FPGA Course (4)

Paul Goossens

Welcome back to the beginner-friendly course we run in support of our extremely popular FPGA Development System. This month we examine the simulation capabilities of Quartus. Simulation makes it a lot easier to design circuits and track down errors in your designs. The accompanying examples show how you can use the audio interface of the prototyping board.

It's handy to be able to test your design during the design process. VHDL allows you to create test benches, which make it a lot easier to test and simulate VHDL designs.
Unfortunately, Quartus does not support VHDL test benches. It has a graphic simulator instead. Although the simulator provides less user functionality, it is easier to use. The simulator is more than adequate for most of the sample applications in this course.

Virtual

The simulator is actually a combination of a virtual signal generator and a logic analyser. Here we use it to simulate the operation of an audio interface implemented with the hardware.

Codec

The 'ex10' sample application described in this instalment uses the audio codec (IC12) on the prototyping board. From the schematic diagram, you can see that a 12.288-MHz clock signal is applied to this IC. The clock signal is also routed to an I/O pin (B12) of the FPGA. Unfortunately, that was not shown in the original schematic diagram.

The data transfer clock (BCKIN) is supplied by the FPGA. This clock signal must be synchronised to the 12.288-MHz clock signal.

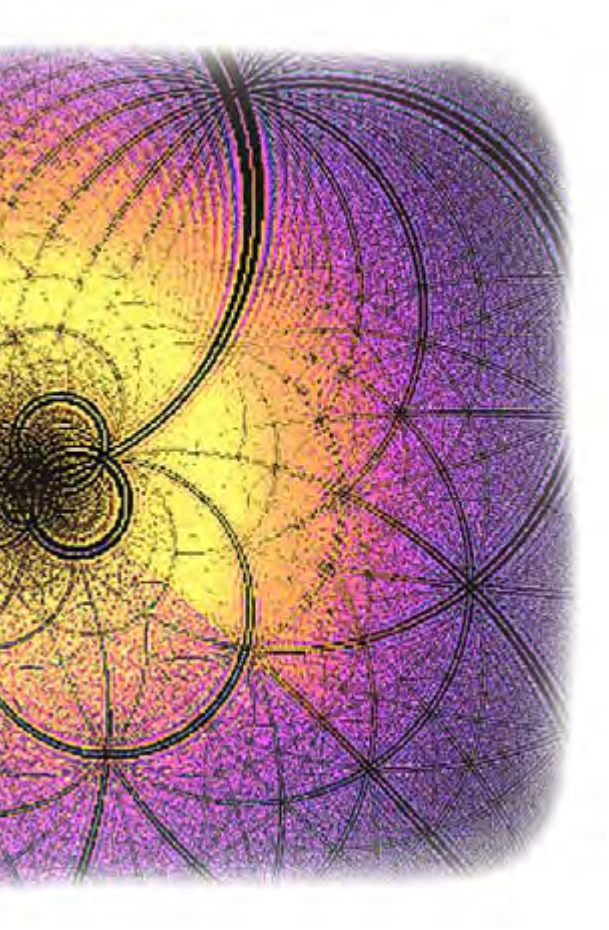
The timing diagram for data transfers between the codec and the FPGA is shown in **Figure 1**. As you can easily see, it takes 156 clock pulses of the CODECCLK signal to transfer a set of samples for the two channels.

The frequency of that signal on the prototyping board is 12.288 MHz. If you divide that figure by 256, you arrive at a sampling rate of 48 kHz.

Counter

All communications are synchronised to this clock. The simplest approach is to first create a counter that counts how many pulses of the CODECCLK signal have been generated. This signal is assigned the name '12_288MHz' in our design.
This counter is also represented by the COUNT signal in the PCM3006.VHD file, which is declared on line 54 as an 8-bit unsigned number. This signal can thus take on values the range of 0–255, which is exactly what we need.
We let this number increment in synchronization with the CLK signal (12.288 MHz).
The NEW_COUNT signal always contains the value that COUNT must assume on the next clock pulse. This is done by the following line:

```
NEW_COUNT <=
(COUNT+1) MOD 256;
```



Part 4: Simulation

This line is not in a process, so the function is evaluated each time COUNT changes.
The value of NEW_COUNT is loaded synchronously into COUNT in line 71.

Simulation

Now you can test this code using the Quartus simulator. For that purpose, we created a simulation file named 'ex10-1.vwf'. If you open this file, you will see several signals in the left-hand column and plots of the input signals versus time on the right.
Before you can use this file, you must configure Quartus so the simulator can use the file.

To do that, select 'Settings' in the 'Assignments' menu. In the new window that appears, select 'Simulator Settings'. Enter the file name 'ex10-1.vwf' in the Simulation Input box.
The simulation starts as soon as you select 'Start Simulator' in the 'Processing' menu. The result of the simulation (**Figure 2**) is displayed after the simulation is completed.
What matters at this point is the COUNT signal. As you can see from the simulation results, this counter is indeed incremented by 1 each time a rising edge occurs on the 12.288-MHz signal line. Note that COUNT has been changed to a 7-bit number due to optimization. Later on we'll explain why that is possible. What matters now is that the counter is synchronized to the clock.

Alias

The next step is to generate the data transmission clock (BCKIN). This signal must be low for two clock intervals of the main clock signal, after which it must be high for two clock intervals. That corresponds exactly to the third bit of the COUNT signal. This signal (called BCKOUT in the VHDL file) can thus be used at the output without any further processing.

The same holds true for the LRCK signal. This signal must be low for the first 128 clock pulses and then high for the following 128 clock pulses. That corresponds exactly to the most significant bit (highest-order bit) of the COUNT signal. That means you can use bit 7 of the COUNT signal as LRCKOUT.
The new status of LRCKOUT can be defined using the following line:

```
NEW_LRCKOUT <=
NEW_COUNT(7);
```

Another way to do this is to use the 'alias' keyword, which allows a signal to have more than one name. If you write

```
ALIAS NEW_LRCOUT : STD_LOGIC is
NEW_COUNT(7);
```

you can use the signal NEW_LRCOUT in the rest of the source code. The compiler will know that this signal is identical to NEW_COUNT(7).

Synchronous

Data bit reception is synchronised to the rising edge of BCK. The signal POSEDGE_BCK is used to detect the rising edge of BCK. It must indicate whether the BCK signal changes from low to high on the next rising edge of the system clock.

That requires knowing the current status of BCK and the status after the next clock pulse. These signals are COUNT(2) and NEW_COUNT(2), which are also assigned the names BCK_INT and NEW_BCK by alias statements in lines 62 and 63.
The POSEDGE_BCK signal is generated by the following line:

```
POSEDGE_BCK <= NEW_BCKOUT
AND (NOT BCKOUT_INT);
```

Data must be sent on the falling edge of BCK as shown in **Figure 2**. The signal NEGEDGE_BCK is generated in a similar manner for his purpose.

Glitches

Now it's time to look at these new signals in more detail. First configure the settings to have the simulator use the file *ex10.vwf*, and then start the simulation.

You will see the signals NEGEDGE_BCK, POSEDGE_BCK and LRCIN in the simulation results. The last signal of this group is the same as

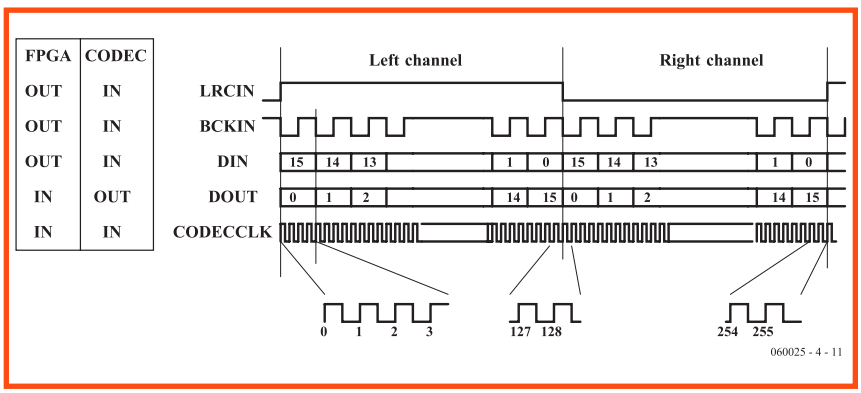


Figure 1. Timing diagram for data transfers between the codec and the FPGA.

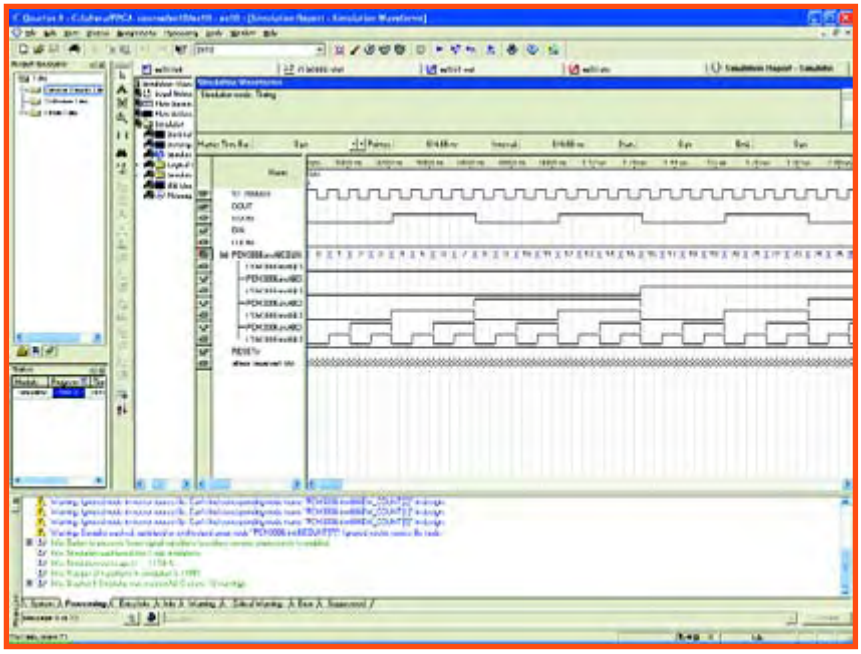


Figure 2. Data is transmitted on the falling edge of BCK, as can be seen from the simulation.

the LRCOUT signal in the VHDL design.
The POSEDGE_BCK signal and its counterpart NEGEDGE_BCK are generated using combinational logic. That means these signals are not synchronised by flip-flops. The disadvantage of this is that these signals can briefly assume an incorrect level if the input signals have different path delays. That phenomenon appears in the simulation in the form of short pulses. The technical term for these short pulses is 'glitches'.

Shift registers

The incoming and outgoing data are read in and 'pushed out' by shift registers. On each rising edge of the BCK signal, the contents of the SHIFTIN register are shifted left by one position. The incoming data is stored in bit 0.
The transmit shift register, SHIFTOUT, operates in a similar manner. It shifts the bits by one position on each falling edge of BCK. The most significant bit of this shift register is also the serial data output.

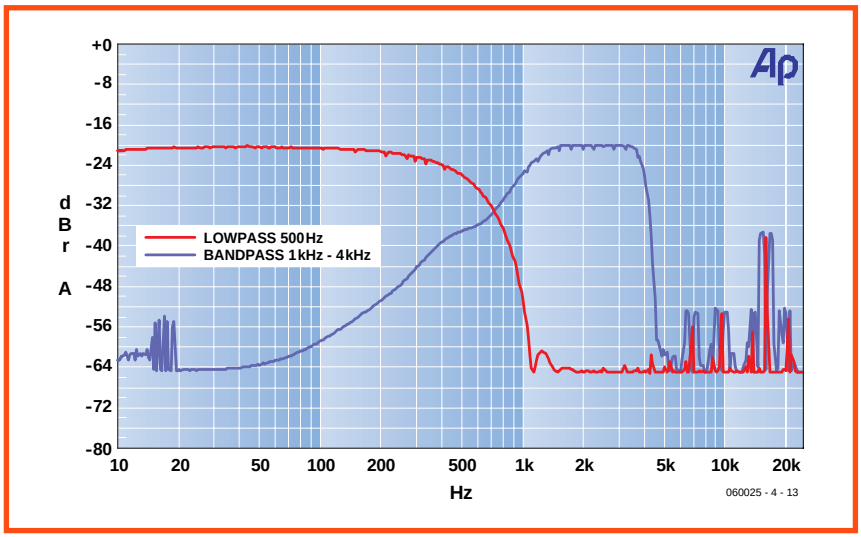


Figure 3. The characteristics of the various filter sets.

Each time a new set of samples must be transmitted, this shift register is loaded using the R_IN and L_IN signals. The contents of the receive shift register are also loaded into the LEFT_OUT and RIGHT_OUT registers.

Interface

At the same time, the NEW_SAMPLE output is set high for one clock interval. This signal indicates that a new set of samples has arrived. The peripheral logic can use this signal to process the new data.

Data must be applied to the inputs of the RIGHT_IN and LEFT_IN inputs in order to be transmitted. A high level at the LOAD input causes the data on the inputs to be stored, and it will then be sent with the next transmission.

Example

In the example, you can see that the data outputs are connected directly to the associated data inputs. The NEW_SAMPLE output signal is connected to the LOAD input.
This causes the received samples to be sent back to the codec on the next transmission. In other words, the signals at the inputs appear unchanged at the outputs after a short delay.

Another simulation

Overall operation of the circuit is illustrated by the file *ex10-3.vwf* . In this simulation, we used a random bit pattern for DOUT. The simulation clearly shows that this bit stream appears at the DIN output after approximately 31 µs. The received data is thus sent back to the codec without any modification.

As you can easily see, several signals are missing in this simulation. The reason for this is that these signals 'disappeared' in during compilation, because the compiler attempts to generate a design that is compact a possible. As a result, certain signals may become redundant, and the compiler will not implement these signals in the FPGA. As a result, Quartus cannot simulate the signals during the simulation session.

Filter

The bypass function described above is not particularly useful in practice. A

more useful technique is to process the input signal and then output the results via the codec.

The example file *ex11* implements an audio filter. Communication with the codec takes place via the previously described PCM3006.VHD core.
The samples are processed in the FIR.VHD code segment. The filter operates on the FIR principle. 'FIR' stands for 'finite impulse response'. Filters of this sort are often used in digital signal processing.

In a FIR filter, the passband characteristics are determined by a set of parameters called coefficients. These coefficients can be modified as desired in Quartus. Just as in the previous instalment, the Memory Content Editor is the tool for that purpose. The coefficients are stored in the memory segment named 'COEF'. The memory segment named 'IN' holds the most recent 128 audio samples.
Several hex files with coefficients are available for the project so you can easily try out various filters. The characteristics of the various filter sets are shown in **Figure 3**.

Signal generator

The final example, *ex12*, implements a simple sinewave generator. It products sinusoidal signals on the outputs. The signal on the right channel lags the signal on the left channel by 90 degrees.

The sinusoidal signals are generated using an arithmetic unit that can compute sine and cosine values. This unit needs an angle (*phase*) and an amplitude (*mag*) for this purpose. The unit then calculates the corresponding *X* (cosine) and *Y* (sine) values.

The arithmetic unit uses the CORDIC algorithm, which is suitable for calculating goniometric functions using simple operations. CORDIC has the unique property that it multiplies the length of each vector by approximately 1.645. That means you have to ensure that the results fit within the range of a 16-bit signed number, so the input value must not exceed 4DDO. If you use a larger value, the resulting sinewave will be highly distorted.

Signal

To obtain a sinusoidal signal, a constant value must be applied the *mag* input. In addition, the phase must be increases slightly for each sample. That is the function of the *mag_phase_accu* block. Each time a new sample is sent, the signal *new_sample* goes high briefly. That tells this block that it must increment the value of *phase* by a certain amount. That amount can be set using the dip switches. The larger the amount, the higher the frequency at the output of the codec. The block *cordic* then performs the calculation and sends the result to the codec.

(060025-4)

CORDIC

CORDIC (**Co**ordinate **R**otation **Di**gital **C**omputer) is a method that can be used to implement goniometric functions efficiently in digital systems. It describes how to calculate goniometric functions using only add and shift operations.

CORDIC uses vectors. These vectors can be described as a combination of real and imaginary numbers (corresponding to their X and Y coordinates), or as a combination of a length and an angle.

If two vectors (A and B) are multiplied together, the length of the resulting vector (C) is equal to the length of vector A times the length of vector B. The angle of the resulting vector is the sum of the angles of vector A times and vector B. This multiplication takes the following form in X,Y notation:

$X_c = (X_a \times X_b) - (Y_a \times Y_b)$

$Y_c = (Y_a \times X_b) + (X_a \times Y_b)$

As you can clearly see, this requires using multiplications. If we ensure that these multiplications are all powers of 2, everything becomes very simple. Multiplication by a power of 2 (such as 2^{-2}) is equivalent to shifting bits. That is very easy to do using digital logic.

The CORDIC method describes how to calculate goniometric functions by multiplying an initial vector by vectors with X coordinates equal to 1 and Y coordinates that are always powers of 2. As a result, this method can be implement very efficiently in digital circuitry.

Example

Consider the following expression as an example:
100 × cos(30°). As our starting point, we take the vector (100, 0), which has length of 100 and an angle of 0°. The desired angle is greater than the current angle, so the first operation is to multiply the vector by the vector (0, 1). Our vector now has an angle of 90°. That is greater than the desired angle.

The next step is thus to reduce the angle. For that purpose, we multiply by the vector (1, –1). Note that the Y value of this vector is negative, so angle of this vector is also negative.

After this multiplication, the angle of our vector is (90° – 45°) = 45°. This is still greater than 30°, so the next step is to multiply by the vector (1, –0.5). This causes to angle of our vector to become (45° – 26.57°) = 18.43°.

The angle is now smaller than what we want. We thus use the vector (1, 0.25) in the next step. That increases the angle by 14.04°. After this step, the angle of our vector is 32.47°. We get closer to 30° with each step, so the result is more accurate with each step.

Each multiplication changes the length of our vector as well as its angle. In the present case, we now have a vector with a length of 100 × (1 × 1.41 × 1.12 × 1.03) = 162.66. That means we have to multiply this value by a correction factor. Another option would be to make the length of the initial vector 61.5 in order to finally obtain a vector with a length of 100. This multiplication factor is always the same, regardless of which angle we want.

No matter which option we choose, the desired cosine value is given by the X coordinate of our vector, while the Y coordinate represents the sine value. We get those values for free!

X	Y	Angle	Length
0	1	90°	1
1	1	45°	1.41
1	1/2	26.57°	1.12
1	_	14.04°	1.03
....	....	....	....