FPGA IMPLEMENTATION OF USB 1.1 DEVICE CORE

Pavel Kubalík
xkubalik@fel.cvut.cz

Jiří Buček
bucekj@fel.cvut.cz

CTU Prague, Faculty of Electrical Engineering
Department of Computer Science and Engineering
Karlovo náměstí 13, Praha 2
http://service.felk.cvut.cz

UTIA Prague, Pod vodárenskou věží 4, Praha 8
http://www.utia.cas.cz/zs

Abstract. *The aim of this work is to develop a USB device core that can be used to transfer large amount of data. Therefore we focused on the BULK transfer mode. The core enables to create a simple connection between a PC and a developed device. The core is described in VHDL language. The USB core implementation and testing were performed on the XSV800 development board.*

1 Introduction

Nowadays extensive usage of FPGAs implies the need for communication that is faster than a simple serial line. The designer has to transmit large volumes of data between his FPGA application and the PC. The USB interface is one of the solutions to this problem. The USB interface conforms to the requirements of simplicity and universal usage, which comes from its widespread presence in current PCs. In order to use the USB, one has to add a USB core and a USB transceiver to the application board.

The designer should invite software tools that simplify the implementation of the USB communication. These tools include a USB driver implementing the basic transfer methods, and a simple application demonstrating the way of communication with the driver on the PC side. The FPGA side consists of the USB transceiver and the USB core, which is designed at the register transfer level (RTL).

There are several designs dealing with various parts of the given problem. The majority of them are commercial solutions, which are not suitable for research. The reason is the absence of source code and thus the inability of modification.

The ultimate goal of our work is to gradually create universal software tools for simple usage of the USB communication. During the design it is not possible to concentrate solely on one part, it is necessary to create them at the same time. The reason is the ability to test and gather specific communication parameters, which determine the fitness for a particular application.

2 Solution

This chapter is divided to five basic parts. The first part describes the USB core itself implemented in FPGA. The second part deals with using the USB transceiver. The third part describes the external microcontroller used for making the initial communication and setting transfer parameters. The fourth part deals with using the USB driver and the final part describes the testing application.

2.1 USB core for FPGA

The USB core is written with respect to simple portability to other developed devices. Special attention was devoted to the minimum resources needed to transfer large amount of data from and to the PC. The second important point is the maximum transfer rate achieved by the USB core.

This section describes four parts (see figure 1). The first two parts are the receiver and the transmitter implementing the low level of the USB communication protocol. Next part deals with the control of the middle level of communication protocol such as handshaking. The final part is the microcontroller interface used for the communication with the external microcontroller.

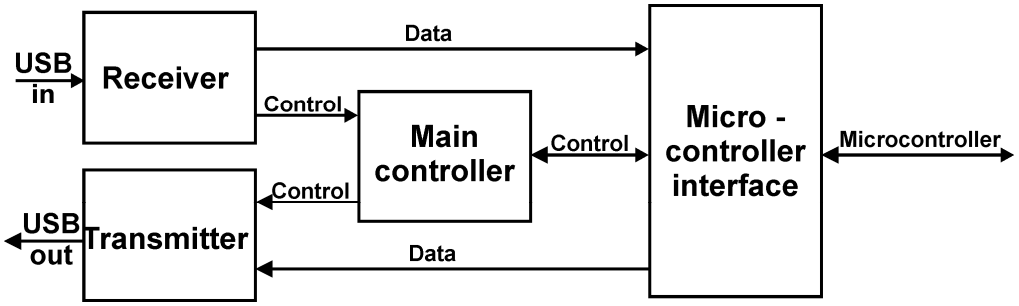

Figure 1: USB core block diagram

Receiver

The receiver processes the data coming from the external USB physical layer transceiver and stores it at various places depending on the type of the packet received. The receiver block is hierarchically divided into several parts.

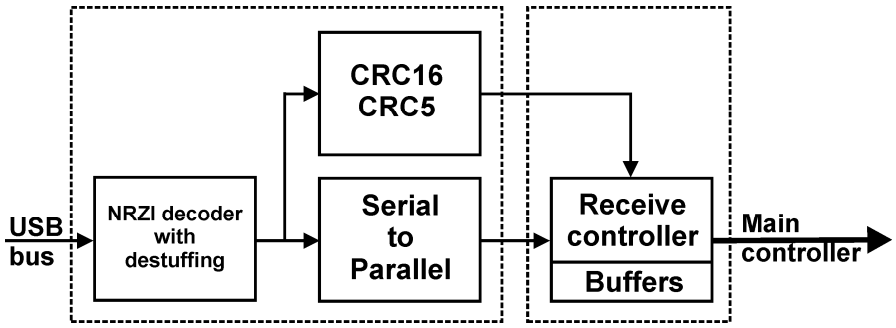

Figure 2: Transceiver block diagram

The top level of the receiver consists of two parts. The first part processes the input data stream and analyzes its structure. The second part is the receiver controller, which uses the information about the structure to deliver the data to the corresponding storage area.

The data received at the primary inputs has the form of a serial signal that is asynchronous to the system clock. Therefore the first block to which the data is routed performs synchronization to the system clock. The synchronization is done using a digital PLL state machine. The data is decoded from the NRZI form in the process of synchronization.

In the next step the stuffed bits in the data stream are detected and marked for removal. The start and the end of the packet are detected and signaled for further processing. The serial data is then converted to parallel.

The parallel data is examined in order to determine the packet type. The packet types recognized at this level are: the token, the data and the handshake. The type of the packet is determined using the PID field, which is the first byte after the synchronization sequence. The data and token packets are secured against transmission errors by a 16-bit and 5-bit CRC, respectively.

If the packet is a token, the received bytes are stored in a temporary register. After the CRC is confirmed to be valid, the packet is transferred to the token register and its information can be used during the current transaction.

If the packet type is data, we check whether there is free space in the buffer of the destination endpoint and whether this data packet is a valid part of a running transaction. If these conditions are fulfilled, the data is written into the corresponding endpoint buffer. Otherwise the data is received but discarded. The data is checked for possible errors using the 16-bit CRC. The receiver status is signaled to the main controller.

Transmitter

The transmitter is used for packet sending. The transmitter is divided into four blocks. The first block, the nearest to the main control (see figure 3), implements the control endpoint buffers and sends its contents to the parallel/serial block. The data stored at each buffer is sent after the send command is received. The commands are received from the main control. The last received token determines the buffer, which will be used for sending data.

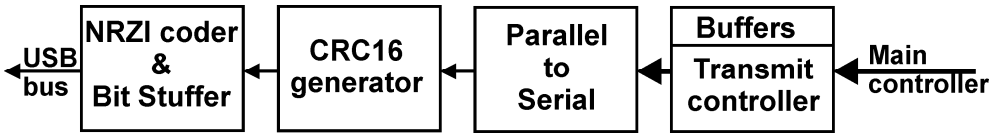

Figure 3: Transmitter block diagram

The transmit control block can only read data from these buffers. The write operations to the buffers are performed by the microcontroller interface. Data is sent to the parallel/serial block by bytes. The first and the last byte marked in order to detect start and end of packet. The parallel/serial block is used for making serial data from received bytes. The generated serial data is sent to the CRC16 block. The CRC16 block generates a 16-bit control checksum. This block is unused when a handshake packet is sent. The last block included in the transmitter is the NRZI coder, which performs the bit stuffing and NRZI coding. This block controls the external USB transceiver. Coded and stuffed data is sent outside of the FPGA.

The transmitter is designed with respect to the maximum time allowed for generating the first bit of the response to the received command.

Transmitter buffers

The first approach was using 1024-byte buffers implemented with the blockram components included in the FPGA and the length of the transferred packet was 1023-bytes. After implementation and testing the USB core buffers were shortened to 64-byte length. We expected that the maximum transfer rate depend on the length of the packet. Of course it did but we can reach the same speed with 64-byte packets. In other words, the minimum length of the packet to reach the maximum transfer rate is 64-bytes. If the USB driver is not written effectively then the max packet length is very important. It is due to the number of packets transmitted in 1 ms. For example, if the length of packet is 1023 we transferred 9 packets. That means 9288 bytes per 1 ms. By using 64-byte packets we transferred 150 packets, i.e. 9600 bytes. In other words using 64-byte packets is faster than 1023-byte packets.

Because of the short time interval between two consecutively transferred packets the double-buffering technique must be used. It means while the first buffer is being filled the second buffer is waiting to be sent. The 64-byte buffers are implemented as a distributed RAM.

Main controller

The main controller manages the middle level of the USB protocol, such as handshaking. The communication on this level is based on transactions consisting of token, data and acknowledge parts. Some of the transactions do not include the data part. Each small part of the transaction is received as a separate packet. The error checking of the packet is performed in the receiver and the information signal is sent to the main controller. The receiver block takes care of the CRC checking and the corresponding response is prepared in the main controller block. The response must be generated immediately after the packet is received to achieve the maximum allowed time given by the USB specification.

The main controller consists of a state machine. The actual state depends on the type of the token and on the position in each transaction. Each endpoint has a few parameters such as the transfer type or the status register that defines its current state. These parameters are stored in structures. When data has been sent or received the interrupt signal is generated and the type of this interrupt is placed into the status register. The status register can be read by the external microcontroller.

Microcontroller interface

The microcontroller interface block takes care of the communication between the USB core and the microcontroller. The register field used for filling or reading buffers and setting basic parameters of transfer is implemented in this block. By using the select endpoint command, the actual endpoint for read or write operations can be selected. Because of the external microcontroller, input registers must be used to prevent metastable states. Input buffers add other delays.

2.2 USB physical layer

The USB core needs only one external circuit implementing the physical layer. The external circuit used is the PDIUSBP11A. This circuit is included on the XSV800 board but the USB core is independent on type of used the USB transceiver and can be replaced by another one.

The USB core was tested only for full speed baud rate because of simplicity the USB core. The USB transceiver is permanently receiving serial data and sending them to the USB receiver block. It means that every packet is received, even the packet sent by the USB transmitter. Unexpected packets in the USB receiver are ignored and information about received packet is not generated. In other words idle packets are dropped.

2.3 External microcontroller

In order to make testing of the USB core simpler, an external microcontroller was used. It enables an easy way of making changes at the higher level of the USB protocol like setting the length of the sent packet. The ATMEL AT90S8515 is used as the microcontroller. The program is written in the C language and compiled by the GCC to the microcontroller code. Before data can be transmitted, initial communication (i.e. enumeration) has to be performed and transmission parameters must be set.

For keeping information about transfer parameters it is necessary to use a table for each endpoint and pipe. Implementing tables in the microcontroller is simple. The solution with microcontroller is better than a hardware state machine because of the complexity of the protocol layer. It means the microcontroller is a suitable solution because of its simple enhancement and adequate parameters. A simple microcontroller core could be used for initial communication and a simple state machine for the data transmission.

Due to the fact that the microcontroller is external to the FPGA, the read and write operations were slowed down. In order to test the maximum throughput of the core, we had to skip filling the buffer. Instead we sent a large amount of identical packets only by repeatedly validating the transmit buffer. This was enabled by the fact that the transmit buffer is not cleared after sending.

2.4 USB driver for PC

The USB core was tested under MS Windows. It is necessary to use a driver to transfer data. The driver must be written as a WDM driver. It ensures the usability under Windows 98/2000/XP. The driver was adopted from the Bulk USB driver in the Microsoft Driver Development Kit. The driver must be modified to improve transfer rate. Namely the length of buffers must be increased and round on multiple length of the packet. After the test was passed the best length of driver buffer was determined.

2.5 Application for PC

A simple application was written for testing purpose only. It contains functions that take care of the initialization of the interface to the USB device driver. After that the transaction are performed using the standard API function such as ReadFile or WriteFile. A lot of the code is adopted from the Bulk USB example in the Microsoft Driver Development Kit.

We implemented a simple 1-megabyte buffer for storing received data during analysis of the maximum transfer rate. Data was read from the USB driver by an integer multiple of the length of the driver buffer.

2.6 Analysis of the USB bus

During the development of a USB device it was necessary to have a closer look at the USB traffic. We designed a USB analyzer for this purpose. The analyzer is a cut down version of the USB core containing merely the receiver part and a FIFO. The analyzer is connected to the PC via serial or parallel interface. The analysis is then performed on the PC.

The first version utilized a RS-232 serial line, which showed to be insufficient. When using the serial line, we could not monitor the data portion of the packets due to the lack of speed (maximum 115200 bits per second on our PCs). In order to see more information e.g. during the enumeration phase, we needed to implement a faster interface. We decided to use the EPP parallel interface. Maximum speed we achieved was ca. 700 kilobytes per second. We used the largest available FIFO implemented using all blockrams available in the FPGA. However, FIFO overflows still occurred and in the time of traffic peaks we lost some information. It was necessary to create a simple line protocol in order to discriminate individual packets.

By using this tool, we can observe the USB traffic and analyze it on the PC. We analyzed the packets that were transmitted between various devices and we used that information during debugging of our USB core.

3 Results

The main result of this work is the USB 1.1 VHDL core. The core implements a BULK data transfer between the XSV800 prototyping board and the PC. The USB core occupies 1165 slices, 520 of these are used for buffers.

This design contains only the USB core without the microcontroller. As an external microcontroller we used the AT90S8515 made by ATMEL. The PC side consists of a USB driver and an application. The driver source code is a part of the driver development kit software by Microsoft. A simple application is written only for testing purpose and gathering transfer parameters.

To achieve the maximum transfer speed it is necessary to adjust the size of the data transfer buffers in the driver. In addition the application must use an integer factor of the data transfer buffers. The maximum transfer rate achieved by the correct settings was 8Mb/s that is 1 megabyte/sec. The ineffective driver source code is the bottleneck of the communication, which prevents achieving the maximum transfer rate.

4 Future Work

In the future we want to embed the microprocessor into the FPGA. We expect to use the microprocessor developed at CTU. We plan to finish the core in such a way that it can be easily integrated into other projects. We need to further optimize the usage of the FPGA internal memory.

5 Summary

The result of our work is USB 1.1 VHDL core written in the VHDL language. We have implemented only the BULK mode for data transfer and the CONTROL mode for enumeration. We achieved 1 megabyte/sec transfer rate for BULK mode. The USB core

occupies 1165 slices, 520 of these are used for buffers. We implemented the USB bus analyzer working concurrently with the USB core.

References

[1] Compaq, Intel, Microsoft, Nec: *Universal Serial Bus Specification, Revision 1.1, 1998*

[2] Kubalík, P.: *Fast AD converter with the USB interface. Diploma thesis, CTU Prague 2002*