DAC AD5541A Pmod Controller (VHDL)

Scott_1767 **DigiKey Employee** **Jul '21**

Logic Home

Downloads

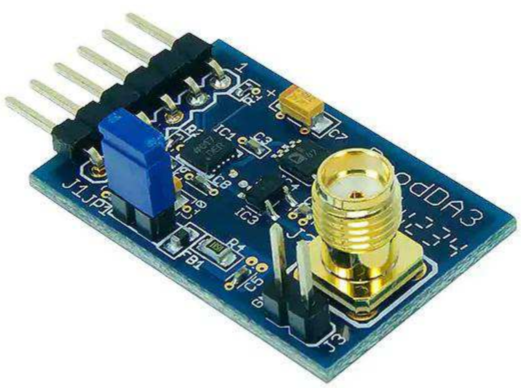
DAC AD5541A Pmod Controller (top-level file): **pmod_dac_ad5541a.vhd** (7.7 KB)
SPI Master (must also be included in project): **spi_master.vhd** (8.8 KB)

Features

- VHDL source code of a streamlined interface to **Digilent's Pmod DA3** (Pmod for Analog Devices **AD5541A** digital-to-analog converter)
- Accepts data to control the DAC using a simple parallel interface
- Includes update enable to optionally control timing of DAC output updates
- Handles all serial communication with the DAC Pmod
- Configurable system clock rate

Introduction

This details a VHDL component that handles interfacing to the Digilent's DAC AD5541A Pmod, shown in Figure 1. Figure 2 illustrates a typical example of this DAC Pmod Controller integrated into a system. As shown, the DAC Pmod Controller connects to the Pmod ports and executes transactions to set the DAC output. Data is latched in on a simple parallel interface which can be connected to user logic or to input ports on the FPGA. An update signal controls when the DAC outputs the data it receives.



I **Skip to main content** D5541A Pmod

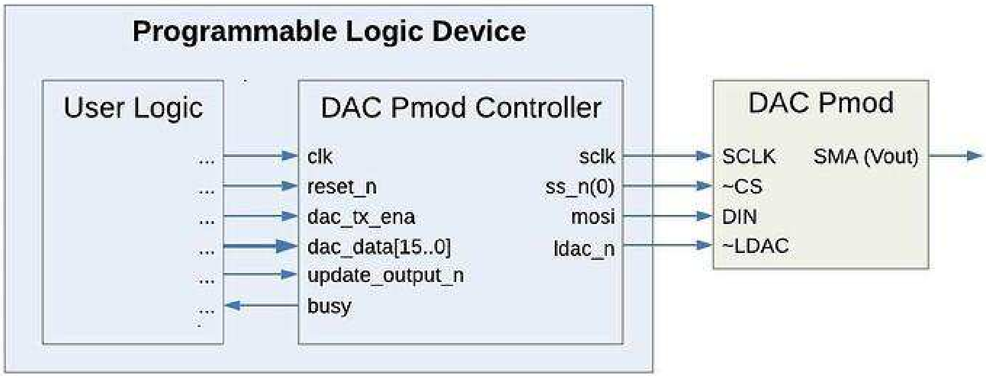


Figure 2. Example Implementation

Background

The DAC AD55541A Pmod provides a 1-channel, 16-bit digital-to-analog converter. The Pmod also includes a 2.5-V **ADR441** voltage reference. The 16-bit data resolution corresponds to approximately 0.038mV per bit.

Theory of Operation

The DAC Pmod Controller uses a simple state machine and the SPI Master component available on eewiki to load data into the AD5541A's data register. AD5541A's output is updated from its data register whenever the *update_output_n* port is '0'. Therefore, if the *update_output_n* port is held at '0', the output updates immediately whenever the data register updates. Alternatively, the *update_output_n* port can be pulsed low to control the precise timing of the output updates.

State Machine

The design uses the state machine depicted in Figure 3 to implement its operation. Upon start-up the component immediately enters the *start* state. It remains in this state for 100us to ensure the Pmod has ample time to power-up. It then proceeds to the *pause* state. Here, it ensures at least 20ns elapse between transactions with DAC (the AD5541A datasheet specifies a minimum of 15ns). It then deasserts the *busy* signal to indicate that the DAC Pmod Controller is ready for a new transaction with the DAC Pmod and proceeds to the *ready* state. It waits in the *ready* state until the *dac_tx_ena* enable signal is asserted, when it latches in the data for the new transaction and advances to the *send_data* state. In this state, it executes the transaction with the Pmod and then returns to the *pause* state. Although not shown, resetting the component at any time returns it to the *start* state.

Skip to main content

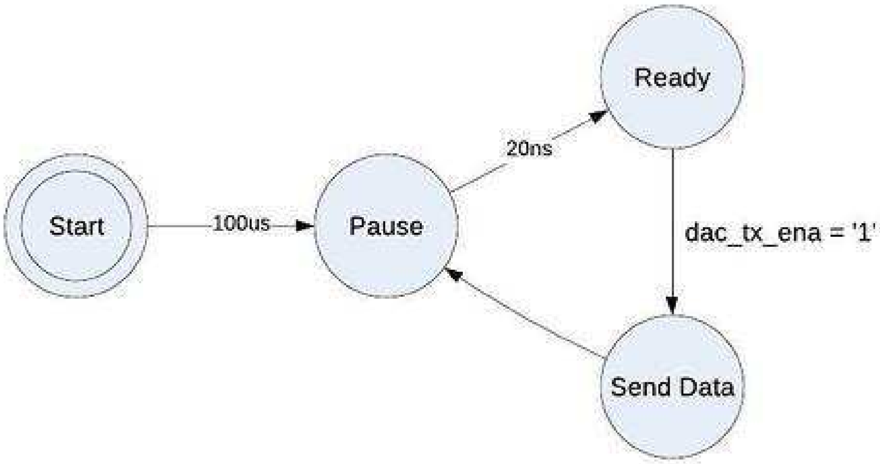


Figure 3. State Diagram

SPI Master

During the *send_data* state, the state machine controls an SPI Master component to communicate with the DAC on the Pmod. Documentation for the SPI Master is available **here**
.

The SPI Master is configured with CPOL = 0 and CPHA = 0, to meet the requirements of the AD5541A converter.

Configuring the Clock

The clocking of this DAC Pmod Controller is configured by assigning values to the GENERIC parameters *clk_freq* and *spi_clk_div*, defined in the ENTITY. The *clk_freq* parameter must be assigned the frequency of the system clock provided on the *clk* input port in MHz. Equation 1 defines how the *spi_clk_div* value is calculated.

$$(1) \qquad spi\_clk\_div = \frac{f_{clk}}{100} \qquad \textbf{(answer rounded up)}$$

where *fclk* is the frequency of the provided system clock in MHz.

For example, the default value specified in the code is *spi_clk_div* = 1. This is arrived at because the component was developed and tested using a system clock of 50 MHz. 50/100 = 0.5, rounded up is 1. Any *clk_freq* ≤ 100 MHz results in the default *spi_clk_div* = 1.

Equation 2 defines the serial clock frequency *fsclk* that results.

$$(2) \qquad f_{SCLK} = \frac{f_{clk}}{2 \cdot spi\_clk\_div}$$

This calculation keeps the serial clock below the DAC’s maximum specified communication f **Skip to main content** en powered by 3.3V. The fastest communication occurs when the input clock frequency (in MHz) is an integer multiple of 100.

Transactions

The DAC Pmod Controller indicates its availability on its *busy* output. When the *busy* signal is ‘0’, the Controller is ready to accept transactions to send to the DAC Pmod. Asserting the *dac_tx_ena* input latches in the current value of *dac_data*. Once latched, the Controller asserts the busy signal to indicate that a transaction is in progress, so it is not currently available. When the transaction is complete, it again deasserts the *busy* signal to indicate that it’s ready to accept another request.

Example Transaction

Figure 4 illustrates an example transaction. The *busy* signal is ‘0’. The user logic then asserts the *dac_tx_ena* signal to send the data presented on the *dac_data* bus to the DAC. The Controller asserts the *busy* signal, indicating the request is latched in, at which point the user logic can deassert the *dac_tx_ena* signal. The Controller sends the serial communication to the DAC Pmod, then deasserts *busy* when complete.

If the *dac_tx_ena* signal is not deasserted, a new transaction request is latched in and begins immediately once the Controller is available.

In this example, the user logic sets the *update_output_n* signal high when it requests to send new data to the DAC. Therefore, the DAC’s output does not update immediately upon receiving the new data. Instead, it updates once the *update_output_n* signal is deasserted (thereby deasserting the *ldac_n* signal to the DAC).

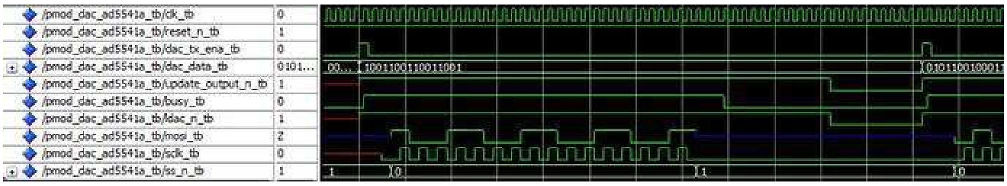


Figure 4. Transaction Example

Port Descriptions

Table 1 describes the DAC Pmod Controller’s ports.

Table 1. Port Descriptions

Port	Width	Mode	Data Type	Interface	Description
clock	1	in	standard logic	user logic	System clock
reset_n	1	in	standard logic	user logic	Asynchronous active low reset
dac_tx_en	1	in	standard logic	user logic	Transaction enable. H: latches in data, then performs a transaction with the DAC. L: no transaction is initiated.
dac_data	16	in	standard logic vector	user logic	Data to transmit to DAC
update_output_n	1	in	standard logic	user logic	Update output enable. H: DAC output does not update. L: updates the DAC output with the DAC's latest received data.
busy	1	out	standard logic	user logic	H: component is unavailable. L: component is ready to accept a new transaction.
ldac_n	1	out	standard logic	DAC Pmod	Update output enable signal to DAC
mosi	1	out	standard logic	DAC Pmod	Serial data output to DAC
			standard logic	DAC Pmod	Serial clock
			standard logic vector	DAC Pmod	Chip select

Connections

This Pmod has a 6-pin connector, J1. Table 2 provides the pinout for this connector. The DAC Pmod Controller's ports need to be assigned to the FPGA pins that are routed to this connector as listed.

Table 2. DAC Pmod Pinout and Connections to DAC Pmod Controller

Pmod Connector	Pmod Pin Number	DAC Pmod Port	DAC Pmod Controller Port
J1	1	~CS	ss_n
J1	2	DIN	mosi
J1	3	~LDAC	ldac_n
J1	4	SCLK	sclk
J1	5	GND	-
J1	6	VCC	-

Reset

The *reset_n* input port must have a logic high for the DAC Pmod Controller component to operate. A low logic level on this port asynchronously resets the component. During reset, the component aborts the current transaction with the DAC Pmod and sets the *busy* output high to indicate it is not available. Once released from reset, the DAC Pmod Controller restarts operation.

Conclusion

This DAC Pmod Controller is a programmable logic component that interfaces to Digilent's DAC AD5541A Pmod. It simplifies data transactions with the DAC and includes an update output enable.

Related Topics

SPI Master (VHDL)

Contact

Comments, feedback, and questions can be sent to **eewiki@digikey.com**.
