FPGA Course (8)

Part 8: Playing with the USB port

Paul Goossens and Andreas Voggeneder

In this instalment of the course We're going to pu together our own USB microcontroller. In terms of technical features, it can certainly hold its own against commercial USB controllers such as the Cypress IC.

The specifications of our home-grown USB controller embedded into an FPGA are certainly nothing to be ashamed of:

- 8052 processor running at 48 MHz with single clock cycle per byte fetch
- full-speed USB controller with five endpoints
- 8 KB of ROM
- 4 KB of RAM

We built the USB controller by extending the familiar 8052 core with a USB controller core, which is also freely available from OpenCores (www.opencores.com). This is not the place to delve into the details of how the USB core works or its internal structure. What matters here is how it is linked to the 8052 core and how to use it in practice. The USB core is the USBHostSlave core from OpenCores (www.opencores.org/projects.cgi/web/usbhostslave/overview). It was designed by Steve Fielding, and it has the following properties:

- register-configurable low-speed / high-speed mode (1.5 Mbps or 12 Mbps)
- 4 freely available endpoints, each with a 64-byte FIFO
- support for control, bulk, interrupt, and isochronous transfers
- host mode
- 8-bit Wishbone bus interface

The USBHostSlave core is coupled to the 8052 microcontroller via the Wishbone bus. As a result, the registers of the USB core appear in the XRAM memory area (in other words, they are addressed as external SRAM), where the USB core occupies 256 addresses. **Figure 1** shows the block diagram of the overall system, while **Figure 2** depicts the memory organisation.

Registers

The USB core has a set of four registers for each endpoint:

- EP[0-4]_Control
- EP[0-4]_Status
- EP[0-4]_Transtype_Status
- EP[0-4]_NAK_Transtype_Status

The Endpoint_Control registers can be used to control or drive endpoints for purposes such as sending data, setting or clearing stall states, or setting the operating mode of an endpoint (isochronous or bulk). The Endpoint_Status registers provide information about the current status of the endpoints – for instance, whether data has been received or an error has occurred.
The Endpoint_Transtype_Status register contains information about the last successful transfer between the PC and the USB core, such as whether a setup package was received. The Endpoint_NAK_Transtype_Status register displays any errors that may have occurred during the data transfer.

Besides these four primary control registers, each endpoint has an Rx FIFO buffer and a Tx FIFO buffer, each with a depth of 64 bytes. These two FIFOs can be controlled using an additional set of six registers for each endpoint:

- EP[0–4]_Rx_Fifo_Data
- EP[0–4]_Rx_Fifo_Data_Count_MSB
- EP[0–4]_Rx_Fifo_Data_Count_LSB
- EP[0–4]_Rx_Fifo_Control
- EP[0–4]_Tx_Fifo_Data
- EP[0–4]_Tx_Fifo_Control

Among other things, these registers are used to read data received via the endpoint from the associated Rx FIFO and check the fill level of the FIFO, to write data to a Tx FIFO, and to empty (flush) the FIFOs.
There are also several general Slave Control (SC) registers, including:

- SC_Control
- SC_Line_Status
- SC_Address
- SC_Interrupt_Status / Mask

The SC_Control register can be used to enable or disable the USB slave portion and configure it in either full-speed mode (12 Mbps) or low-speed mode (1.5 Mbps). The SC_Line_Status register can be used to check the line status (link to the PC) to determine whether a link actually exists (plug connected) or the link is interrupted somewhere. The SC_Address register is used to set the slave address assigned by the PC during the enumeration process. The SC_Interrupt_Mask register specifies which interrupts are allowed (which means they can generate an 8051 interrupt) and which ones are not allowed. The microcontroller can use the SC_Interrupt_Status register to determine the source of an interrupt.

Besides these slave-specific registers, there is also a large set of registers for the USB host portion, but they are not described here. As described later on in this article, the FPGA board cannot act as a host.

Hardware

The pull-up resistance that defines the connect condition can be controlled by the microcontroller via port line P3.7. P3.7 is high after a reset. MOSFET T6 (which connects the pull-up resistor to D+) is cut off under this condition, so the PC does not see any connected device. When the software has initialised the USB core and is ready to communicate with the PC, it pulls P3.7 low. This causes the D+ to be connected to +3.3 V via the 1.5-kΩ resistor. As a result, the PC sees that a full-speed device (12 Mbps) is connected and initiates communication with the FPGA board.

The USBHostSlave core needs a 48-MHz clock. This is stipulated by the USB specification. A clock signal at this frequency is derived from the 50-MHz clock signal by a PLL in the FPGA. The USB core and the T8052 core are both driven by the 48-MHz clock signal. As the T8052 core can execute all instructions in 1 to 4 clock cycles (except the divide instruction, which takes 12 clock cycles), this provides a considerable amount of computing power (at least for an 8051).

The USB core in the FPGA handles all the tasks necessary for USB communication with the PC. All that remains to be provided is the USB physical interface (USB Phy). Its job is to provide the electrical interface (level adjustment). This task is also handled by the FPGA.

Mixed Language

Besides VHDL, there are several other ways to design a digital circuit. One of them is to use a graphic drawing package. We described this option in earlier instalments of this course.

Other entry methods include Verilog, AHDL, SystemC, and so on. Fortunately, the Quartus compiler can also handle some of these languages. Using two or more different languages is called 'mixed-language design'.

The USB core used in this instalment was designed in Verilog. Some of the other components were designed in VHDL. Our design can thus be regarded as a mixed-language design (part Verilog and part VHDL).

The main advantage of this is that you are not limited to using only VHDL cores or only Verilog cores. You can use them in combination in your designs without any problems. However, you do need to know the Verilog equivalents of some VHDL terms, such as STD_LOGIC signals.

The link between the VHDL part and the Verilog part is located in the USB.vhd file. This file is what is called a 'wrapper', and its purpose is to export Verilog signals as VHDL signals.

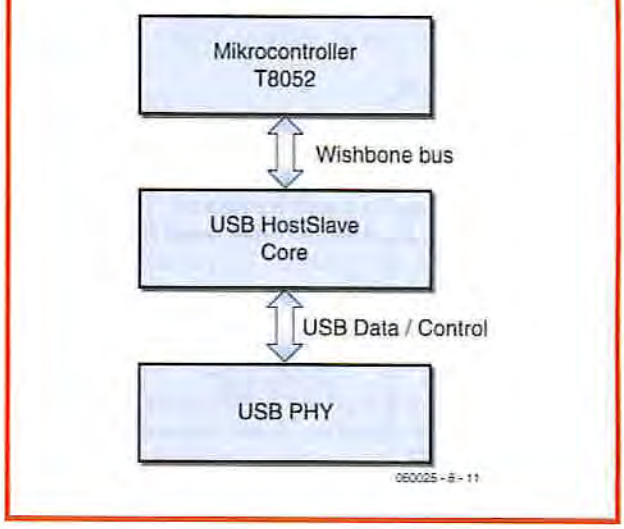

Figure 1. The USBHostSlave core in the T8052 system.

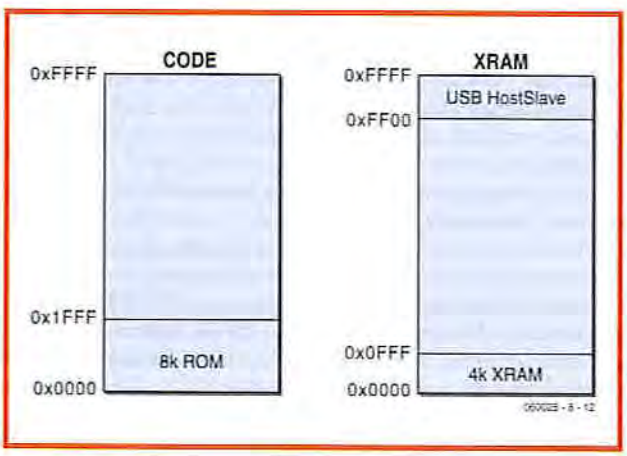

Figure 2. T8052 memory organisation.

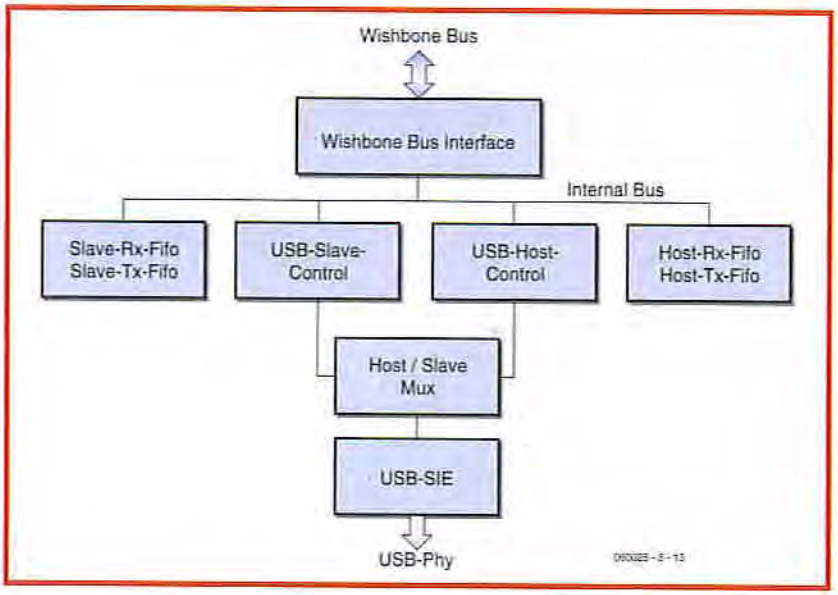

Figure 3. Block diagram of the USB core

The USB core also supports host mode, which means it can act as a PC for functions such as enumerating and driving a USB mouse or other type of USB device. However, host mode cannot be used with the FPGA prototyping board because it lacks the necessary hardware (external to the FPGA). Here the USB core can only be used in slave mode with low-speed (1.5 Mbps) or full-speed (12 Mbps) data transfer.

Sample application

Our demo application here (*ex22*) involves connecting an HID keyboard. It identifies itself to the PC as a keyboard, which causes the PC to use the Human Interface Device (HID) protocol defined in the USB standard. The main purpose of the HID protocol is controlling input devices such as keyboards, mice and joysticks. The principal advantage of this protocol is that all major operating systems (Window, Linux and MacOS) incorporate drivers for it, so they do not have to be written for individual devices.

Endpoints 0 and 1 are used for communication with the PC. Endpoint 0 is a control endpoint, which must be present in every USB device. It is used for basic communication with the PC (enumeration). Enumeration is a process that takes place immediately after a USB device is connected to a host (PC). Each new device is recognised by the 1.5-kΩ resistor connected to the D+ or D– line in every USB device. If the resistor is connected to the D+ line, the device is a full-speed device (12 Mbps); otherwise it is a low-speed device (1.5 Mbps). The FPGA prototyping board has a switchable resistor for each of these lines, so you can configure it as desired as a full-speed or low-speed device. The PC detects the type of device during enumeration and assigns a slave address to the device. The appropriate driver is then loaded based on the Device_Descriptor data sent by the device. After this has been completed, an application can communicate with the device. In this case, the driver is the HID driver provided by the operating system. Other endpoints are not enabled until after the driver has been loaded. In the case of the HID keyboard, this is endpoint 1, which acts as an interrupt endpoint (IN = device to PC) and is used for transmitting keystrokes.

On the *Elektor Electronics* FPGA prototyping board, push-buttons S2–S4 and the eight DIP switches in S5 are used as 'keys', with S4 serving as a Shift key. S1 acts as the reset button for resetting the 8052 and the USB core. LED 7 acts as the indicator for the Number Lock function, while LED 6 acts as the Caps Lock indicator and LED 5 as the Scroll Lock indicator.

Bonus application

Our second sample application (*ex23*) is a PS/2 to USB adapter. You can use this adapter to connect a keyboard with a PS/2 connector to a USB port of a PC. The PC will see the keyboard as a USB keyboard. The FPGA causes the keystrokes of the PS/2 keyboard to be transferred via the USB interface just as though they originated from a real USB keyboard.

Naturally, this example uses the PS/2 interface as well as the USB interface. This means the associated firmware is more extensive, but if you read the PS/2 instalment you should be able to understand how this application works As usual, the software for the examples described in this instalment is available on the *Elektor Electronics* website under item number 060025-8-11 (go to Magazine → January 2007 → FPGA Course Part 8).

(060025-8)

Andreas Voggeneder

Andreas Voggeneder studied Hardware/Software Systems Engineering (HSSE) at the University of Applied Sciences in Hagenberg, Austria (**http://hsse.fh-hagenberg.at/**), where he presently still gives occasional lectures.

As an enthusiastic electronics specialist, he devotes part of his time to designing digital circuits with the help of VHDL and Verilog. He is also a moderator at OpenCores (**http://www.opencores.com/**), a forum dedicated to the freely available T51 8051 processor core used in several instalments of this course.