My name is Alexandr, currently I’m a PhD student in the Hong Kong Polytechnic University. My primary research area is 3D Imaging in optical microscopy and I’m trying, along with the many other researches all around the world, to develop and validate the optical microscope configuration and data processing techniques, allowing fast and reliable 3D imaging for label free intracellular studies, and designing instrumentation accessible to a wide range of research laboratories.

Strictly speaking, I’m not an optical engineer, even though I’m working with optics. I’d rather describe myself as an electronic engineer with some experience in radio frequency electronics and hardware design in application to radar technologies, which turned out to be very useful in the little project I’m going to talk about in this post.

In my research I focus on a modification of the Structured Light Illumination technique, which, as it might be clear from the name, uses some specific illumination patters to get more information about the object and reconstruct its image with better resolution than a Wide Field Microscope can do.

The key element in my optical setup is so called galvo-mirror, i.e. a mirror attached to a DC motor. The position of the mirror is proportional to external voltage applied to the motor. In my optical design this mirror plays the critical role, since it is the element that provides the sample with the structured illumination.

My ability to reconstruct the fine image depends on my knowledge of the current mirror position (i.e. angle of deflection of the mirror from the neutral position and hence the incidence angle of the light, illuminating the sample), as well as on the ability of the mirror to maintain it during the image acquisition.

Conceptually, the Structured Light Illumination Unit is depicted in Figure 1.

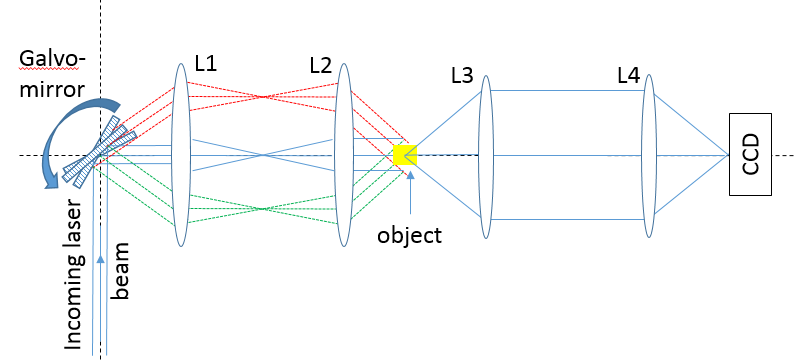


Figure 1. Structured Light Illumination Unit.

In this design the galvo-mirror is used to bend the light coming from the laser and redirect it onto the sample, using 2 telecentric lenses L1 and L2. The light’s path for the galvo-mirror being in neutral position (0°) is shown in blue solid line; the light’s paths for the two extreme angular positions ( and ) are shown in green and red dashed lines respectively. Lenses L3 and L4 along with CCD camera are used to capture the object’s image for every incident angle of illumination light.

During my experiment the galvo-mirror progressively changing its position from to with some increments and for its every position the image of the object is taken.

Figure 2 shows the analog command voltage waveform used to drive the mirror.

In this particular experiment it was desired to drive the galvo-mirror with the control signal, which amplitude changes from -7 to +7 V in 0.02 V increments. Galvo-mirror must reside in every position for the time sufficient for the CCD camera to acquire the image (flat shelves in in control sequence).

|  |  |
| --- | --- |
|  |  |
| 1. Command signal | 1. Command signal (zoom) |

Figure 2. Command signal.

However, the actual driving waveform, shown in Figures 3 and 4, is not as smooth as it was expected to be.

|  |  |
| --- | --- |
|  |  |
| 1. Actual command signal | 1. Actual command signal (zoom) |

Figure 3. Actual command signal



Figure 4. Voltage surges in actual command signal.

It is easy to see the 2 major problems with the actual command waveform from those Figures:

* Voltage surges, produced by function generator, when it switches between adjacent voltage levels;
* Non-uniformity in amplitude increments (see Figure 5).

Both problems are apparently typical for the function generators from the shelf, since I’ve tried 2 different devices from different vendors and both of them had similar problems. Possibly those function generators are simply not designed to produce an AC signal by means of updating DC signal, whereas in my application it is important to use DC control voltage to get a nice synchronisation between the galvo-mirror and CCD and avoid capturing of smeared images.

|  |  |
| --- | --- |
|  |  |
| 1. Actual increments in control signal | 1. Actual increments in control signal (zoom) |

Figure 5. Actual increments in control signal.

So, why are those problems with the actual control signal are not acceptable in my design?

First of all, the voltage surges produce huge output current that drive my mirror into the fault mode, which is absolutely not desired if I don’t want to buy the new one soon enough. Secondly, no image can be taken in the vicinity of those surges because during the fault mode the mirror goes into the one of the extreme positions and then recovers from it during some time after that. This leads to a few missing samples from the overall object’s image. Thirdly, I expect the function generator to update the driving signal with the giving increments of 0.02V, but in fact the actual increment can be something else, which means that the error is accumulated during the measurements and by the end of experiment I have no clue what is the actual mirror position and what is the actual incidence angle of my illumination. This may lead to the huge error of a few tenth of a degree, resulted in incorrect image reconstruction. I’d like to underline here the fact that I don’t really care about the exact value of increment, but I need to know it, in other words I need a feedback, which a simple function generator from the shelf cannot give me.

Finally, all the function generators I used in my design had non-uniform resolution over the whole span of the driving signal. For instance, they can update the output DC signal with minimum increments of 20mV for the span from to V and with 2mV increments between -1 and +1 V, which sometimes is not good enough for my application.

All those problems have finally lead me to a conclusion that a standard function generator could be used as a driver for the galvo-mirror in my design and I need to look for another solution to get an accurate experimental result.

The design criteria I formulated for myself sounded like:

* The future function generator must be driven by the PC (Matlab specifically), to give me an option to synchronise the camera and the galvo-mirror;
* To convert a digital command into the analog one I will need a digital-to-analog converter;
* To get the galvo-mirror position feedback signal I will need an analog-to-digital converter;
* To effectively transfer command signal from Matlab into the code acceptable for DAC and ADC I will need a programmable device, either a microcontroller or an FPGA board.

A choice between a microcontroller and an FPGA is an easy one for me. Compared to any microcontroller an FPGA board will always have a definite advantage of being faster and more flexible design option, since it’s simply a programmable hardware, capable of executing a few operations in parallel.

The next thing to decide is how the board is going to communicate with the Matlab. Another very simple to answer question. I had in hands Nexys 4 DDR board by Digilent, which has one remarkable feature (among the many others) - a shared UART/JTAG USB port, which can be used to supply power to the board from the standard USB port on your desktop. At the same time, it can be used as a USB-UART bridge – a very convenient option if you want to keep the number of wires running around your lab lower.

The last step in hardware selection is to pick suitable data converters for my design. Ideally, a DAC should be able to supply an output voltage within V range and have a 2^16 bit resolution for the fine sampling of command waveform. Similarly, an ADC module must meet the DAC’s requirement to work with it in unison.

At that moment I didn’t have proper data converters, so I had to order them, but what I had was a couple of Pmods from Digilent – DA4 and AD2, so I could use them to verify my design concept while waiting for the delivery of their big brothers.

So, here is a brief summary of my design resources:

* PC with Matlab and USB port (DTE – Data Terminal Equipment);
* FPGA board [Nexys 4 DDR](http://store.digilentinc.com/nexys-4-ddr-artix-7-fpga-trainer-board-recommended-for-ece-curriculum/) board (with UART/JTAG USB port and couple of Pmod ports);
* DAC [PmodDA4](http://store.digilentinc.com/pmod-da4-eight-12-bit-d-a-outputs/);
* ADC [PmodAD2](http://store.digilentinc.com/pmod-ad2-4-channel-12-bit-a-d-converter/);
* USB A to micro-B cable;
* [2x6 Pin cable](http://store.digilentinc.com/2x6-pin-cable/);
* 2 [2x6 headers](http://store.digilentinc.com/2x6-pin-header-5-pack/).

**UART bridge design considerations**

The first thing to do in UART design is to go at the FTDI website and download and install a suitable for your operation system [Virtual COM Port driver](http://www.ftdichip.com/Drivers/VCP.htm) that will allow you to establish communication between on-board FTDI FT2232HQ USB\_UART bridge with DTE as if the latter was a standard RS232 device. Once it is done, the virtual port can be opened in Matlab, configured and used for data transfer.

The next thing to consider is the port settings. You can find a nice guidance on the baud rate selection in the following [UART User Guide](https://training.ti.com/system/files/docs/keystone-UART-user-guide.pdf?keyMatch=UART%20UG&tisearch=Search-EN). The suitable values of the Baud Rate and corresponding Divisor Values are found using expression (1), given in the mentioned above document and listed in Table 1 (the maximum on-board clock frequency is 100MHz).

. (1)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Baud Rate** | **Divisor Value** | | **Actual Baud Rate** | | **Error (%)** | |
| **16x** | **13x** | **16x** | **13x** | **16x** | **13x** |
| 9600 | 651 | 801 | 9600.614 | 9603.38 | -0.006 | -0.035 |
| 14400 | 434 | 534 | 14400.922 | 14405.071 | -0.006 | -0.035 |
| 19200 | 326 | 401 | 19171.779 | 19182.812 | 0.147 | 0.09 |
| 38400 | 163 | 200 | 38343.558 | 38461.538 | 0.147 | -0.16 |
| 57600 | 109 | 134 | 57339.45 | 57405.281 | 0.452 | 0.338 |
| 115200 | 54 | 67 | 115740.741 | 114810.562 | -0.469 | 0.338 |
| 230400 | 27 | 33 | 231481.481 | 233100.233 | -0.469 | -1.172 |
| 460800 | 14\* | 17 | 446428.571 | 452488.688 | 3.119 | 1.804 |

\* - not recommended.

Table 1. Example Baud Rate and Divisor Values for 100MHz clock.

Other Baud Rate values are not recommended for this design, since lower rates will make your system too slow and the higher ones (921600 bits/s for instance) have big error of more than 3 and 4% for 16x and 13x divisors respectively, which will result in the data transfer errors.

The other port settings for this design are:

* No Parity check;
* 1 Stop bit;
* Byte Order is Little Endian;
* Flow Control is None.

It was decided to choose Little Endian as a Byte Order since in this case the Least Significant Bit of the Least Significant Byte will be transferred first and the Most Significant Bit of the Most Significant Byte will be the last, i.e. the data will be transferred in the reverse order (from right to left) and it will be very straight forward to handle it at the hardware receiver/transmitter.

Flow control could be set as “Hardware” or “None”, but the latter option will give me more freedom as it will allow you to have full control over RTS and CTS signals.

Since both data converters have 12-bit resolution, duration of the single data package will be 16 bit: 12 data bits with 4 leading zeros and, hence, the input and output buffers’ capacity (in Matlab) is going to be 2 bytes.

The last thing to bear in mind is that the binary read/write operations do not require terminators and therefore make the data package smaller.

A comprehensive overview of the [serial port](http://www.mathworks.com/help/instrument/serial-port-overview.html), [binary read](https://www.mathworks.com/help/matlab/ref/serial.fread.html) and [binary write](https://www.mathworks.com/help/matlab/ref/serial.fwrite.html) operations could be found at the Mathworks website, here, only a quick summary will be given.

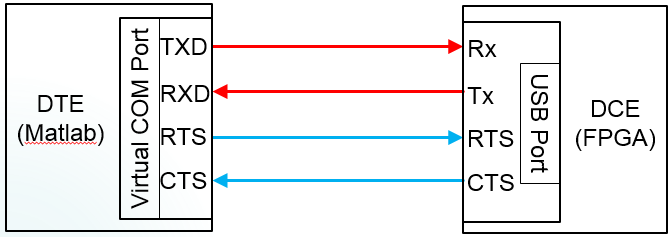


Figure 6. Communication over a serial port.

In our case, only 4 signals are used to establish communication and data exchange between DTE and DCE (Data Circuit-Terminating Equipment, i.e. FPGA). The red colour is used to mark the data lines – transmission and reception. Note that DTE’s transmitter is connected to the DCE’s receiver and vice versa, but the lines are named from the DTE’s point of view. Blue colour is used to mark the control signals – “Request-To-Send” and “Clear-To-Send”. When DCE is ready to receive the data it pulls CTS LOW, if CTS remains high transmission of data from DTE to DCE is impossible. If DTE RTS is LOW, DCE may send data to the DTE.

Hence, using these 2 control lines, it is easy to synchronise data transmission between the equipment.

**Function generator design considerations**

Figure 7 below shows the FSM (Finite State Machine) diagram for the custom function generator.

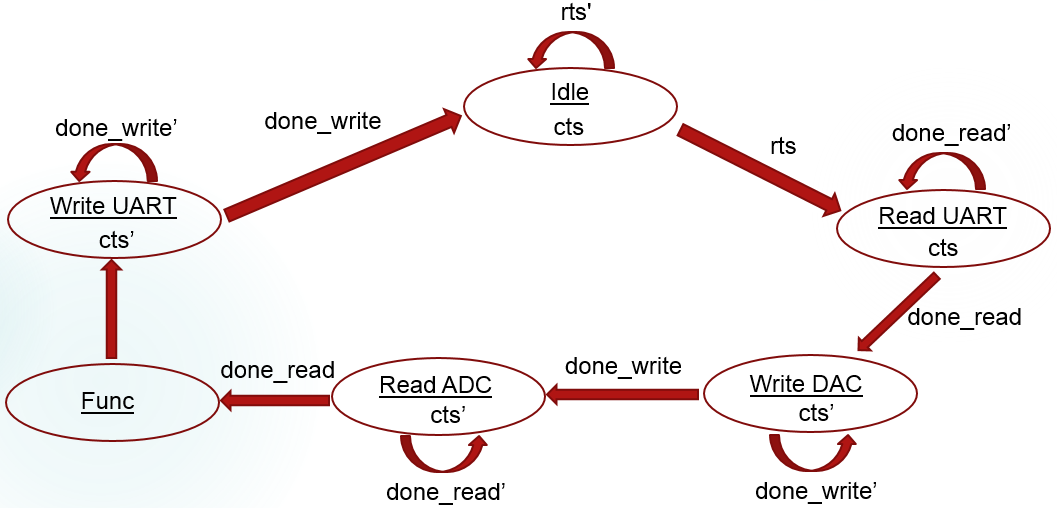


Figure 7. Top-level FSM for custom function generator.

Since it is necessary to synchronise the image capturing with the galvo-mirror positioning all operations in DTE and DCE are going to be executed sequentially, in the fashion similar to the one shown above.

Initially the system is resting in the “Idle” state waiting for the RTS signal to go LOW, which will mean that the serial port is open and the DTE is ready to communicate. When RTS is set LOW the state machine goes into the “Read UART” state and pulls CTS LOW, signalling to the DTE that the data transmission can be started. In fact, “Read UART” state as well as the other states in Figure 7 (except for “Idle” one) hide low-level FSMs that handle all necessary operations for that particular state and will be considered in greater details later.

When operation of receiving data from the DTE is over, DONE\_READ\_UART flag is asserted and FSM moves into the next state “Write DAC”, where the control signal, received from the DTE, is written into the DAC. DONE\_WRITE\_DAC flag indicates completion of this operation and sends the FSM in to the next state “Read ADC”. In this state ADC acquires the actual level of the command analog signal supplied to an external device. DONE\_READ\_ADC flag is asserted when this operation is over and the FSM may proceed to the “Write UART” mode. In the last state the data received by ADC is sent back to the DTE via UART and the flag DONE\_WRITE\_UART is set HIGH, sending the FSM into the “Idle” state to wait for the next command signal to be transmitted. Note, in every state, except for “Read UART” mode, CTS signal is HIGH, blocking any possible attempts from the DTE to send the next data package until the previous one is being processed.

It is also necessary to mention one important part of the design, which is not shown in Figure 7, namely a Baud Clock Generator.

Its function is to produce a Baud tick signal, i.e. a pulse having duration of one period of the board clock () with the pulse repetition frequency of , which is used for read and write UART operations and synchronizing ADC and DAC state machines with the top-level one.

**“Read UART” state**

Low-level “Read UART” FSM diagram is shown in Figure 8 below.

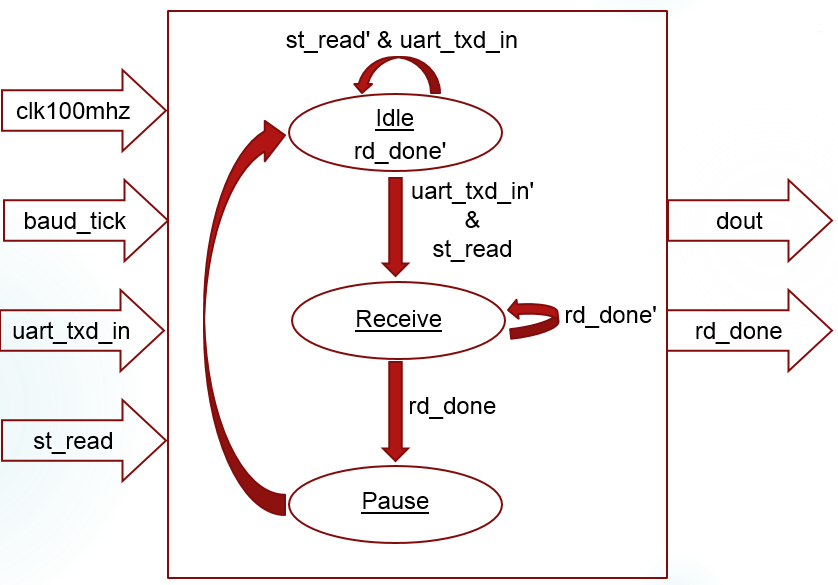


Figure 8. Low-level “Read UART” FSM.

The top-level FSM moves into the “Read UART” state once the serial port is configured and opened in DTE.

The low-level “Read UART” FSM, handling reception of data, has 4 input and 2 output signals:

* clk100mhz and baud\_tick are the clocking signals, the former is the system clock and the latter is the Baud clock;
* uart\_txd\_in line is used to send data from DTE to DCE;
* st\_read is the flag, being set HIGH by the top-level FSM, activates the low-level “Read UART” one;
* rd\_done is the low-level flag, indicating the end of the read operation; it is passed to the top-level FSM, such that the function generator can shift into the next state and start writing DAC;
* dout is the data register to store the information received from the DTE, it is also passed to the top-level FSM and used in writing DAC operation.

The low-level “Read UART” FSM has 3 states. It remains “Idle” until the STAR\_READ flag is asserted by the top-level FSM and START bit appears on the transmission line. “Receive” state is responsible for the data reception, storing it into the data register and asserting the RD\_DONE flag. The last state “Pause” simply deasserts the RD\_DONE flag and returns FSM into the “Idle” state.

Let us take a look now at how actual data is being transmitted over the UART bridge.

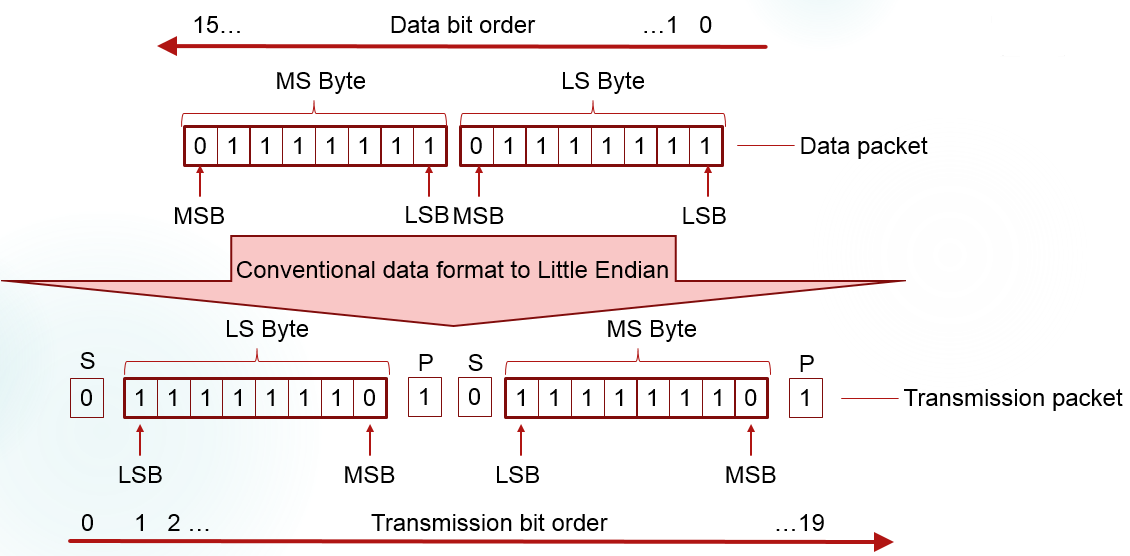


Figure 9. Data transmission over UART bridge.

For example, if we want to transmit a 2-byte data packed “01111111 01111111” using the settings, described above, the 16-bit packet will be converted into the 20-bit one, because a START (S) bit (‘0’) will be added before the LSB bit of every byte and a STOR (P) bit will be appended at the end of every byte. Because a Little Endian format is used, the packet will be sent in the reverse order (see Figure 9).

Figure 10 demonstrates the principles of operation of the “Read UART” FSM, where a 2-byte data packet “11111111 11111111” is being transmitted.



Figure 10. Transmission of a packet between DTE and DCE.

Let us take a detailed look at how the process of reading data is executed in hardware.

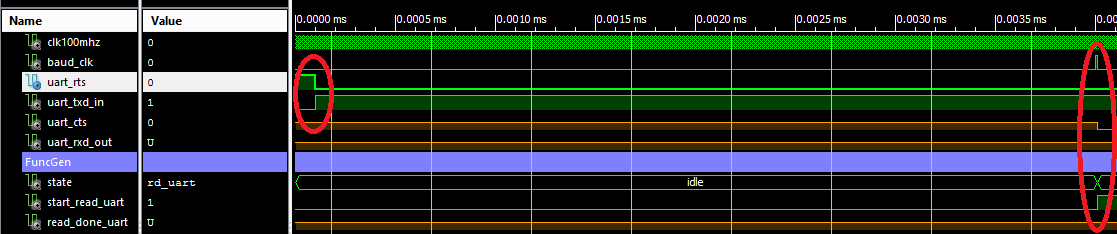


Figure 11. RTS signal detection.

Once the DTE is ready to send the data, it asserts RTS signal (about 100ns after beginning of simulation). The read and write UART operations are controlled by the Baud clock signal (baud\_clk) and the states of the UART lines are sampled on the rising edge of the Baud clock signal. Hence, once the Baud tick appears (about 4 us), the change in RTS line state is detected; the top-level FSM asserts START\_READ\_UART flag and goes into the “Read UART” state.

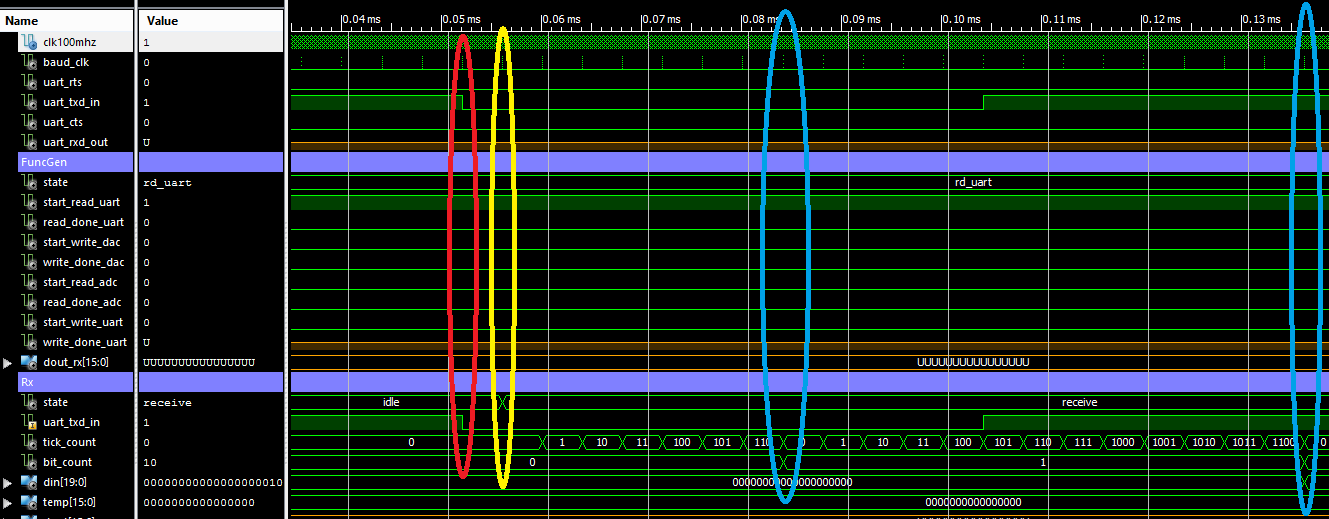


Figure 12. Start bit detection and transmission line sampling.

Low-level “Read UART” FSM continues to remain in the “Idle” state even after the ST\_READ flag was asserted, awaiting for the HIGH-to-LOW transitions of the Tx line, which will appearance of the START bit on the line (about 52 us). Once the START bit is detected (about 56 us), “Read UART” FSM goes into “Receive” state and starts counting the number of Baud ticks issued since the beginning of the transmission.

In the current example 13x divider is used, therefore, every transmitted bit has duration of 13 Baud ticks. To avoid an error, normally the value on the Tx line about the middle tick is regarded to be the true value of the bit being transmitted. Thus, the value of the first bit, measured at the 7th Baud tick (about 84 us) is ‘0’, i.e. a START bit is detected and stored into the data register. The next measurement is done in 13 Baud ticks after the previous bit detection. The first actual data bit being transmitted is detected and stored at about 136 us.

Two counters are used to control the process, one counts the number of Baud ticks, the second one counts the number of bits read and it is incremented every time when the first counter overflows.

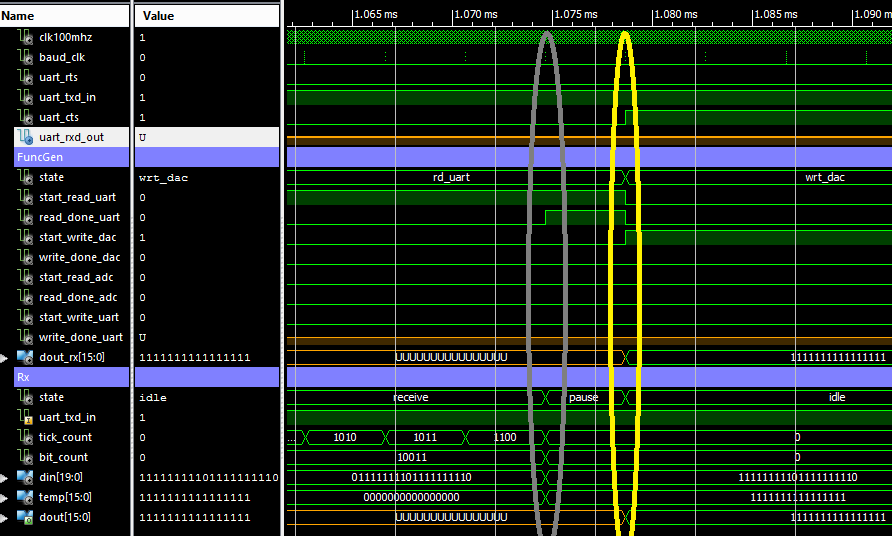


Figure 13. Completion of “Read UART” process.

Once all 20 bits has been received and stored (in the reverse order! such that the first bit received becomes the LSB) into the data register “din”, the whole content of the “din” register, except for the START, STOP bits and the “dummy” zeros we may have appended to the data package in Matlab, is being saved into the “temp” register. At the same time the READ\_DONE\_UART flag is set HIGH.

The low-level FSM moves into the “Pause” state, where it deasserts the flag and passes the data from “temp” register to the output “dout”.

The top-level FSM having detected the READ\_DONE\_UART flag clears STAR\_READ\_UART flag, sets CTS HIGH (to prevent transmission of the next packet), set flag START\_WRITE\_DAC HIGH and shifts into the next “Write DAC” state, while the low-level “Read UART” FSM returns to the “Idle” mode.

**“Write DAC” state**

Low-level “Write DAC” FSM diagram is shown in Figure 14 below.

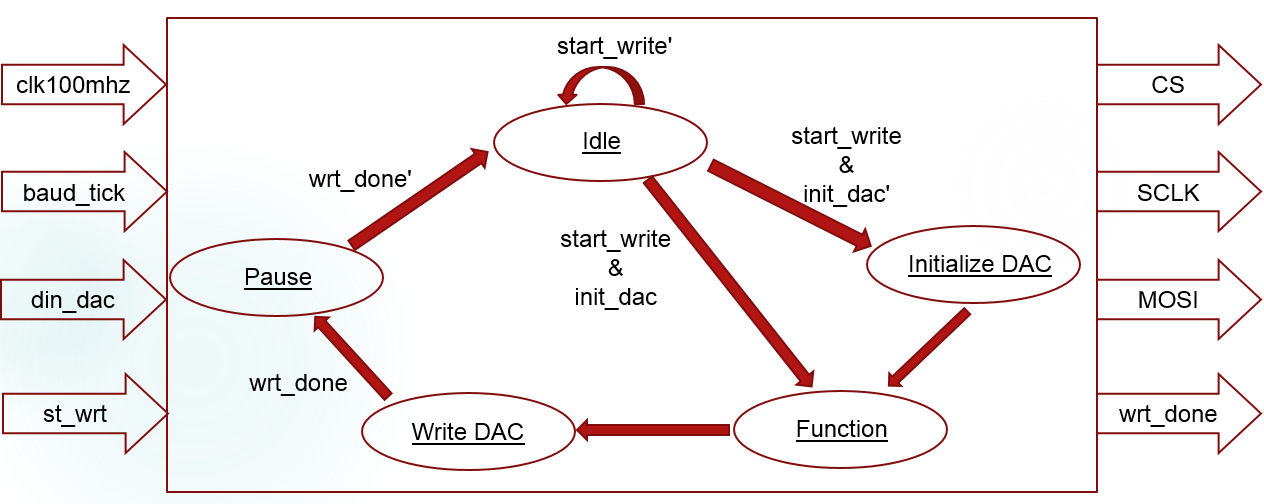


Figure 14. Low-level “Write DAC” FSM.

The top-level FSM moves into the “Write DAC” state once the read operation of UART is completed.

The low-level “Write DAC” FSM, is responsible for writing the command signal, received from the DTE, to the Digital-to-Analog Converter; it has 4 inputs and 4 output signals:

* clk100mhz and baud\_tick are the clocking signals, the former is the system clock and the latter is the Baud clock;
* st\_write is the flag, being set HIGH by the top-level FSM, activates the low-level “Write DAC” one;
* din\_dac is the command signal received via UART and to be written to the DAC;
* wrt\_done is the low-level flag, indicating the end of write operation; it is passed to the top-level FSM, such that the function generator can shift into the next state and start reading ADC;
* CS is the chip select signal, must be set LOW when DAC is written;
* SCLK is the output clock signal to drive DAC;
* MOSI is the serial DAC’s data line.

The low-level “Write DAC” FSM remains in the “Idle” state until the START\_WRITE flag is not set HIGH by the top-level FSM. If it is going to be the first start of the DAC after power-up it needs to be initialized. INIT\_DAC flag keeps trace of whether it’s been initialized yet or not. If this flag is LOW than “Write DAC” FSM goes into “Initialize DAC” mode.

At this stage it will be very useful to refer to the datasheets of the [DAC](http://www.analog.com/media/en/technical-documentation/data-sheets/AD5628_5648_5668.pdf) and [PmodDA4](http://store.digilentinc.com/pmod-da4-eight-12-bit-d-a-outputs/).

To both initialize and write DAC a 32-bit long command word is used. In this design the DAC will be initialized with 0x08000001 value.

The first 4 bits are unused; the next 4 bits are Command Bits and 0x8 configuration tells DAC to use internal reference voltage of 2.5V.

The following 4 bits are Address bits and setting them to 0x0 we are telling the DAC to do conversion on the channel A.

The next 12 bits are the data bits, but they play no role in initialization procedure, so we set them to zero.

The 12 lest significant bits are don’t care bits in write and initialize operations, except for the LSB, which is don’t care in write mode and either 1 or 0 in initialization process, depending on whether the internal or external reference used, respectively. Since we want to use internal reference, it is set to ‘1’ for initialization.

Example timing diagram for the device might be found in the data sheet and for this reason the simulation results, shown in Figures 15-18, are not explained in details here.

To write the initialization word to DAC, the CS signal set LOW and the content of the set-up register is written into the MOSI line bit-by-bit at the rising edge of the SCLK signal. A counter keeps trace of the number of bits written.

Once the initialization procedure is over, the “Write DAC” FSM goes into the “Func\_DAC” state, which copies the content of the din\_dac signal (command signal from DTE) into the corresponding data bits of the 32-bit command word. For the write operation Command bits are set to 0x3, which tells DAC to write to and update DAC channel *n*, set by the Address bits 0x0 and pointing at channel A. The last 8 bits and the first 4 bits of the command word are set to zero, as they are don’t care for the write operation.

At the next rising edge of the SCLK “Write DAC” FSM shift into “Wrt\_DAC” state, the content of the command word is written into the MOSI line, in the fashion similar to initialization procedure.

Once it is done, the WRT\_DONE flag is set HIGH and the low-level FSM goes into the “Pause” mode where it waits for the synchronization Baud clock pulse. At the rising edge of the Baud\_tick WRT\_DONE flag is cleared and the “Write DAC” FSM returns to the “Idle” state. Simultaneously, the top-level FSM clears the START\_WRITE\_DAC flag and shifts into the “Read ADC” state, asserting START\_READ\_ADC flag HIGH.

Note, that even though the DAC remains idle between the active operations it maintains the output voltage in accordance with the last command word written into it, since the data remains latched in the DAC register.

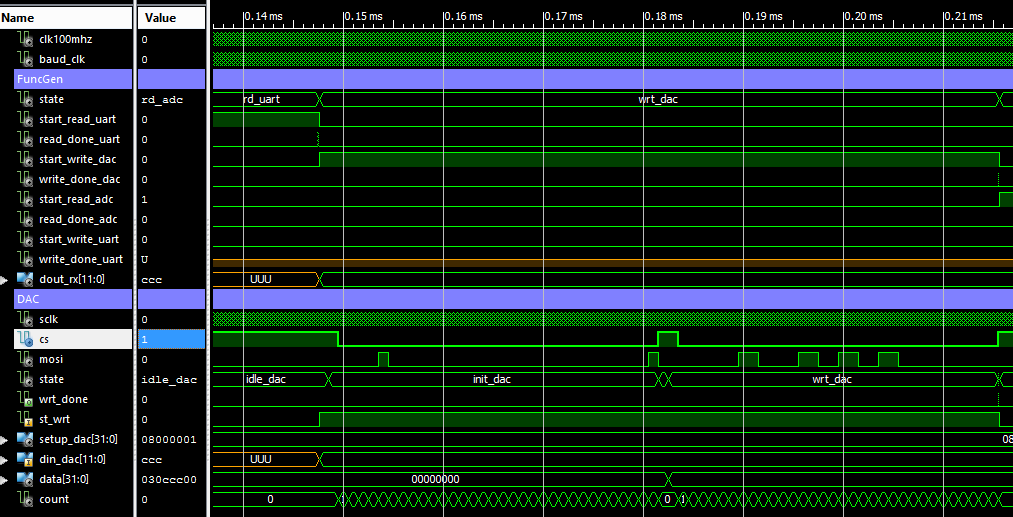


Figure 15. Timing diagram for “Write DAC” FSM.

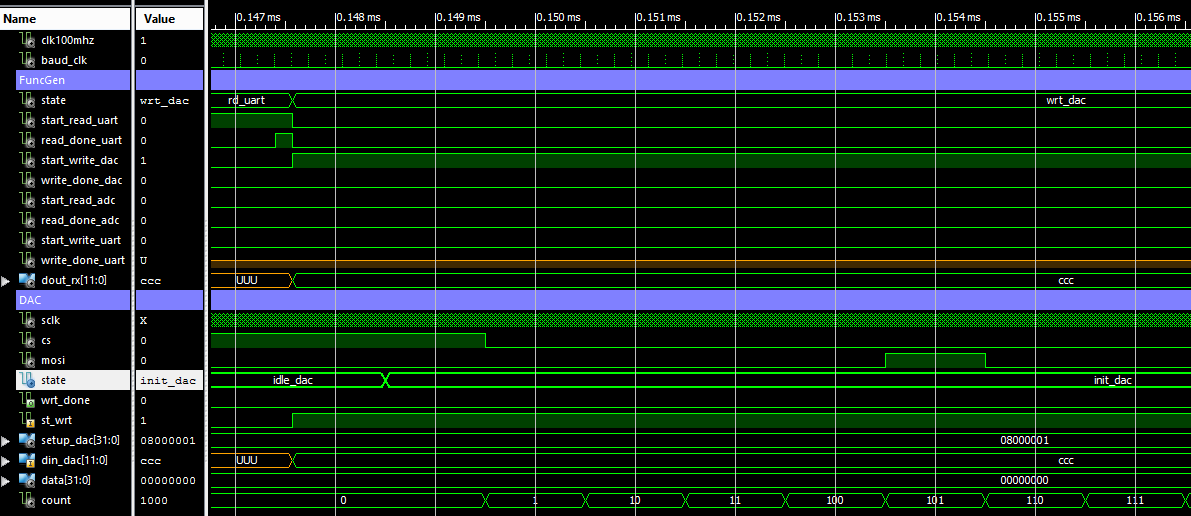


Figure 16. Timing diagram for “Write DAC” FSM (zoom on “Idle”-“Init\_DAC” transition).

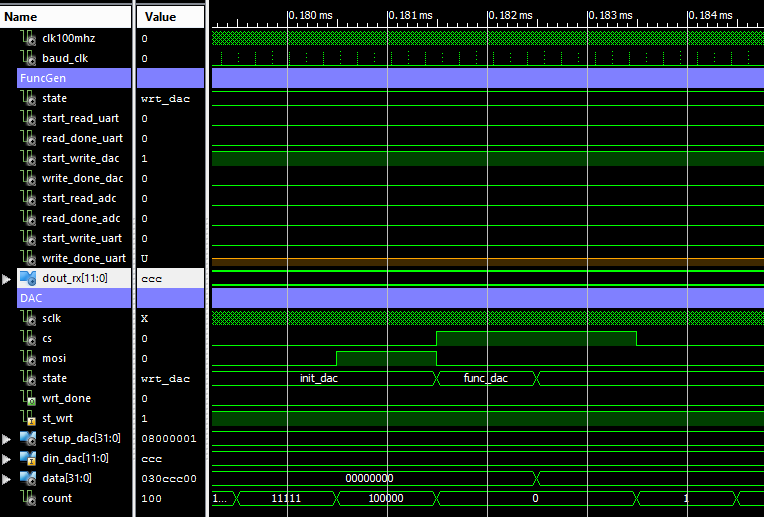


Figure 17. Timing diagram for “Write DAC” FSM (zoom on “Func\_DAC” state).

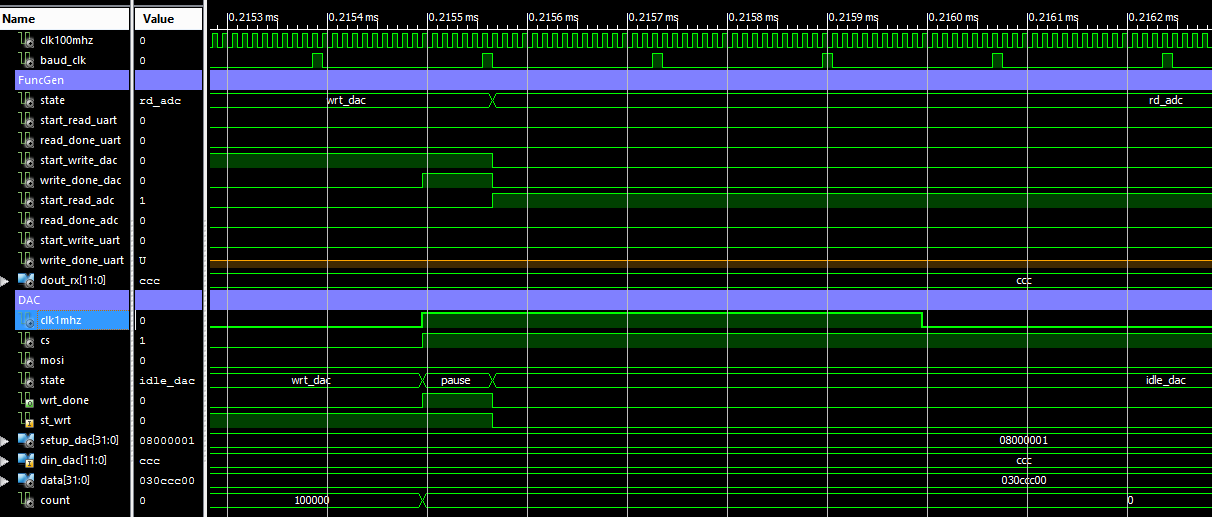


Figure 18. Timing diagram for “Write DAC” FSM (zoom on “Pause” state).

**“Read ADC” state**

Low-level “Read ADC” FSM diagram is shown in Figure 19 below.

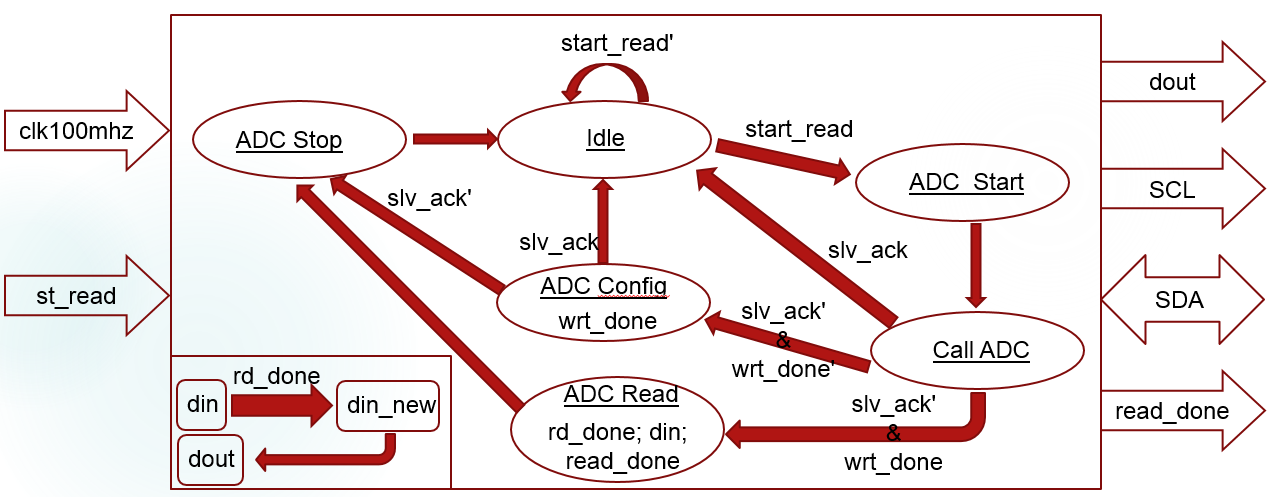


Figure 19. Low-level “Read ADC” FSM.

The top-level FSM moves into the “Read ADC” state once the write operation of DAC is completed.

The low-level “Read ADC” FSM, is responsible for reading the analog command signal supplied by the Function Generator to an external device (i.e. provides feedback); it has 2 inputs, 3 outputs and one bidirectional signal:

* clk100mhz is the clocking signals from the system clock;
* st\_read is the flag, being set HIGH by the top-level FSM, activates the low-level “Read ADC” one;
* read\_done is the low-level flag, indicating the end of the read operation; it is passed to the top-level FSM, such that the function generator can shift into the next state and start writing UART;
* SCL is the output clock signal to drive ADC;
* dout is the feedback signal read by ADC and supplied to the top-level FSM;
* SDA is the bidirectional ADC’s data line.

Before we consider the technical details of operating ADC let us say a few words about general settings of the data converter. Again, it will be very helpful to refer to the datasheets of [PmodAD2](http://store.digilentinc.com/pmod-ad2-4-channel-12-bit-a-d-converter/) and ADC [AD7991](http://www.analog.com/media/en/technical-documentation/data-sheets/AD7991_7995_7999.pdf).

Since we want both DAC and ADC to use the same reference voltage – 2.5V it is necessary to provide the ADC with the external reference, otherwise it will use Vcc (3.3V) signal supplied by the board. A simple voltage divider and on-board Vcc power pin will do the job (see Figure 20.b)).

It is also necessary to set the content of the Configuration register properly. In our case, we want to convert data on channel 1 (Vin0), which is done by setting 4 upper bits of the control word to 0x1 and we want to use external reference, provided through the channel 4 (Vin3). The other setting we leave unchanged, having the lower 4 bits of control word as 0x8. So, the whole control word is 0x18. Note, it is also necessary to set jumper JP1 to “V4” as shown in Figure 20.a).

|  |  |
| --- | --- |
|  |  |
| 1. Jumper settings | 1. Reference voltage |

Figure 20. Setting external reference.

Because ADC uses the I2C serial bus interface, the Data Line SDA is shared between 2 devices – Master (FPGA board) and Slave (ADC AD7991). Since 2 different devices are driving the same bus, we attach Master to SDA through a tri-state buffer, to avoid any possible damage or loss of the data, and implement a pullup on the data line, such that SDA will always be attached to a weak logic HIGH signal when neither Master nor Slave drive it.

Any read or write operation begins with issuing a START condition, when SDA line is pulled LOW while SCL is HIGH. Then a corresponding Slave device must be addressed by the Master. In our case, the Slave’s address is 0b0101000 (7 bit). The 8th bit (LSB) in the address byte is either read or write bit (1/0). Thus, if we want to write (configure) ADC the address byte is 0b01010000, else, for the read operation, the ADC must be called using the following address 0b01010001. Slave device acknowledges reception of the command by pulling SDA line LOW. Note that SDA line must be written only during the low period of SCL and the data bit must remain stable during the high period of the clock, otherwise it may be interpreted as START or STOP condition.

Any device that has been addressed by another device must acknowledge it by pulling SDA line LOW.

In case when the Master calls for the Slave, the latter issues acknowledgement.

Once the Slave device has been called and acknowledged it a Configuration word can be written into it and another acknowledge must be issued by Slave. After that, a STOP condition is asserted by Master, by pulling SDA line HIGH during the HIGH period of SCL line.

ADC needs to be configured only once after it’s been powered up. When configuration procedure is completed, ADC can be read immediately upon another call from the Master using the read address (0b01010001) and reception of acknowledgement bit from Slave.

During the read operation SDA line is being driven by Slave, which outputs information in 8-bit portions. The first 8 bits comprise 4 status bits (2 leading zeros and 2 channel identifier bits), followed by 4 data bits. Next Master must acknowledge the reception of the first packet by pulling SDA LOW and the remaining 8 data bits from Slave will follow. To stop ADC and put it into shut-down mode Master issues no acknowledge upon reception of the 2nd byte and asserts STOP condition, which will close a single conversion cycle.

A single data conversion cycle is shown in timing diagram in Figure 21.



Figure 21. Timing diagram for “Read ADC” FSM.

There are many different ways to implement I2C protocol, but for this design it was done in a way similar to DDR technique. 2 different clock signals were generated, the 1st one – slow clock (100kHz) – drives SCL line; while the 2nd one – fast clock (200kHz) – is used for read/write SDA line. Both read and write are done on the rising edge of the fast clock and both clocks are mutually displaced (by 500 ns in this case) to make sure that rising edges of the fast clock occur when the slow clock is completed its transition and remains in either HIGH or LOW state.

With such clock arrangement the read and write operations as well as issuing START and STOP conditions are done as follows:

* START – HIGH-to-LOW transition of SDA line on the Rising edge of the Data (fast) clock when SCL is HIGH;
* STOP – LOW-to-HIGH transition of SDA line on the Rising edge of the Data (fast) clock when SCL is HIGH;
* Write SDA - on the Rising edge of the Data (fast) clock when SCL is LOW;
* Read SDA - on the Rising edge of the Data (fast) clock when SCL is HIGH.

Figures 22-25 illustrate those concepts.

The signal SDA\_prime, shown in the Figures below, is the Master’s signal attached to SDA line through a tri-state buffer, it drives SDA line during Write, START and STOP operations.

In Figure 25, SDA line is being driven by Slave and it is possible to notice that SDA\_prime (Master’s signal), detached from SDA by tri-state buffer, remains HIGH regardless of actual SDA state.

Figure 26 illustrates how internal and external flags (RD\_DONE and READ\_DONE) are issued upon completion of a single cycle by “Read ADC” FSM.

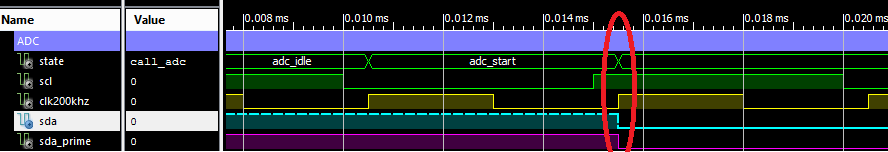


Figure 22. START condition

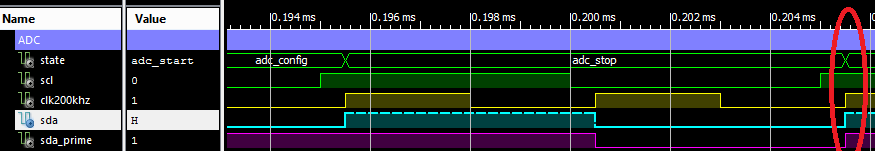


Figure 23. STOP Condition

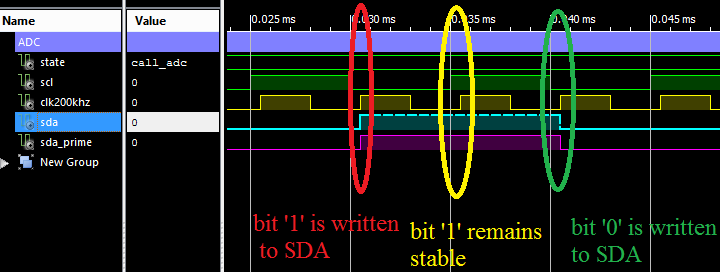


Figure 24. Write SDA

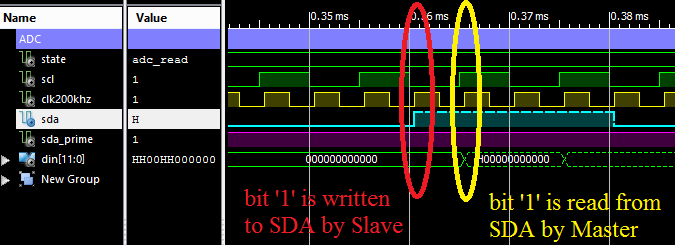


Figure 25. Read SDA

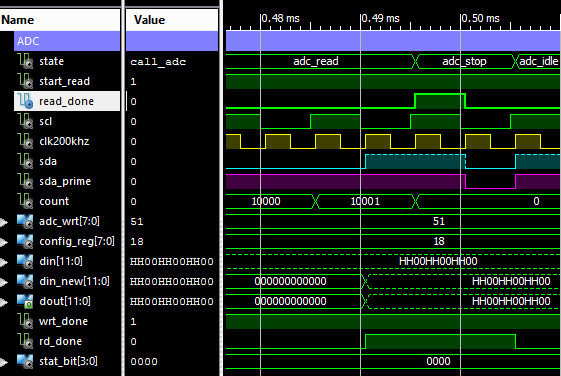


Figure 26. Completion of “Read ADC” FSM operation.

Internal flag RD\_DONE is used to update output signal “dout” in the separate process inside the “Read ADC” FSM. External flag READ\_DONE is signalling to the top-level FSM that read ADC operation is completed and Function Generator may go into the next states “Func” and “Write UART”.

**“Func” state**

This is a single clock cycle state that simply converts the data read by ADC into transmission format, appending Start and Stop bits along with leading zeros to the data packet (see Figure 9).

**“Write UART” state**

Low-level “Write UART” FSM diagram is shown in Figure 27 below.

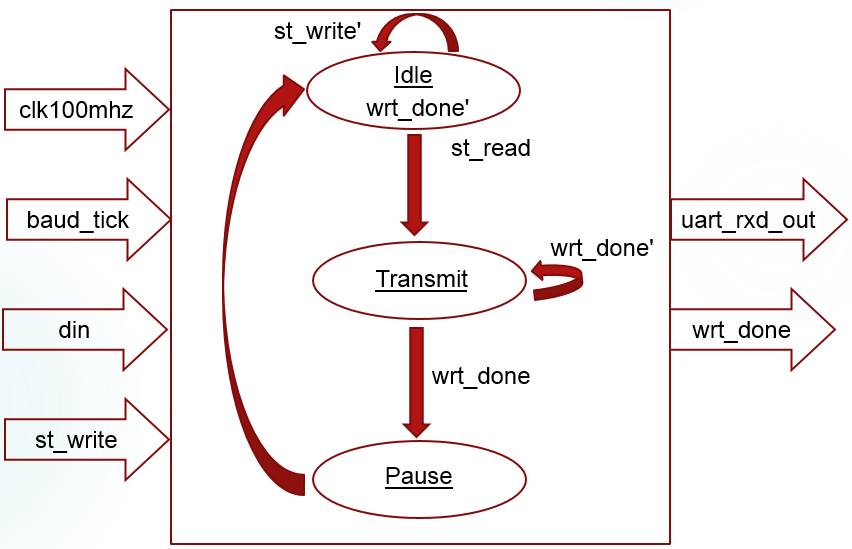


Figure 27. Low-level “Write UART” FSM

The low-level “Write UART” FSM, is responsible for sending back to the DTE the digital feedback signal read by the ADC; it has 4 inputs and 2 outputs:

* clk100mhz and baud\_tick are the clocking signals;
* uart\_rxd\_out line is used to send data from DCE to DTE;
* st\_write is the flag, being set HIGH by the top-level FSM, activates the low-level “Write UART” one;
* wrt\_done is the low-level flag, indicating the end of the write operation; it is passed to the top-level FSM, such that the Function Generator can return into “Idle” state waiting for the next transmission;
* din is the input data, read by ADC, and to be transmitted to the DTE.

Similar to the other low-level FSM’s the “Write UART” one remains in the “Idle” state until the ST\_WRITE flag is set HIGH by the top-level FSM.

During transmission procedure, the each bit, being currently transmitted, is simply held on the uart\_rxd\_out line during the number of Baud\_ticks (13 or 16 ticks). As before, the order of the data packet transmission is defined by the Little Endian format, i.e. the LSB of the Least Significant Byte is sent first.

Once the last bit is transmitted, the WRT\_DONE flag is set HIGH and transmission line is pulled HIGH as well. After that both FSM’s – “Write UART” and “FuncGen” – return into “Idle” mode.

**Display**

The 5 low-level FSM’s working together under the aegis of the top-level one comprise the fully functional core of the Function Generator, but it would not be a complete product without a display.

Hence, we are going to add to our design another FSM, responsible for displaying the current level of the analog signal. The ADC’s readings will supply the data for displaying.

Thus, we have on hands a 12-bit binary number and 8 Seven Segment Displays (SSD) on board. The task is to convert a binary number to a Binary Coded Decimal (BCD), which can be easily displayed. The process of driving a SSD is thoroughly explained on pages 18-20 of [Nexys 4 DDR datasheet](https://reference.digilentinc.com/_media/nexys4-ddr:nexys4ddr_rm.pdf).

Before any binary to BCD conversion it is necessary to translate a 12-bit logic vector representing an analog signal into actual voltage.

Since we used an external reference of 2.5V () and ADC has a 12-bit resolution (N) than the LSB is equal to:

, ([2](http://www.analog.com/media/en/technical-documentation/data-sheets/AD5628_5648_5668.pdf))

.

To find what actual voltage level is represented by a binary number we need to multiply this number by the LSB first.

For example, if the binary number produced by the ADC after measuring the analog voltage level is 0b110011001100, than after multiplication we will find that it corresponds to 1.99836V (3276\*610e-6).

This operation looks easy in decimal notation, but we are going to do it in binary format.

The LSB’s value in binary notation will be: 0b0.0000 0000 0010 1 (1 bit for integer part and 13 bits for fractional part). Binary number representing the voltage can be written as 0bXXXX XXXX XXXX (12 bits for integer part, 0 bits for fractional part). After multiplication in binary format we will have a 26-bit long number like this: 0bX XXXX XXXX XXXX.XXXX XXXX XXXX X (13 bits in integer and 13 bits in fractional part).

We know that the maximum possible value of the analog signal is 2.5V and minimum one is 0V. Since there are only 8 SSDs on the board, we must dedicate the first digit to the integer part of the number, representing analog voltage and 7 digits will be reserved for the fractional part.

Given that the highest possible integer number representing the analog voltage is 2, we may keep only 2 least significant bits in integer part and discard the rest without losing any precision.

Since we dedicated 7 SSD’s for displaying the fractional part, the smallest number we can display is 2-7=0.007812510 (check, there are 7 digits after decimal point in decimal format) and as a result we must truncate the fractional part to keep only 7 most significant digits.

The result now looks like: 0bXX.XXXX XXX. Truncation of the fractional part will result in the loss of precision, but not significant, thus the maximum displayable number will be 0b10.0111111=2.492187510≈2.49795V (the maximum voltage which ADC can measure).

The next step in our design is to convert integer and fractional parts into Binary Coded Decimal format, suitable for displaying on SSD.

Let us consider conversion of integer numbers first, using a 0b1111 1111 binary number as an example.

First we need to determine the number of BCD digits to display this value. 0b1111 1111=25510 and therefore we require 3 digits, every BCD digit is encoded using 4 bits from 0b1001 to 0b0000. Thus the size of the shift register is 12 bits.

In theory conversion procedure looks like that. We start right shifting the binary number bit by bit into the BCD shift register. After each shift operation we need to check whether the content of any shift register exceeds 910 and if so, we adjust it by subtracting 1010 from the content of the current shift register and adding 1 to the content of the senior BCD register.

In practice there is an easier way to implement such conversion.[[1]](#footnote-1)

We shift binary number all the same, but check if it is greater than 410 and if so, we add 310 to the content of BCD register, the carry-out will do the increment of the higher level BCD register automatically. Table 2 below illustrate the whole process.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| BCD shift register | | | Binary number | | Description of operation |
| BCD digit 2 | BCD digit 1 | BCD digit 0 | 4 MSB | 4 LSB |
| 0000 | 0000 | 0000 | 1111 | 1111 | Start |
| 0000 | 0000 | 0001 | 1111 | 1110 | Right shift |
| 0000 | 0000 | 0001 | 1111 | 1110 | Check, no adjust (4>1)10 |
| 0000 | 0000 | 0011 | 1111 | 1100 | Right shift |
| 0000 | 0000 | 0011 | 1111 | 1100 | Check, no adjust (4>3)10 |
| 0000 | 0000 | 0111 | 1111 | 1000 | Right shift |
| 0000 | 0000 | 0111 | 1111 | 1000 | Check, need to adjust (4<7)10 |
| 0000 | 0000 | 1010 | 1111 | 1000 | Adjust |
| 0000 | 0001 | 0101 | 1111 | 0000 | Right shift |
| 0000 | 0001 | 0101 | 1111 | 0000 | Check, need to adjust (4<5)10 |
| 0000 | 0001 | 1000 | 1111 | 0000 | Adjust |
| 0000 | 0011 | 0001 | 1110 | 0000 | Right shift |
| 0000 | 0011 | 0001 | 1110 | 0000 | Check, no adjust (4>3;4>1)10 |
| 0000 | 0110 | 0011 | 1100 | 0000 | Right shift |
| 0000 | 0110 | 0011 | 1100 | 0000 | Check, need to adjust (4<6)10; no adjust (4>3)10 |
| 0000 | 1001 | 0011 | 1100 | 0000 | Adjust |
| 0001 | 0010 | 0111 | 1000 | 0000 | Right shift |
| 0001 | 0010 | 0111 | 1000 | 0000 | Check, no adjust (4>1;4>2)10; need to adjust (4<7)10 |
| 0001 | 0010 | 1010 | 1000 | 0000 | Adjust |
| 0010 | 0101 | 0101 | 0000 | 0000 | Right shift |
| 2 | 5 | 5 |  |  | Stop. No adjust, because nothing more to shift and all BCD registers < 910 |

Table 2. Integer Binary to Binary Coded Decimal conversion.

As you can see, we successfully converted a binary number 0b1111 1111 (25510) into its Binary Coded Decimal counterpart 0b0010 0101 0101.

Similarly we may convert the fractional part into Binary Coded Decimal. The only difference is, now we are going to left shift the data and adjust the BCD registers if their content is greater than 710 by adding 1310 (or subtracting 310) and discarding the carry-out.[[2]](#footnote-2) Table 3 illustrates conversion of the binary number 0.11111 into BCD format.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Binary number | BCD shift register | | | | | Description of operation |
| BCD 1 | BCD 2 | BCD 3 | BCD 4 | BCD 5 |  |
| 0.11111 | 0000 | 0000 | 0000 | 0000 | 0000 | Start |
| 0.01111 | 1000 | 0000 | 0000 | 0000 | 0000 | Left shift |
| 0.01111 | 1000 | 0000 | 0000 | 0000 | 0000 | Check, need to adjust (7<8)10 |
| 0.01111 | 0101 | 0000 | 0000 | 0000 | 0000 | Adjust |
| 0.00111 | 1010 | 1000 | 0000 | 0000 | 0000 | Left shift |
| 0.00111 | 1010 | 1000 | 0000 | 0000 | 0000 | Check, need to adjust (7<8;7<10)10 |
| 0.00111 | 0111 | 0101 | 0000 | 0000 | 0000 | Adjust |
| 0.00011 | 1011 | 1010 | 1000 | 0000 | 0000 | Left shift |
| 0.00011 | 1011 | 1010 | 1000 | 0000 | 0000 | Check, need to adjust (7<8;7<10;7<11)10 |
| 0.00011 | 1000 | 0111 | 0101 | 0000 | 0000 | Adjust |
| 0.00001 | 1100 | 0011 | 1010 | 1000 | 0000 | Check, need to adjust (7<12;7<10;7<8)10; no adjust (7>3)10 |
| 0.00001 | 1001 | 0011 | 0111 | 0101 | 0000 | Adjust |
| 0.00000 | 1100 | 1001 | 1011 | 1010 | 1000 | Left shift |
| 0.00000 | 1100 | 1001 | 1011 | 1010 | 1000 | Check, need to adjust (7<12;7<9;7<11;7<10;7<8)10 |
| 0.00000 | 1001 | 0110 | 1000 | 0111 | 0101 | Adjust |
| 0.00000 | .9 | 6 | 8 | 7 | 5 | Stop |

Table 3. Fractional Binary to Binary Coded Decimal conversion.

As you can see, we successfully converted a binary number 0b0.11111 (0.9687510) in to its Binary Coded Decimal counterpart 0b0.1001 0110 1000 0111 0101.

Once conversion for the both integer and fractional parts is done, the resultant BCD number can be easily displayed on the Seven Segment Displays.

A separate process responsible for updating the Display is run simultaneously with the top-level FSM, effectively utilising the capability of FPGA for parallel data processing.

This simple display for the Function Generator concludes the project. The test results are given in the next section.

**Function Generator test**

Figure 28 shows the test results for the UART Tx-Rx data channel. There is no data converters attached to the board, UART transmission line is directly connected to the UART receiver line in hardware to see if there are any errors during communication process.

|  |  |
| --- | --- |
|  |  |
| 1. Tx vs Rx signals | 1. Error between Tx and Rx signals |

Figure 28. UART test

UART core was tested for all Baud Rates given in Table 1 for both divisor values. It is error free for any mode except for 16x 460800 bits/s, since for the given board clock frequency it is impossible to generate a proper Baud clock for the given divisor value.

Figure 29 shows the results for the complete Function Generator with data converters connected to each other (no external equipment, loop in hardware), tested at Baud Rate of 460800 bits/s with 13x divisor value. Figure 30 shows error overlay for the different cycles of command signal.

|  |  |
| --- | --- |
|  |  |
| 1. Tx-DAC vs ADC-Rx | 1. Error |

Figure 29. Function Generator test



Figure 30. Error overlay

From Figure 30 it can be seen that the error patterns repeats itself continuously, which means that the system is very stable. On the other hand, the error value is not the same for the different command signals, which is, in fact, the result of loading effect from the DAC. Since ADC must be driven by a low impedance source, it will be beneficial to place an input buffer amplifier (see [datasheet](http://www.analog.com/media/en/technical-documentation/data-sheets/AD7991_7995_7999.pdf)), but unfortunately I didn’t have any low-power one on hands to do so.

However, this little project can be considered successful, since it allowed me to verify the Function Generator’s concept and implement it using the “bigger” data converters with a very few modifications.

1. Pong P. Chu, “FPGA Prototyping by VHDL Examples”, 2008, pp. 147-148, Wiley and Sons, USA [↑](#footnote-ref-1)
2. H.Singh, M.Manohar,A.C.Gupta, “16 Bit Fractional Binary to Digital Medium Speed Converter with 4-Digit Display”, 1973, IETE Journal of Research, ISSN:0377-2063 [↑](#footnote-ref-2)