.

Warnings :

We have six warnings :

```
[Common 17-1361] You have specified a new message control rule that is equivalent to an existing rule with attributes ' -id {Synth 8-6859}  -new_severity {ERROR} '. The existing rule will be replaced.
```

This error corresponds to the rules used in Vivado.

```
[Synth 8-3917] design demo_pmod_ad1 has port led[13] driven by constant 0
[Synth 8-3917] design demo_pmod_ad1 has port led[12] driven by constant 0
[Synth 8-3917] design demo_pmod_ad1 has port led[13] driven by constant 0
[Synth 8-3917] design demo_pmod_ad1 has port led[12] driven by constant 0
```

Those errors are caused by the fact that we set led[13] and led[12] to zero.

```
[Synth 8-7080] Parallel synthesis criteria is not met
```

Our architecture is very small. Vivado can't parallelize the synthesis.

## 4) Utilization - Place Design

### 1. Slice Logic

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice LUTs | 23 | 0 | 0 | 20800 | 0.11 |
| LUT as Logic | 23 | 0 | 0 | 20800 | 0.11 |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| Slice Registers | 88 | 0 | 0 | 41600 | 0.21 |
| Register as Flip Flop | 88 | 0 | 0 | 41600 | 0.21 |
| Register as Latch | 0 | 0 | 0 | 41600 | 0.00 |
| F7 Muxes | 0 | 0 | 0 | 16300 | 0.00 |
| F8 Muxes | 0 | 0 | 0 | 8150 | 0.00 |

### 1.1 Summary of Registers by Type

| Total | Clock Enable | Synchronous | Asynchronous |
|---|---|---|---|
| 0 | _ | - | - |
| 0 | _ | - | Set |
| 0 | _ | - | Reset |
| 0 | _ | Set | - |
| 0 | _ | Reset | - |
| 0 | Yes | - | - |
| 4 | Yes | - | Set |
| 84 | Yes | - | Reset |
| 0 | Yes | Set | - |
| 0 | Yes | Reset | - |

### 2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 24 | 0 | 0 | 8150 | 0.29 |
| SLICEL | 14 | 0 | | | |
| SLICEM | 10 | 0 | | | |
| LUT as Logic | 23 | 0 | 0 | 20800 | 0.11 |
| using O5 output only | 0 | | | | |
| using O6 output only | 12 | | | | |
| using O5 and O6 | 11 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| LUT as Shift Register | 0 | 0 | | | |
| Slice Registers | 88 | 0 | 0 | 41600 | 0.21 |
| Register driven from within the Slice | 13 | | | | |
| Register driven from outside the Slice | 75 | | | | |
| LUT in front of the register is unused | 66 | | | | |
| LUT in front of the register is used | 9 | | | | |
| Unique Control Sets | 5 | | 0 | 8150 | 0.06 |

Resource summary :

- 24 slices
- 23 LUTs
- 88 DFFs

3. Memory

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Block RAM Tile | 0 | 0 | 0 | 50 | 0.00 |
| RAMB36/FIFO* | 0 | 0 | 0 | 50 | 0.00 |
| RAMB18 | 0 | 0 | 0 | 100 | 0.00 |

4. DSP

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| DSPs | 0 | 0 | 0 | 90 | 0.00 |

5. IO and GT Specific

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Bonded IOB | 25 | 25 | 0 | 106 | 23.58 |
| IOB Master Pads | 13 | | | | |
| IOB Slave Pads | 11 | | | | |
| Bonded IPADs | 0 | 0 | 0 | 10 | 0.00 |
| Bonded OPADs | 0 | 0 | 0 | 4 | 0.00 |
| PHY_CONTROL | 0 | 0 | 0 | 5 | 0.00 |
| PHASER_REF | 0 | 0 | 0 | 5 | 0.00 |
| OUT_FIFO | 0 | 0 | 0 | 20 | 0.00 |
| IN_FIFO | 0 | 0 | 0 | 20 | 0.00 |
| IDELAYCTRL | 0 | 0 | 0 | 5 | 0.00 |
| IBUFDS | 0 | 0 | 0 | 104 | 0.00 |
| GTPE2_CHANNEL | 0 | 0 | 0 | 2 | 0.00 |
| PHASER_OUT/PHASER_OUT_PHY | 0 | 0 | 0 | 20 | 0.00 |
| PHASER_IN/PHASER_IN_PHY | 0 | 0 | 0 | 20 | 0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY | 0 | 0 | 0 | 250 | 0.00 |
| IBUFDS_GTE2 | 0 | 0 | 0 | 2 | 0.00 |
| ILOGIC | 0 | 0 | 0 | 106 | 0.00 |
| OLOGIC | 0 | 0 | 0 | 106 | 0.00 |

6. Clocking

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| BUFGCTRL | 1 | 0 | 0 | 32 | 3.13 |
| BUFIO | 0 | 0 | 0 | 20 | 0.00 |
| MMCME2_ADV | 0 | 0 | 0 | 5 | 0.00 |
| PLLE2_ADV | 0 | 0 | 0 | 5 | 0.00 |
| BUFMRCE | 0 | 0 | 0 | 10 | 0.00 |
| BUFHCE | 0 | 0 | 0 | 72 | 0.00 |
| BUFR | 0 | 0 | 0 | 20 | 0.00 |

Resource summary :

- 0 Block RAM
- 0 DSP
- 1 clock tree (BUFGCTRL)
- 0 PLL

5) Timing Summary – Route Design:

Design Timing Summary

| WNS(ns) | TNS(ns) | TNS Failing Endpoints | TNS Total |
|---|---|---|---|
| 6.307 | 0.000 | 0 | 163 |

Set-up

| WHS(ns) | THS(ns) | THS Failing Endpoints | THS Total Endpoints |
|---|---|---|---|
| 0.152 | 0.000 | 0 | 163 |

Hold

| WPWS(ns) | TPWS(ns) | TPWS Failing Endpoints | TPWS Total Endpoints |
|---|---|---|---|
| 4.500 | 0.000 | 0 | 89 |

Pulse width

Slack, also known as Worst Negative Slack (WNS), is measured at 6.307 nanoseconds, and it is required to be greater than or equal to 0. Additionally, Worst Hold Slack (WHS) should also be greater than or equal to 0.

Calculation of the maximum frequency :

$$f_{max} = \frac{1}{period - WNS} = \frac{1}{(10 - 6.307) * 10^{-9}} = 271MHz$$

period = 10 ns

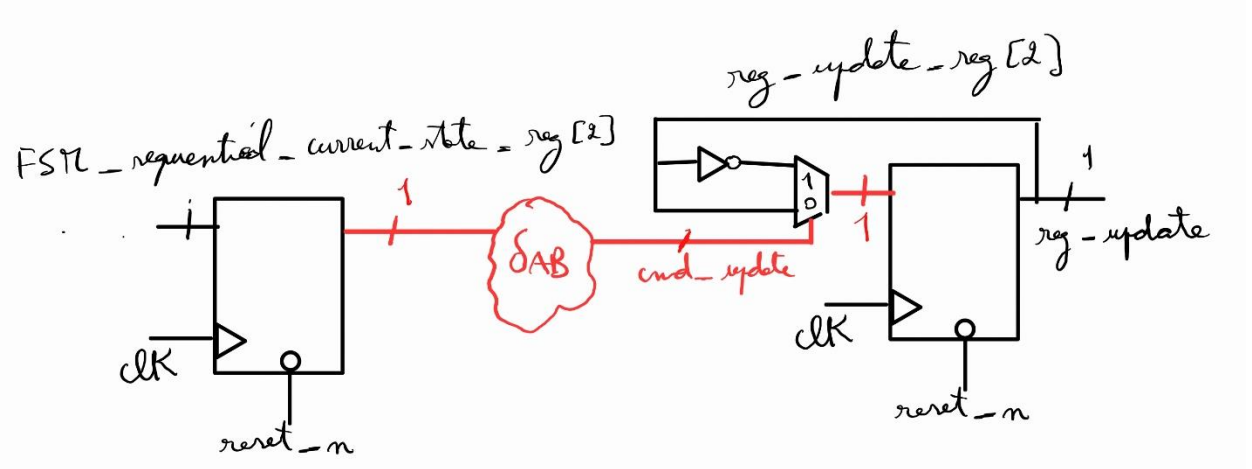
```
----------------------------------------------------------------------------------------------------
| Clock Summary
| -------------
----------------------------------------------------------------------------------------------------

Clock        Waveform(ns)       Period(ns)      Frequency(MHz)
-----        ------------       ----------      --------------
sys_clk_pin  {0.000 5.000}      10.000          100.000
```

This is what we specified in the .xdc file.

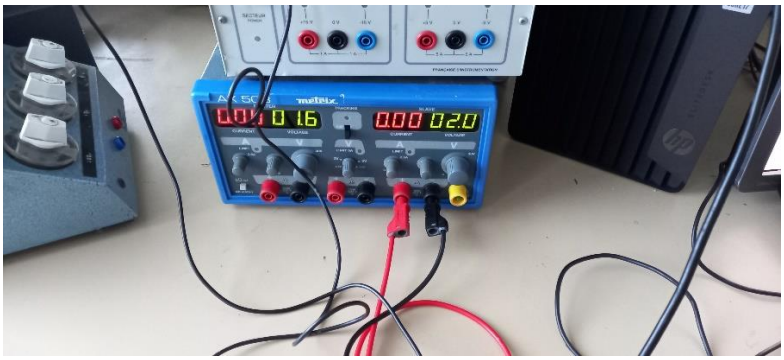
We determine the critical path :

```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             6.307ns  (required time - arrival time)
  Source:                 dut/FSM_sequential_current_state_reg[2]/C
                            (rising edge-triggered cell FDCE clocked by
sys_clk_pin  {rise@0.000ns fall@5.000ns period=10.000ns})
  Destination:            reg_update_reg/CE
                            (rising edge-triggered cell FDPE clocked by
sys_clk_pin  {rise@0.000ns fall@5.000ns period=10.000ns})
  Path Group:             sys_clk_pin
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            10.000ns  (sys_clk_pin rise@10.000ns - sys_clk_pin
rise@0.000ns)
  Data Path Delay:        3.414ns  (logic 0.773ns (22.642%)  route 2.641ns (77.358%))
  Logic Levels:           1  (LUT4=1)
  Clock Path Skew:        -0.039ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    4.847ns = ( 14.847 - 10.000 )
    Source Clock Delay      (SCD):    5.144ns
    Clock Pessimism Removal (CPR):    0.258ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

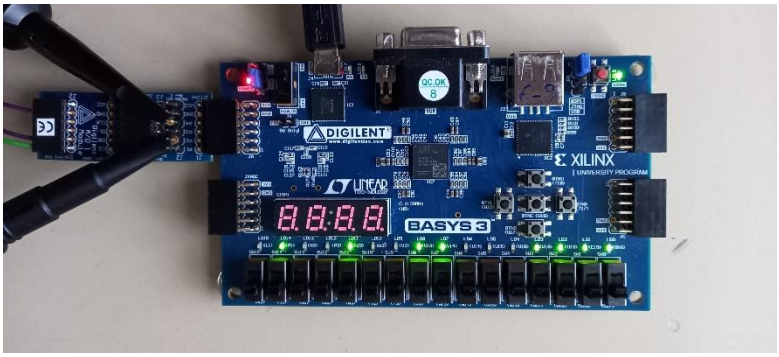


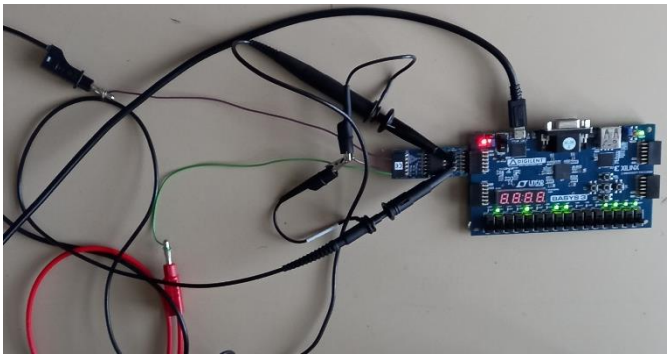

We can see here a drawing of the critical path. The source is a register from the state machine “FSM_sequential_current_state_reg[2]” and the destination is the reg_update register “reg_update_reg”.

## 4. Bonus

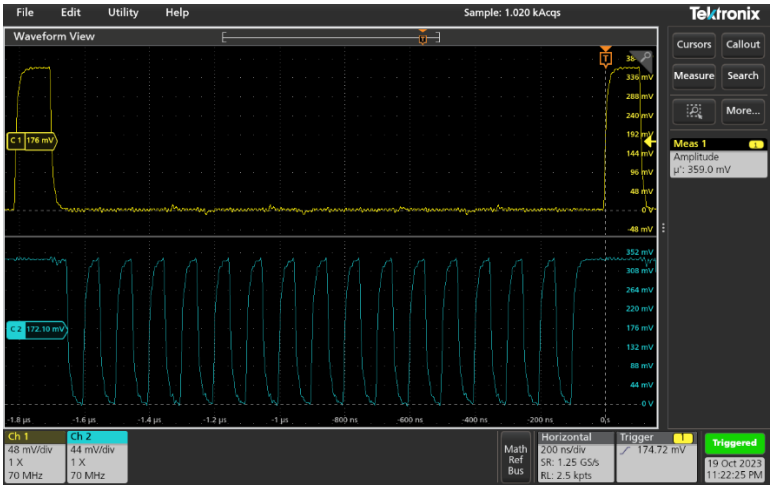


We set a DC voltage equal to 2.0 V on the power supply and. We use the oscilloscope to visualize first the “CS” and “CLK”pins of the PmodAD1. We connect the power supply and the oscilloscope probes to our PmodAD1 as shown in the following images:



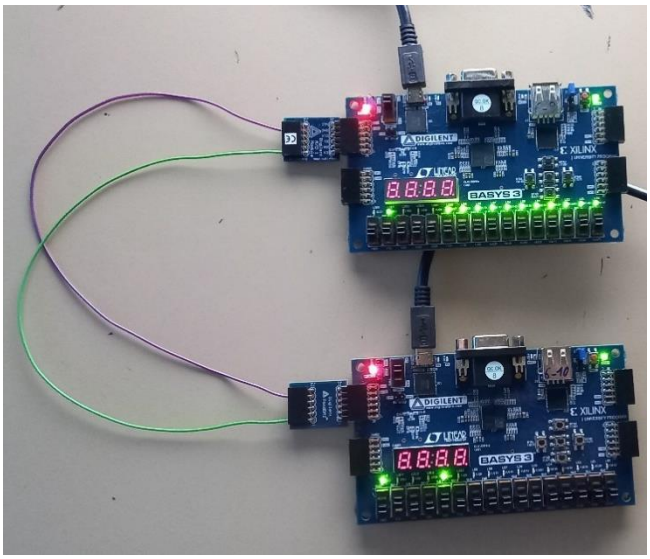
We read on the leds : 100110001111 (2) in binary which is equal to 1.97 V.
In the following figure, we can see our cs and sclk signals on the oscilloscope. The result is the same as from the timing diagrams in our test bench. When cs is set to 1, sclk is also set to 1. When cs is asserted to 0, sclk starts clocking and after 16 clock cycles, cs is de-asserted and sclk is set back to 1.



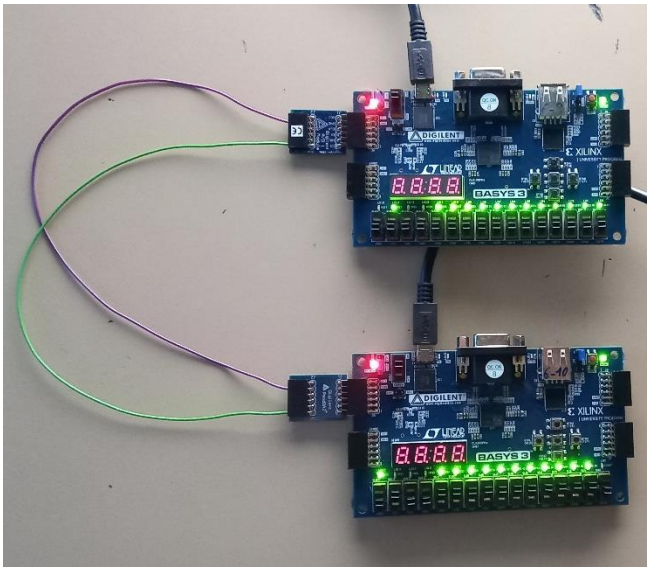

Now, we display the sclk signal with the sdata1 signal on the oscilloscope with the input analog voltage set to 2V. We can see in the following figure that for the first three rising edges of sclk, we receive the zeroes that are sent by the ADC at the beginning of each transmission as shown in Figure 1. We then receive the following message: 100110001001, which corresponds to a value of 1.96V.

In this part, we connect the DAC of another group to our ADC and we read the voltage set by the DAC. In the following photos, the DAC is the one on the bottom and the ADC is on top.

In the first Figure, The DAC outputs a voltage of 1.65V which corresponds to a value of « 100000000000 » in binary as we can see on the leds. The Leds on the ADC indicate a value of «011111111111» which is a value of 1.649V.




In the second Figure, The DAC outputs a voltage of 3.3V which corresponds to a value of « 111111111111 » in binary as we can see on the leds. The Leds on the ADC indicate a value of «111111111111» which is also 3.3V.



In the third Figure, The DAC outputs a voltage of 0V which corresponds to a value of « 000000000000 » in binary as we can see on the leds. The Leds on the ADC indicate a value of «000000100111» which is a value of 31 mV. This value is negligible and can be caused by some noise or electromagnetic interferences on the ADC.