FPGA Course (6)

Part 6: connecting a keyboard

Paul Goossens and Andreas Voggeneder (FH Hagenberg)

You no doubt have an old PS/2 keyboard gathering dust somewhere. Now you can put it to good use again as an input device for the FPGA prototyping board. This instalment of our FPGA course tells you how.

Most prototyping systems use an RS232 link to send and receive data. This link normally goes to a PC running a terminal emulator program. The user thus uses the monitor of the PC as the actual output device and the keyboard as the input device.

You can dispense with the PC if you connect a monitor and keyboard directly to the prototyping board.

PS/2

Until recently, PC keyboards were always connected to the PC via the PS/2 bus. The FPGA prototyping board also has two PS/2 ports. Besides the power supply lines, a PS/2 port has a data line and a clock line. Both of the these lines are bi-directional, and here they are connected directly to several pins of the FPGA on the circuit board.
Data is transmitted from a device (such as a keyboard) to the host (in this case the FPGA) as follows. First, the device transmits a start bit. The start bit is always a 0. It then sends the eight data bits starting with the least significant bit. Next comes a parity bit with odd parity, which means the parity bit is a 1 if an even number of 1 bits are present in the transmitted byte. Finally there is a stop bit, which is always a 1.
The clock signal for the transmission is generated by the device. The signal level on the data line changes only when the clock signal is high (see **Figure 1**).

From host to device

Communication from the host to the device is somewhat more complicated. First, the host must indicate that it wants to transmit data. It does this by setting the clock line low, which terminates any communication that may already be in progress. After a brief delay, it sets the data line low. Finally, it sets the clock line high again.
As a result, the device knows that the host will be transmitting data. In response to this, the device generates a clock signal. The host then sends one data bit for each clock pulse. This starts with a start bit (0) followed by eight data bits. These data bits are also sent starting with the least significant bit. The next bit is a parity bit (odd). Finally, the host sends a stop bit (1), but the communication session is not yet complete. The final action is that the device sends an 0 as an acknowledgement (ACK) if the data was received correctly (see **Figure 2**).

Software versus hardware

Of course, it is possible to implement the PS/2 protocol in software. This approach was taken with the I²C bus in a previous instalment. A disadvantage of this method is that the processor must spend some of its time processing the signals. It also doesn't make the software any simpler.
An alternative approach is to design a hardware interface that looks after generating and processing the signals. Here we describe how the PS/2 interface can be implemented in hardware. With a hardware interface, the microcontroller (8051) does not have to be concerned with the details of how the PS/2 interface works.
Before examining the hardware the implementation of the PS/2 interface, let's have a look at the T8052 microcontroller.

T8052

We've already used the T51 softcore processor several times in this course. The microcontroller consists of a processor portion called T51 and several peripheral devices, such as a UART, timers, and so on. Like every MCS51 processor, the T51 uses a special system bus to drive the peripheral devices. The registers on this bus are called Special Function Registers (SFRs). The original microcontroller (8051) did not use all available addresses. This was done intentionally to allow room for other peripheral devices. This possibility can be used to add a PS/2 interface to the microcontroller as shown in our *ex17* example.

The code in *T51_Glue.VHDL* looks after decoding the addresses of the SFRs. We add the following lines here:

```
ps2_data_sel      <= '1' when IO_Addr = "1011001" else '0'; -- 0xD9
ps2_data_wr       <= '1' when IO_Addr_r = "1011001" and IO_Wr = '1' else '0'; -- 0xD9
ps2_ctrl_stat_sel <= '1' when IO_Addr = "1011000" else '0'; -- 0xD8
ps2_ctrl_stat_wr  <= '1' when IO_Addr_r = "1011000" and IO_Wr = '1' else '0'; -- 0xD8
```

This bit of code makes register *PS2_DATA* available at SFR address 0xD9. Register *PS2_CTRL_STAT* is available at address 0xD8.

PS/2 interface

The actual PS/2 interface is described in *PS2Keyboard-a.VHDL*. The interface with the SFR bus is defined starting with line 218. Line 218 is part of a process that is evaluated on a rising clock edge. If a rising clock edge occurs when *data_wr_i* is high, the content provided by the processor is loaded into register *CmdReg*. In addition, a 1 is loaded into bit 8 (which is the ninth bit!).

If *ctrl_stat_wr_i* is high, several registers are loaded with the corresponding bits present on the data bus. As you can clearly see, the interface is returned to the quiescent stat if bit *7* is a 1. It waits for data from the device when it is in this state.

The content of *SFR_Data_o* is read by the processor in case of a read operation. Line 236 causes the content of the internal *DataReg* register to be passed if it is selected by the *data_sel_i* signal. Otherwise the content of the status register is passed.
The actual data transfer to the processor is handled by the *T8051.VHDL* file.

The remainder of the *PS2Keyboard-a.VHD* file describes the communication between the FPGA and the PS/2 bus. The comments in the source code should make it reasonably easy to understand how this works. If you can't figure it out, you can always use the simulator to examine the interactions between the internal signals.

Example

The first example of how to use the PS/2 interface is *ex17*. Start by connecting a keyboard to the connector labelled 'KEYBOARD', and then use the accompanying configuration file to configure the FPGA.
If everything goes as it should, a message will appear on the LCD. From now on, all data sent by the keyboard will

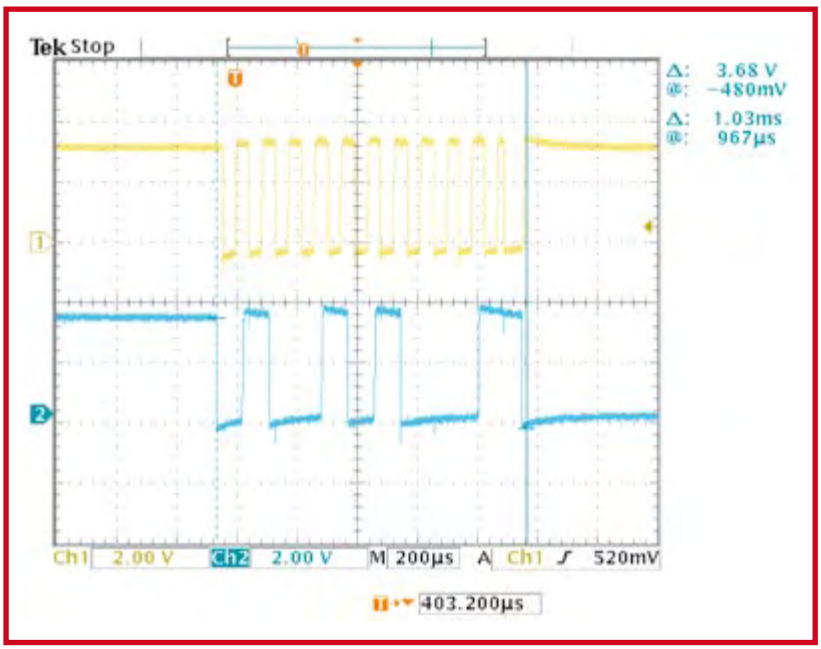

Figure 1. Data transmission from the device to the host. The upper trace shows the clock signa

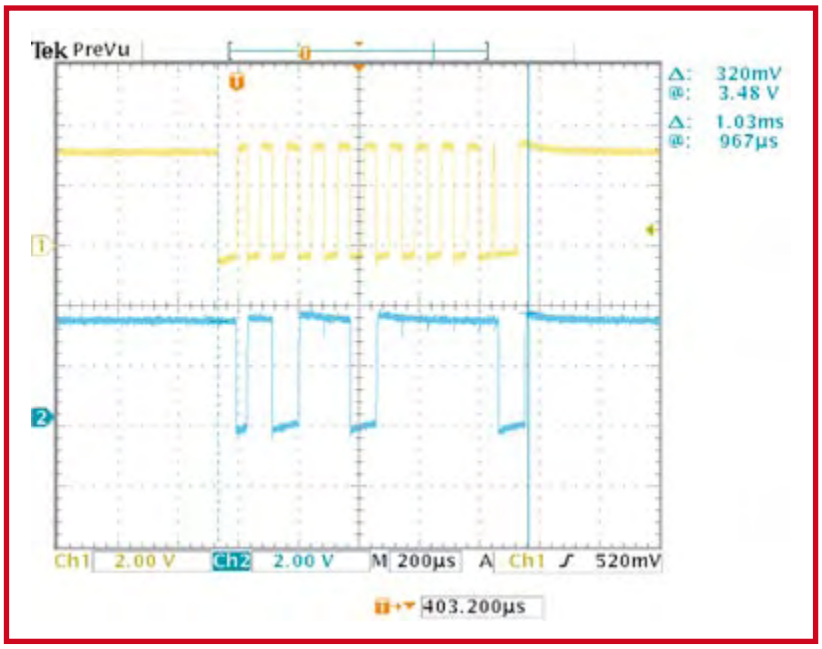

Figure 2. A data block transmitted from the host to the device.

be shown on the LCD in hexadecimal form. The keyboard will not send any data when it is not being used.
Start by pressing any desired key of the keyboard. The corresponding data will be sent immediately to the FPGA to report this action. If you hold a key pressed for longer than a certain interval, the keyboard will transmit the corresponding code repeatedly.
Let's suppose you pressed the 'w' key. According to the LCD, the keyboard sent the code 0x1D. The keyboard actually sends two codes when a key is released. The first code it sends is '0F', which indicates that the user has released a key. After this, it sends the code of the key concerned.

Firmware

How is all of this handled in the software? First, the LCD is initialised as usual. After this, the *init_kb()* routine is called. This is located in *kb.c*. Initial values are assigned to the variables here. After this, the final action is to call *InitKbd()*, which is located in *fpga_lib.c*. It shows how data can be sent to the keyboard and read from the keyboard. The first action is to send the code 0xFF to the keyboard to perform a reset. The keyboard returns an ACK (0xFA) if this code is received correctly. If the keyboard returns 0xFE, an error has occurred during the communication process.
If everything has gone as it should up to now, the keyboard will execute a self-test. This is indicated by briefly lighting the LEDs on the keyboard. If this test is completed successfully, the keyboard sends 0xAA as a sign that it is OK. From this point on, the keyboard operates the way it is supposed to.
The rest of the code has been kept very simple. Whenever data is received, the interrupt routine *ext_int2_isr()* causes the data to be stored in a buffer.

Scan codes

OK, now you can communicate with the keyboard. Keypresses are also being sent to the processor, but what you actually want to receive is normal ASCII characters. For this reason, we extended the interrupt routine of *kb.c* in a new example (*ex18*). The new version of the interrupt routine converts the scan codes into ASCII characters.
Our more inquisitive readers will have certainly also tried the CapsLock key in the previous example. If you did so, you would have seen that the associate LED did not respond at all. That's also the way it should be, since the LEDs are driven by the host. The routine *Set_LED()* in example 8 shows how you can drive the keyboard LEDs. In this example, the number of typed characters is counted and the result is indicated by the LEDs on the keyboard. As only three LEDs available, only numbers in the range of 0 to 7 can be indicated.

This instalment clearly shows that the full power of the FPGA can only be utilised if you make use of the advantages of configurable logic relative to conventional microcontroller systems. Of course, you could also implement the entire PS/2 protocol in software, but that would cost extra processing time. With the hardware implementation described here, the processor does not have to look after handling the electrical signals on the PS/2 bus, so it has more time for other tasks.
The I²C protocol, which was implemented in software in a previous instalment of this course, could also be implemented in hardware in a similar manner.

In the next instalment, we will add a fully functional VGA output to the microcontroller system of this month's instalment. Naturally, it will also be implemented in hardware.

(060025-VI)