CUSTOM USB INTERFACE FOR NEUROPROSTHESIS

BY

RAHUL KUMAR

ELECTRICAL AND COMPUTER ENGINEERING

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Engineering
in Electrical and Computer Engineering
in the Graduate College of the
Illinois Institute of Technology

Approved _____
Adviser

Chicago, Illinois
December 2004

ACKNOWLEDGEMENT

I would like to take this opportunity to thank all those without whom this thesis would not have been possible. I would like to thank Dr. Stine for his excellent teaching that gave me a good grounding in Verilog RTL and digital systems design. I would like to thank Dr. Troyk for encouraging and instilling scientific thought, giving me his suggestions on how to implement a reliable and robust design, and performing numerous thesis reviews.

I would also like to thank Cathie for helping me with all the purchase orders, David and Nishant, who shared numerous design and implementation ideas with me. I would like to thank Keyur for suggestions regarding device driver and application development and Swapna for fixing my computer and printing problems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Neuroprosthetic research is nearing a point where it requires high-speed link to the PC, owing to the higher number of electrodes being used. This research aims to verify and validate the use of the USB, which is a widely available, high bandwidth PC communication link. The USB provides flexibility in the different kinds of devices and transfer types it supports. The USB protocol, along with the potential application types are also described and discussed. An open source implementation of the USB IP core is tested and customized for interface with two neuroprosthetic stimulator chips developed at the Pritzker Institute of Bio-Medical Science and Engineering. Bandwidth analysis for the application interfaces was done considering the bus timing limitations of the USB. The difficulties faced while verifying the USB core, while implementing the hardware development board, and while designing the interfaces are also discussed. The development board was developed by hand and not as a PCB, as it gave complete flexibility in implementing modifications. The problems encountered were resolved and the USB core and two chip interfaces were successfully tested and validated using FPGAs. The two different cores have been fabricated in ASIC.

CHAPTER 1

INTRODUCTION

The Bio-Medical engineering field in recent years has enjoyed considerable success with the use of the pacemaker for patients who have irregular heart rhythms and cochlear implants that assist patients with hearing loss. Extensive research is currently being conducted in the field of neuroprostheses, where the goals of the research vary from brain-machine interface to stimulation of paralyzed muscle. Regardless of the application, often the success of any implantable device depends on its functionality, the amount of power it requires to function and the physical size of the device.

Power for an implanted neuroprosthesis device is typically provided over an inductive transcutaneous link using a radio-frequency (RF) carrier, and that same link is often used for data transmission to and from the implanted device. Using a separate communication link for the implant is less common since the RF power link can be easily used as a communication link, and setting up a separate link would require another inductive coil pair operating at another frequency, and may consume valuable chip area and additional power. To justify implantation, a device should also have sufficiently high enough functionality. Most systems that are in the research stage are not fully implanted since their functionality has been insufficiently researched or validated. Portions of the system are left external so that their requirements can be refined in preparation for the design of a fully implantable device. For modern neuroprotheses designs, as the functional complexity, and number of communication channels are increased, so must be increased the speed of communication to and from the implantable device. To select a reliable, robust communication link that allows for high-speed computational control of the implant, the use of personal computers must be considered. Only proven communication links are widely available in the form of computer peripheral buses. Other commercially available communication systems are more specialized and are not as common as the

peripheral buses. Adapting custom bio-medical devices to these buses is advantageous, in terms of available support and availability on PC's. Earlier research projects on such devices had extensively used advances in computer technology. For instance, many research projects used the serial port for data transfer, while some others had a specialized hardware interface to the internal PC bus. Understandably system development for devices using the serial port is relatively simple then for as compared to specialized internal buss hardware devices. The specialized hardware devices, on the other hand have better performance than the serial port, due to their direct access to the high-speed PC data buss. For emerging neuroprosthesis research, there is a need for a communication link, that is easy to implement and has higher performance than the serial and parallel ports. This research establishes the feasibility of using the USB as a communication link for a visual cortical stimulator that has been developed at the Pritzker Institute of Bio-Medical Science and Engineering. Two prototype devices are designed, implemented and tested. Each device is tested in FPGA and ASIC form.

1.1 NEUROPROSTHETIC SYSTEM FUNCTIONALITY

A neuroprosthetic system can be considered as a combination of devices that provide electrical stimulation to neurons as compensation for functional deficit or disease. An implantable neuroprosthetic system generally consists of three components; the external component, called the external control unit (ECU), the implanted electronic circuitry (IEC), and the stimulating or recording electrodes. The ECU consists of a PC that interfaces with custom or off-the-shelf hardware and a communication link mechanism, to send digital data/instructions to the hardware. For a stimulating neuroprosthesis, the IEC hardware then converts the digital instructions into analog waveforms for driving the

electrodes. These waveforms then stimulate the neurons via charge injection through the implanted electrodes. Such a system is pictorially depicted in Figure 1.1.

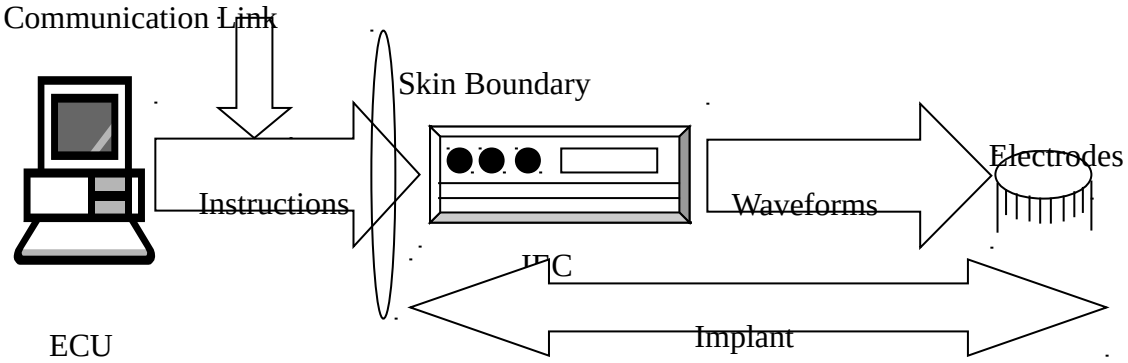


Figure 1.1. A Neuroprosthetic System with most Functionality Implanted.

Figure 1.1 shows that most of the hardware functionality is implanted. This requires that the IEC hardware be flexible to the kinds of instructions it can accept, thus demanding a complex data structure so that a variety of commands can be accommodated Having this high level of flexibility is a major requirement due to the fact that changing the functionality, or replacing a unit with a new one requires the risk of surgery, which should be avoided.

Addressing and instructing a large number of electrodes requires a higher speed and bandwidth than provided by the widely used PC based RS-232 or printer parallel buses. Used in this study, the cortical stimulator, for a visual prosthesis, is capable of stimulating, in its current form, 256 electrodes. It is expected that 4 stimulators would be implanted and combined for a 1,024-channel system. The increase in the number of addressable channels, over simpler devices like a cochlear implant (8-22 channels) has lead to an increase in the demand for bandwidth between the PC and the implanted

Implant

devices. To update 1024 instructions, one for each electrode, in a reasonable small amount of time requires an unusually wide bandwidth for the command link. An estimate of the required bandwidth can be obtained as follows. Each instruction in the present form requires 2-bytes of storage. So, 1,024 instructions will require a storage and transfer of 2,048 bytes. The transfer rate of these bytes will depend on the refresh rate that is the average in visual applications. A conservative refresh rate of once every 10ms can be used. Thus the transfer rate in this case would be, 2,048 bytes/ 10ms. This number can be reduced to 204,800 bytes/s or 204Kb/s, at the absolute minimum. This figure only includes updating the frame buffer. It does not include the instruction that will command the device to use the buffer. A variety of different commands can be implemented, for instance some instructions can start, stop the stimulation, give a range of electrodes to stimulate, etc. In addition to the above speed requirement, a communication link must provide for a conservative amount of excess bandwidth that will take care of communication bottlenecks that are not quite obvious at the start of any research project. The various available communication links are discussed in section 1.2. It has been determined through detailed analysis that in the visual prosthesis project the link operate at 1.2Mbps.

1.2 PC COMMUNICATION LINKS

The available literature on the subject reports a variety of implantable neuroprosthesis systems and communication strategies. Reviewing the research conducted over a 16 year period shows that these techniques are necessarily technology dependent. A more detailed comparative analysis of these research projects is done in Chapter 2. An overview of each technique is given below.

Embedded System: An embedded system is a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function. The hardware usually comprises of a microprocessor, random access memory (RAM), read only memory (ROM), digital-to-analog converter (DAC) and analog-to-digital converters (ADC). The ROM is used to store program instructions, which direct the embedded system to perform intended functions. Since most embedded system software is developed on IBM-PC's and Sun Microsystems workstations, the parallel port or the serial port is used to download the program instruction object code into the ROM. The RAM is used to store run-time data generated during the functioning of the system. The DAC is usually used to convert digital signals into analog form. These converters are usually used in systems where digital data is converted into analog form, for instance in digital cell phones and television sets. The ADC can be used with additional circuitry to accept interrupts and process them. The data transfer speeds of an embedded system vary depending on the processor speed and the data bus width that the system supports. Most embedded systems have a maximum clock speed of 5-10 MHz and a data bus width of 8-16 bits. The ideal maximum data transfer speed in such systems is about 20 Mbps.

RS-232 Serial Port: The serial port is a standard port available on every IBM-PC. It can be accessed using C / C++ libraries and is a low cost, low performance communication solution. The serial port can support a maximum of about 120 Kbps.

LPT Port: The parallel port (LPT) enables data transmission in parallel bytes, instead of serial bit stream like the RS-232 port. It can also be accessed using C / C++ libraries. The speed of the parallel port is up to 200 Kbps.

PC bus (Custom Hardware): Custom designed hardware can be directly interfaced with the PC bus. This requires the custom hardware to be electrically and functionally compatible with the PC bus. A designer has to make sure that the timings of all the signals are within a safe error margin mentioned in the particular bus specification.

PC bus (Commercial Hardware): Commercial hardware can be directly used with the PC bus, without knowing how the bus works, electrically or functionally. Operating system software and drivers are provided along with the hardware card. Programming can be done using standard programming languages such as C/C++ or using Labview or Matlab. The data transfer speed of the PC bus is about 40Mbps.

PC bus (Hybrid Hardware): The PC bus can be used as a combination of custom and commercial hardware cards. For the custom hardware, all electrical and functional and timing requirements must be met as discussed above.

USB (Commercial Hardware): Using the USB to exchange information between PC and hardware is relatively new. This architecture uses commercial USB compatible hardware to connect to the PC. Along with the hardware, operating system drivers are also provided. The maximum data transfer speed of such systems can be 480 Mbps.

Table 1.1 shows a comparison between all the communication links described above. For the cortical stimulator system under consideration, the only communication links that give us the required bandwidth are the PCI bus specialized hardware, the USB and firewire. Out of these the PCI bus based specialized hardware is rejected as it is not available with every PC and requires the use of a special card inserted into the PC. Firewire and USB are both new communication links aimed at high bandwidth transfers.

This research used USB 1.1 instead of 2.0 and firewire due to the easy availability of a USB 1.1 core and better support with respect to the hardware IP core and the device and application software.

Table 1.1. Summary of Various Communication Links.

Communication Links	Transfer Speed	Positive Attributes	Negative Attributes
Embedded Systems	20Mb/s	1) Developer has control over every aspect of implementation.	1) Complex to implement.
Serial Port	120Kb/s	1) Can have long cables, and requires only a few data lines.	1) Slow
Parallel Port	200Kb/s	1) Easier to implement, compared to serial port.	1) Uses large number of data lines.
PCI Bus (Specialized Hardware)	40Mb/s	1) High functionality, speed. 2)Good development support.	1) Too specialized, not available with all PC's
USB 1.1/2.0	12Mb/s (1.1) 480Mb/s (2.0)	1) High speed. 2) Good development support. 3) IP Cores available.	1) High initial learning curve. 2) Involved device driver development.
Firewire	400Mb/s	1) High speed. 2) Good development support.	1) Lack of freely available cores. 2) Initially developed for high speed video and audio applications.

CHAPTER 2

REVIEW OF PREVELANT COMMUNICATION LINKS

The previous chapter introduced various different architectures that have been used in the past and ones that are currently prevalent. A description of each research project and how different hardware architectures have been used in the past is included in this chapter.

2.1 OVERVIEW OF VARIOUS HARDWARE ARCHITECTURES

I. Embedded Systems.

1. A computer controlled vest for cardiopulmonary resuscitation (CPR) [2]. Objective: The objective of this research was to design and implement an embedded controller for vest Cardiopulmonary Resuscitation (CPR).

Hardware: The hardware consists of three different modules. The first one is the PC that is used to download program code to the second module. The second module is an embedded computer. It consists of an Intel 8088 microprocessor, ROM (to store the execution program), RAM (to store temporary data generated during runtime), a universal transmitter/receiver to enable RS-232 communication between the PC and the embedded computer, a timer for interrupt generation, an A/D converter and I/O ports. The third module is the power stage circuit that interfaces with the I/O ports. This stage drives the valves that control the flow of air in and out of the CPR vest.

Software: Two different programs are used in this system. One collects and stores the data and the other one analyzes the data and prepares it for use. The

first program collects data from the user. This data directs the timing and durations of the valve opening and closing. Once the user enters this data, it is sorted chronologically. This information is stored as a file in the PC and is also sent to the embedded computer by the second program. The embedded computer then controls the CPR vest as directed by the user input.

2. A portable neuromuscular stimulation system for use in paralyzed upper extremities [4]. Objective: This research aims to design a portable system for neuromuscular stimulation of paralyzed arms.

Hardware: The hardware for this system was located in three separate PCB's. These were for the processor, signal conditioning and stimulation output respectively. The CPU used was a Motorola CMOS MC146805E2 microprocessor. An 8-bit programmable timer, 112 bytes of RAM, a 64Kbit EPROM, eight channels of ADC and DAC were implemented on the same PCB. The signal conditioning hardware included two channels of gain and low-pass filtering. The stimulation output circuitry was located on the third circuit board and consisted of intra-muscular electrodes that are implanted percutaneously. This system uses input from the user to function. The input goes to the signal conditioning hardware and then through an ADC to the microprocessor system, which makes the decision on the stimulation output.

Software: The system software was developed on a DEC minicomputer system and was written in assembly language. A cross assembler generates the object code for the microprocessor. A PROM programmer is then used to enter the object program into the EPROM.

3. System architecture for a digital signal processor based microcomputer for use in a multielectrode cochlear implant system [16]. Objective: This research aimed to implement a multielectrode cochlear implant system.

Hardware: The hardware was based mainly on a Texas Instruments DSP processor. It has 4K of 16 bit program memory, a commercial codec chip, an interface chip for the parallel DSP and the serial codec chip, analog conditioning chips and an RS-232 interface that is used to download program code into the ROM.

Software: DSP techniques were used to extract features from the speech signal.

4. A microprocessor-based data-acquisition system for measuring plantar pressure from ambulatory subjects [25]. Objective: The aim of this research was to design a microprocessor based data acquisition system for measuring pressure data from ambulatory subjects.

Hardware: The portable microprocessor based acquisition system consists of 14 polymer pressure sensors, 14 amplifiers, 8-bit ADC a Hitachi microprocessor, an 8-kbyte CMOS ROM, four-32Kbyte CMOS RAM's and interfacing I/O circuits. Data stored in the portable unit are downloaded to an IBM-PC through the parallel LPT port.

Software: The software download to PROM is not mentioned in the research paper.

5. A microprocessor-based multi-channel stimulator for skeletal muscle cardiac assist [6]. Objective: To design a microprocessor based multi-channel

stimulator for skeletal muscle cardiac arrest. This is a treatment for chronic heart failure.

Hardware: The system is built using the Motorola MC68HC811 microcontroller. It is used to send control signals to an analog module to generate desired pulse sequences. Pulse sequences are defined using software and are downloaded into the microcontroller using the RS-232 serial port. The microcontroller has a built in serial port, 256 bytes of RAM, 2K of EEPROM, an 8 channel ADC and various timers.

Software: The software was designed as an easy to use graphical user interface. It specifies pulse sequences. These sequences are stored in a file as events that the microcontroller executes.

6. CMOS Neurostimulation ASIC with 100 channels, Scalable Output, and Bidirectional Radio-Frequency Telemetry [23]. Objective: This research designed, implemented and tested a 100 channel Neurostimulation circuit comprising of a CMOS ASIC chip. A radio-frequency communication link for power and data was also designed. The ASIC was designed primarily as a treatment of degenerative disorders of the retina.

Hardware: The overall system comprised of an external image processor, an external encoder/transmitter, internal receiver/decoder stimulator. An image captured by a CMOS camera, is processed by the external image processor, which processes the image into a 10x10 array of pixels. Within the external encoder/transmitter, each pixel is translated into an encoded RF telemetry sequence. Upon receiving the RF signal the implanted internal receiver/decoder stimulator decodes the instruction received and stimulates the appropriate electrodes.

Software: A programmable protocol extracts and pixelizes the acquired image into a 10x10 pixel array.

7. A Wireless Implantable Multichannel Digital Neural Recording System for a Micromachined Sieve Electrode [1]. Objective: This research developed a wireless implantable Multichannel digital neural recording system for a micromachined sieve electrode.

Hardware: The hardware consists of three modules, a microcontroller with transmitter electronics, a receiver circuit with instruction decoder and the electrode module. A Motorola 68HC11 microcontroller was used to generate serial encoded data to be transmitted over the inductive link. Based on the channel address and mode of operation different instructions were generated, transmitted, decoded and executed, thus generating an appropriate waveform on the electrodes.

Software: The software design for the research was not discussed.

II. RS-232 Serial Port

1. A Programming and Data Retrieval System for an Upper Extremity FES Neuroprosthesis [10]. Objective: This research aimed to design a stimulation and data retrieval system for upper extremity stimulation system.

Hardware: The hardware consists of a stimulation unit that has electrodes and a shoulder position transducer, an electrical isolation pod, an interface module and an interface module controller box for user input. The PC is used to download software into the interface module. The RS-232 bus is used to

communicate with the interface controller. The PC provides for real-time software programming and data retrieval from the stimulation system.

Software: The software was developed on the IBM-PC. It provided for real-time communication with the stimulation system

2. Accuracy of Drug Infusion Pumps Under Computer Control [7].

Objective: This research designed a prototype system to automate drug infusion. To do this a microcomputer was interfaced to a drug infusion pump, through a serial communications interface. The flow rate of three commercially available drug infusion pumps with an internal or add-on serial communication interface was tested under computer control.

Hardware: The pumps were connected to the PC through the RS-232, to compare the available infusion pumps in the market.

Software: The software design for the research was not discussed.

3. A Custom-Chip Based Functional Electrical Stimulation System [3].

Objective: This research designed a functional electrical system based on a custom ASIC chip. Using this system up to 32 chips can be connected serially to a host computer.

Hardware: In this design, up to 32 chips can be connected to the RS-232 serial port. Each chip can be addressed individually. Each ASIC chip is able to work in either master or slave mode. Each system requires one ASIC chip to be in the master mode with oscillators attached. All the slave chips derive their clock from the master chip. In this configuration, one address is required for each individual slave chip. Since each chip can control/address 8 stimulation

channels, this configuration results in an address space of 256 independently controllable stimulus channels per communication link.

Software: The software uses library calls to transmit real-time program information to the chips

III. LPT Parallel Port

1. A Telemetry System for the Study of Spontaneous Cardiac Arrhythmias [19]. Objective: This research designed a data acquisition system to gather data relating to cardiac arrhythmias.

Hardware: The hardware consists of two main components. The implantable unit and a back pack unit. The implantable unit consists of analog input electrodes, multiplexers and an ADC. The back pack unit consists of a custom designed serial card, that converts serial data from the implanted electrodes to parallel data that can be read by the PC parallel port. The received data is then processed by a CPU and prepared for transmission via a wireless LAN card. The data was received from the test subjects directly on the LAN, and thus the localization of data analysis software was eliminated.

Software: The software was written using a commercially available data viewing and analysis language PV-Wave. Special routines were implemented to provide custom viewing functions and to speed input/output and plotting functions.

IV. PC Bus: Custom Hardware

1. A 16-channel 8-Parameter Waveform Electrotractile Stimulation System [11].

Objective: To study the psycho-physiological performance associated with various stimulation waveforms by designing a general-purpose electro-tactile stimulation system.

Hardware: The stimulation system consists of a waveform generator, a PC, and analog system (ADC and DAC's), voltage to current converters, knobs/sensors and electrodes. The knobs/sensor analog data is converted to digital format using the ADC/DAC's in the analog module. The PC through the PC bus reads this digital data. The PC then outputs timer data to the waveform generation module. Once the waveform generator determines the wave shape, it is passed on to the voltage-to-current converter and then to the electrodes.

Software: the PC through the connected bus controls the entire system. Customized software package translates a user input file containing commands for all waveform parameters. All software was written in Turbo C and Turbo assembler for time critical tasks.

2. Computerized Trancutaneous Control of a Multichannel Implantable Urinary Prosthesis [20].

Objective: This research describes a PC interface of a multi-channel, implantable, urinary prosthetic device.

Hardware: The hardware for this system consists of six modules. The first three are an IBM-PC, a microcomputer hardware interface, and an AM modulator. The second group consists of an AM demodulator, an AC to DC

converter and a multi-channel CMOS chip. The outputs of the CMOS chip are connected to electrodes. The microcomputer hardware interface is used to convert parallel data from the PC bus to serial data that is used by the AM modulator for transmission. Once the data is received by the AM demodulator, it is passed onto a multi-channel CMOS chip, which is basically a microprocessor that executes 24-bit command words at 300Kbits per second. In the output stage, the CMOS chip contains control and current source blocks to interface with the implanted electrodes.

Software: The software designed was a multifunction program that allowed the user to communicate with the stimulator hardware. All the I/O tasks were programmed in assembly language. The data analysis and processing tasks were designed in Pascal. The software was designed to accept commands from a basic user interface, or from a command file.

V. PC Bus: Commercial Hardware

1. A New Approach to Man Machine Communication for Computerized Microscopy [13].

Objective: The research aimed to design a new computerized microscope. This microscope was fitted with objective and stage encoders and a built in high-resolution computer display to superimpose dialog, drawing and messages onto the optical microscope image.

Hardware: The hardware consisted of a microscope, a video monitor driven by a VGA standard graphics display card, encoder card to control the x and y position of the microscope. These cards communicate with an IBM-PC through the PC bus.

Software: The software is structured as a collection of different modules. One module is used to mark any object of interest. Another module is used to calculate the size of a particular object. The software can also display and store already examined portions of the sample under observation. Another module can be used to take printouts. The software is also capable of sharing the data stored on an internal office network.

2. A Computer-Controlled Research Ventilator for small Animals: Design and Evaluation [22]. Objective: This research aimed to design a computer-controlled ventilator for small animals.

Hardware: The hardware for the system consisted of an IBM-PC, DAC and ADC cards installed on the IBM-PC, a linear motor and a linear motor power amplifier. Three valves were also used to control the airflow into the animal compartments. These valves were controlled using a DAC. The cylinder (airflow) and tracheal pressure was measured using a pressure transducer and converted to digital format using the ADC.

Software: The software design for the research was not discussed.

3. A New Video-Synchronized Multichannel Biomedical Data Acquisition System [24]. Objective: This research designed a data acquisition system for bio-medical data. The system was video-synchronized and simultaneously acquired data with video time codes on a hard drive.

Hardware: The system used a video camera connected to a video tape recorder (VTR). The VTR was connected to a TV monitor and to an interface board. Biomedical data is also routed to this interface board. This interface board is

connected to an IBM-PC compatible Data input/output card (national instruments). The system records storage-intensive video images onto a videotape and simultaneously acquires biomedical data and video time codes onto a computer hard drive.

Software: LabView graphical programming was used to program the data acquisition, processing, storage and replay and VTR control.

4. A Multichannel Continuously Selectable Multifrequency Electrical Impedance Spectroscopy Measurement System [8]. Objective: To design a multichannel, multifrequency electrical impedance spectroscopy (EIS) measurement system.

Hardware: The EIS was designed to be modular to enable upgrade and modification of any component as necessity dictated. The computer used to control the EIS was a 200 MHz Pentium pro. EIS channel modules were implemented on custom PCB's. Each PCB controlled 8 channels. Communication between the PC and the EIS cards was achieved using a commercial digital I/O card. A waveform generator was used that was capable of generating arbitrary functions by direct signal synthesis. Data acquisition was also performed using a commercial board that had 4 input channels, with a 200KHz rate and a 16-bit resolution.

Software: The software was written using libraries provided by the commercial board providers. The hardware interface software for the EIS was implemented as an ActiveX control in C++. The user interface was designed and implemented in Visual Basic.

5. A Fast digitally Controlled Flow Proportional Gas Injection System for Studies in Lung Function [12]. Objective: To design a device used for gas injection in mechanically ventilated patients.

Hardware: The system included a PC, an ADC, a pressure sensor demodulator, flow sensor and a valve array. The flow sensor detects the pressure of the gas and the pressure transducer converts this information to an analog signal. The PC then converts this analog signal into digital format for use in determining the pressure. The software in the PC then regulates the valve array to increase/decrease the amount of gas flowing into the flow sensor, and thereby to the patient.

Software: The software design for the research was not discussed.

6. Computer Controlled Mechanical Stimulation of the Artificially Ventilated human Respiratory System [15]. Objective: To design a computer controlled artificial lung to simulate various lung pathologies.

Hardware: The hardware used an existing mechanical simulator including the necessary sensors, actuators, interface electronics and controllers. The main compartment is an air compartment with a piston that can be controlled using an electrical motor. The air compartment was connected to a flow-resistance compartment. The resistance compartment also has a resistance sleeve to control the resistance of airflow. The functioning of the system was studied at different flow resistance settings. The flow resistance sleeve was positioned using a servo-motor through a ADC. The ADC was also used to input the various physical parameters of the resistance compartment and converted into digital format. A motion controller card was then used to run a servo- motor for the main air compartment.

Software: The real-time software was written in MATLAB by communicating with the interface card using the real-time toolbox available with MATLAB.

7. BCI2000: A general purpose Brain-Computer Interface System [21].

Objective: To design a universal computer-brain interface model to assist severely motor-handicapped patients.

Hardware: The system model was designed using ADC, which received amplified and filter brain EEG signals. The computer software then processed the signals internally. The system model was then implemented using different hardware components (PC and Data acquisition boards). Performance was measured using the different hardware components. The systems were compared for output latency, jitter, clock jitter and processor load.

Software: The software was implemented in the C++ libraries provided by the board manufacturer.

8. Development of Brain-Computer Interface: Preliminary Results [18].

Objective: This research aimed to evaluate the feasibility of using EEG signals for control and communication with a computer, thereby moving animated objects on the computer screen.

Hardware: The hardware consisted of a PC, EEG amplifiers and a data acquisition card. The subject is placed in front of the screen, and gel filled electrodes are placed on specific, predefined locations on the scalp. An EEG is used for signal amplification and the acquisition card is used for signal digitization. The computer also has a video card the splits the video output to two high-resolution monitors.

Software: The software design for the research was not discussed.

9. Implementation of a Telemonitoring System for the Control of an EEG-Based Brain-Computer Interface [17].

Objective: This research presents a remote monitoring system for an EEG based brain-computer interface.

Hardware: The hardware consists of three major components, the supervisor system, the patient system, and network system. The brain computer interface consists of a laptop, a National Instruments data acquisition card, a PCMCIA card and an EEG amplifier. This system is connected via a network cable and a network card to a multimedia PC. The monitor system at the supervisor end serves as a monitoring station for training purposes.

Software: All three systems use the Microsoft Windows operating system. The brain computer interface (BCI) is programmed using MATLAB and SIMULINK. Software like PCAnywhere and Netmeeting were used for training purposes.

VI. PC Bus: Hybrid Hardware

1. A Real-Time Experimental Prototype for Enhancement of Vision Rehabilitation using Auditory Substitution [5]. Objective: This research designed a prototype system for the vision rehabilitation using auditory substitution.

Hardware: The hardware consisted of the following components: a miniature camera to capture visual stimulus, a video digitizer, 2 sound production boards (one for experimenter and one for subject, each is connected to a

headphone) and video monitors. The image digitizer is connected to the PC through the bus and is run continuously in frame grabbing mode. The 2 sound cards were custom designed using off-the-shelf music processors and were also interfaced with the IBM-PC through the bus.

Software: The software was written in C. The software initialized all the components of the system and started the frame grabber. The image acquired by the camera was processed and displayed and then converted into sound using custom algorithms. The sound amplitude was then transferred to the sound cards through the PC bus and then output to the headphones.

2.Wireless In-Shoe Force System [14]. Objective: This research presents a wireless in-shoe force system to acquire, process and transmit foot-floor force information that has been proven feasible for use with normal and paraplegic subjects.

Hardware: the system consists of various sub-systems: insole, transmitter, receiver and PC/Operator interface. The insole measures the actual force applied between the foot and the floor at four or six key points under the foot. Force applied to the foot, gets converted into voltage, which is further processed and used as an input into an analog to digital converter. Digital voltage readings are then used by a microcontroller to compute the actual force in pounds. A transmitter system then transmits the data to an external receiver. The receiver formats the received data into an appropriate data structure required by an external processing unit. The PC/Operator subsystem prepares the transmitter for data acquisition by calibrating the force sensors for a particular person. The subsystem consists of an IBM compatible PC and a special serial interface. The transmitter is calibrated using a conventional

serial interface. The program is downloaded into the transmitter using a serial connector.

Software: The software design for the research was not discussed.

VII. USB: Commercial Hardware

Brain-Computer Communication and slow cortical Potentials [9].

Objective: To design and implement a brain-computer interface using slow cortical potentials.

Hardware: The hardware consisted of a PC connected to an EEG machine through the USB. The system is designed for used in a closed loop. The EEG acquired brain signals and the computer processed (amplified and filtered) these signals and accordingly the brain-computer user interface performed certain functions. The occurrence of certain events then triggered an eye movement signal that proceeded in the same fashion to the PC and performed another function and so on.

Software: The software design for the research was not discussed.

2.2 ANALYSIS OF HARDWARE ARCHITECTURES

We can see a trend in the above overview. The earliest studies were done using the embedded architecture, and then came the PC Serial and Parallel port. Years later the PC bus was extensively used as PC became widely used. Since in the early days of the PC, there were limited hardware options available, and most people chose to design custom built hardware to interface with the PC bus. Then as more and more commercial vendors provided various solutions, researchers started using commercial off-the-shelf hardware and sometimes used a combination of custom and commercial solutions. More recently

there has been significant shift in the communication interfaces available in a PC. The Universal Serial Bus (USB) is a serial port but with significantly higher bandwidth than the original serial port. Due to the higher performance of USB and the standardization of the port, many commercial hardware providers are developing USB based hardware that can be exploited by bio-medical researchers.

This research designed and implemented a custom USB device for use in a research project that is discussed in chapter 3. It also evaluates the performance of USB as it relates to the research and to other more general applications. The device implemented to complete this research can be easily modified to be used as a generic custom-built USB device.

CHAPTER 3

MOTIVATION FOR PRESENT WORK

The custom USB device mentioned in section 2.2 was designed and implemented for use in the Visual Prosthesis (VP) project currently under research in our laboratory. The VP project aims to design, fabricate and test a multi-channel transcutaneous, cortical stimulation system to be used in a prototype of an artificial vision system. A block diagram of the proposed prototype is shown in figure 3.1. The aim is also to provide a minimum of 256 implantable cortical electrodes. The figure in 3.1 is a diagram of a sub-module that addresses and stimulates only 64 electrodes. Four such sub-modules will increase the number of electrodes to 256.

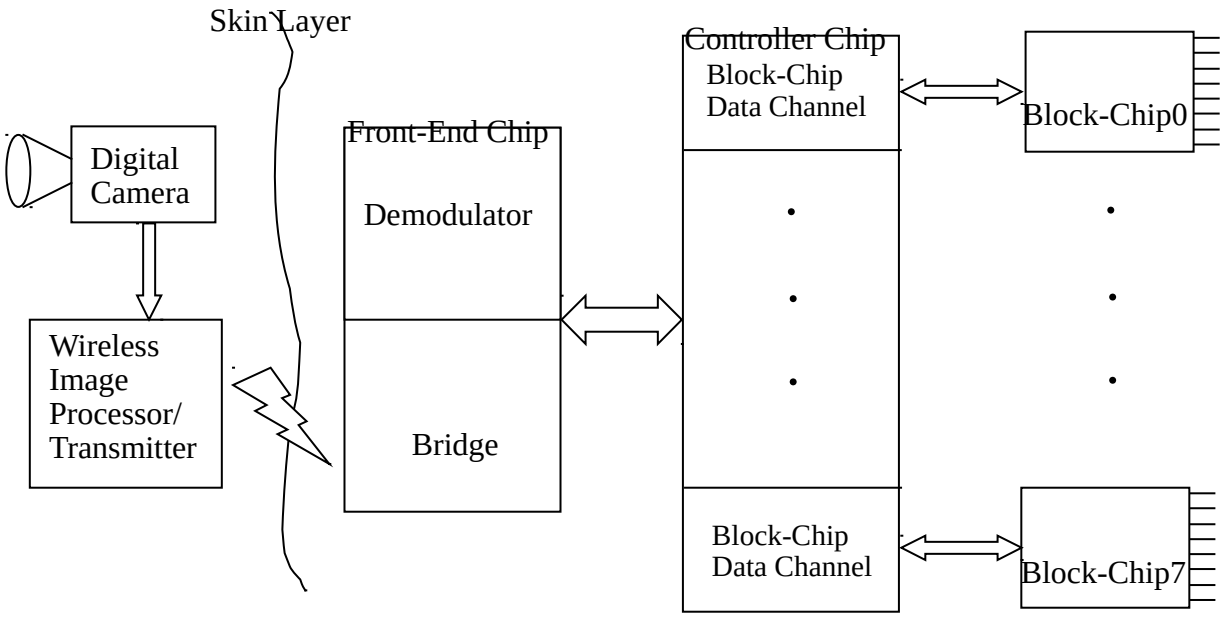


Figure 3.1 VP Prototype Block Diagram for a 64 electrode Sub-module.

Using a design employing smaller sub-modules has advantages that are beneficial for the VP project. The power supplies and transmission/receiving circuits of each sub-

module are separate, thus if the power regulation in one module fails, it does not fatally effect the entire implant. Redundancy is also a main feature of the design inside each sub-module. Each block chip has its own controller data channel, and each channel on each block chip has its own current driver, thus in the event of a current driver failing, the other channels of the block chip will still be usable.

The way such a device would work is as follows. The digital camera captures an image and transfers it to the wireless image processor/transmitter. This module pixelizes the image and makes a decision as to which particular cortical electrode should be activated, how much current should be applied and for how long. Instruction for stimulating single, or groups of electrodes would be sent over the wireless link to the implanted devices underneath the skin. The bridge circuit of the module then generates power for system-wide use and accepts reverse telemetry for transmission to an external module. The demodulator generates the master clock, data clock, timer clock, serial data stream and other control signals. These signals are then sent to the controller chip and are shared by all block-chip data channels. The controller chip is a finite state machine (FSM) that decodes the incoming instructions, issues the instructions to their corresponding destinations (8-channel block chips). The block chips then execute the instructions they receive by generating waveforms on one or more electrodes as specified, to generate a pixelized image for the patient.

To design the prototype system described above, various intermediate systems need to be devised and tested on real animal/human subjects. One such stimulator system was designed and extensively tested to acquire data on visual cortex view-field mapping. This system is described in detail in section 3.1 along with disadvantages when considering system flexibility.

3.1 VISUAL PROSTHESIS STIMULATOR SYSTEM

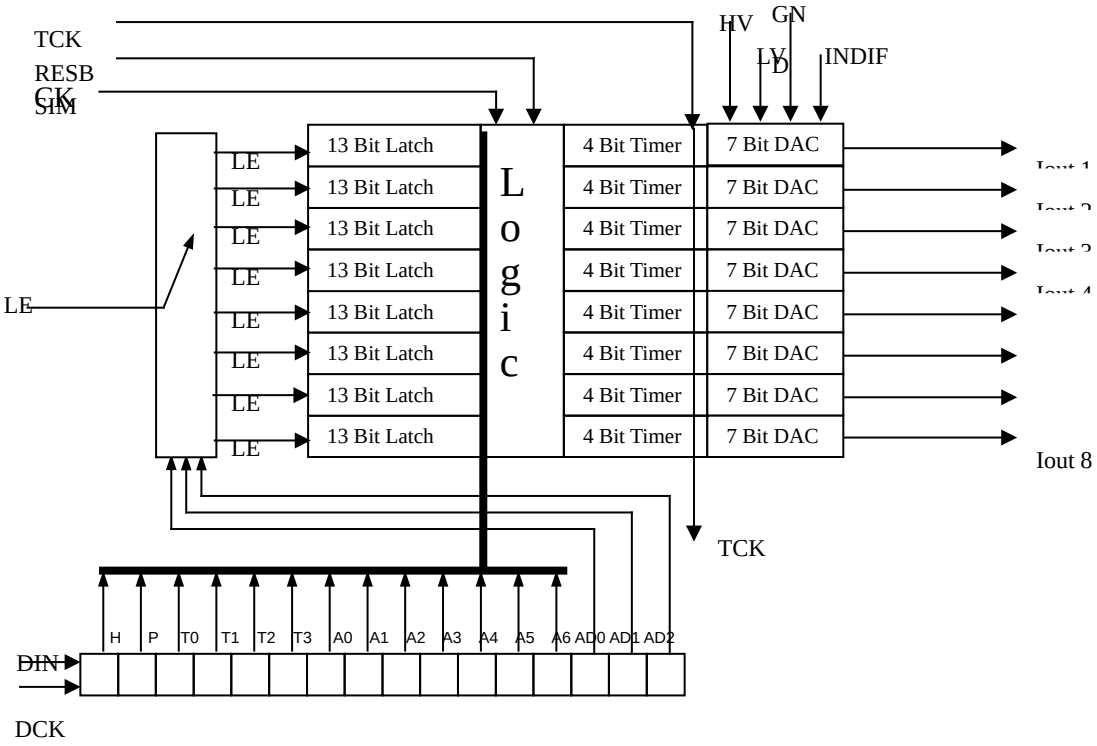
A benchtop, visual prosthesis stimulator system was designed, as part of an earlier phase of the visual prosthesis project to have the capability to address and control 128 electrodes. The goal of this phase was to create a benchtop stimulator to evaluate various stimulation techniques in an animal model, and to evaluate the design from an implantable system standpoint. The benchtop stimulator provided a basis for the design of the USB interface that is the subject of this work. This system was comprised of 16 block chips, each having 8 addressable electrode channels. The system was designed to optically isolate the input signals of the block chip from the computer. On most bio-medical systems an optically isolated implant is required to protect against unwanted voltage spikes on the computer power supply, affecting the signals going to the implant. Since the complete system was powered using a 12-volt lead-acid rechargeable battery, in addition to using the optoisolators, the implant was completely isolated from any AC power supply. In this initial benchtop system, a high-performance PC was used in conjunction with a commercial off-the-shelf digital input/output (DIO) card from Adlink technologies, as the interface to generate and issue instructions to the stimulator. The DIO card was programmed to output instructions to the stimulator system, using the vendor provided C++ and Visual Basic libraries. A graphical user interface was designed in Visual Basic to simplify the instruction generation process. To understand the limitations of this stimulator system, a deeper understanding of the system architecture is required. This includes the DIO card and the block chip architecture. Doing so will clearly demonstrate the limitations and provide in insight into the various available solutions to eliminate and effectively deal with those limitations.

I. **The NuDAQ PCI-7300 Digital Input/Output Card.** The DIO card is a PCI form factor ultra-high speed card with 32 input/output channels. It performs high-speed data transfers using bus-mastering DMA via the 32-bit PCI bus

architecture. The maximum data transfer rates can be up to 80MB per second. Extensive software support is also provided with the card. Software drivers for packages like LabView, HP VEE, DASYLab etc. are provided. Libraries for Borland and Microsoft C/C++, Visual Basic are also provided for both Windows and Linux platforms.

II. **Block-Chip Architecture.** The logic level diagram of block-chip circuitry, as shown in figure 3.2, has 5 input signal lines, and 4 power lines, 8 stimulation channels. Each channel consists of a 4-bit timer and 7-bit current output DAC. The DAC produces the required biphasic pulse and the timer controls the duration of the pulse. Generating a pulse requires a 16-bit data stream to be read in to the shift register. The first 3 bits that are read are the channel address lines, AD2, AD1 and AD0 respectively. These address lines set the multiplexer to the appropriate channel. The next 7 bits are the amplitude bits meant for the DAC and are read in as A6, A5, A4, A3, A2, A1 and A0 respectively. The next 4 bits are the timer bits that decide the duration of the output pulse and are used as inputs for the 4-bit Timers. The last two bits are the polarity and holdoff bits respectively. The polarity bit decides whether the output pulse will have the cathodic phase or anodic phase first. The holdoff bit is used for simultaneous stimulation. If it is set to 0, stimulation will start at the next rising edge of the TCK, after the rising edge of the LE. If it is set at 1, stimulation will not begin until the next rising edge of the TCK after SIM is asserted.

Figure 3.2. Block-Chip Architecture.



1. **Signal Description:**

   a) DIN: This is the serial 16-bit data instruction that is stored in the shift register.

   b) DCLK: This is the clock used to shift the 16-bit data into the shift register. The frequency of this clock was set to 5MHz.

   c) LE: This is the latch enable signal that latches the 13-bit waveform attribute bits into one of the 13-bit latches shown in figure 3.2.

   d) TCLK: This is the clock used by the 4-bit timers to output waveforms with the desired pulse width duration.

e) RESB: A logic low (0V) on the RESB line resets the 4-bit timers and the state machine logic circuit, thus terminating all stimulation pulses currently in progress.

f) SIM: The SIM line is used to generate pulses on multiple channels simultaneously. The rising edge of the SIM sets a flip-flop, and on the next rising edge of the TCK, all the channels with the holdoff bit set will start stimulating simultaneously.

2. Timing Description. Figure 3.3 shows the timing characteristics of the block chip along with minimum times for certain signal lines. Before a block chip can start stimulating, it needs to be provided a 16-bit instruction as described above. When the DIN signal is settled in any one particular state, a rising edge occurs on the DCLK signal, thus shifting in the state on the DIN signal into the shift register. This keeps repeating for 15 more DCLK cycles to shift in all the 16 bits. Once the instruction has been read in, the LE signal is used to latch the data into the 13-bit latches in the block chip. The remaining 3-bits are used to decide the address of the channel, for which the data is meant. The LE pulse is supposed to go high at least 200ns after the last data bit has been read in and it should remain high for at least 100ns. In addition, the minimum time between the rising edge of the LE and the next rising edge of the TCLK should be 200ns. If this timing constraint is not met, the stimulation starts normally at the next rising edge of TCLK.

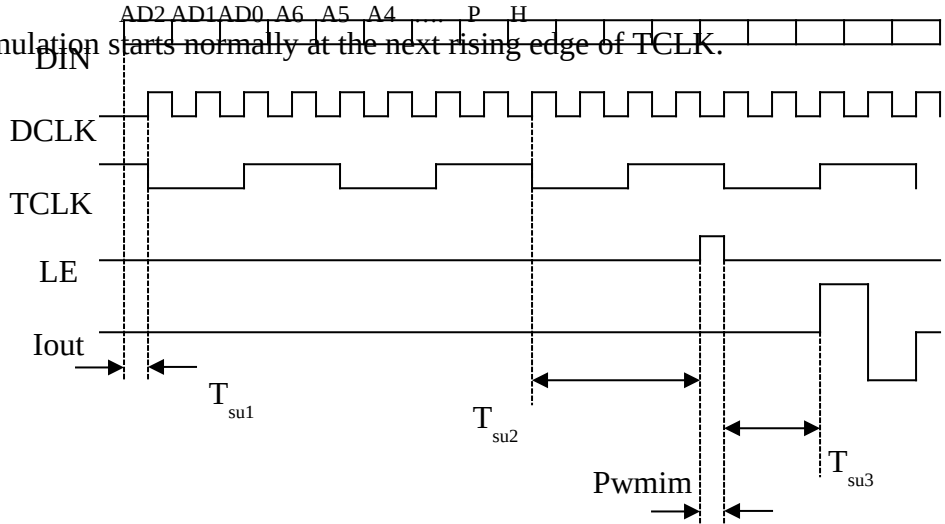


Figure 3.3. Block-Chip Timing Diagram.

T_{su1} = Minimum time data must be stable before rising edge of DCLK.

T_{su2} = Minimum time between clocking in of last data bit and rising edge of LE.

T_{su3} = Minimum time between the rising edge of LE and the rising edge of TCLK. If this time constraint is not met, stimulation will begin normally at the next rising edge of the TCLK.

III. **DIO card, Block-Chip Interface.** The stimulator system was designed to address and stimulate 128 electrodes. This required using 16 block chips, since each block chip can address 8 channels. The system diagram of the complete stimulator system is shown in figure 3.4. Since the DIO card is PCI based, it conforms to the electrical, functional and timing specifications of the bus. The PC used had a SCSI bus, as most of the waveform instructions were stored on the hard drive. To reduce the latency of transferring the instructions from the drive to the DIO card, the faster SCSI bus was used. The stimulator bus consisted of the above mentioned block chip signals. Figure 3.5 shows the stimulator bus in more detail.

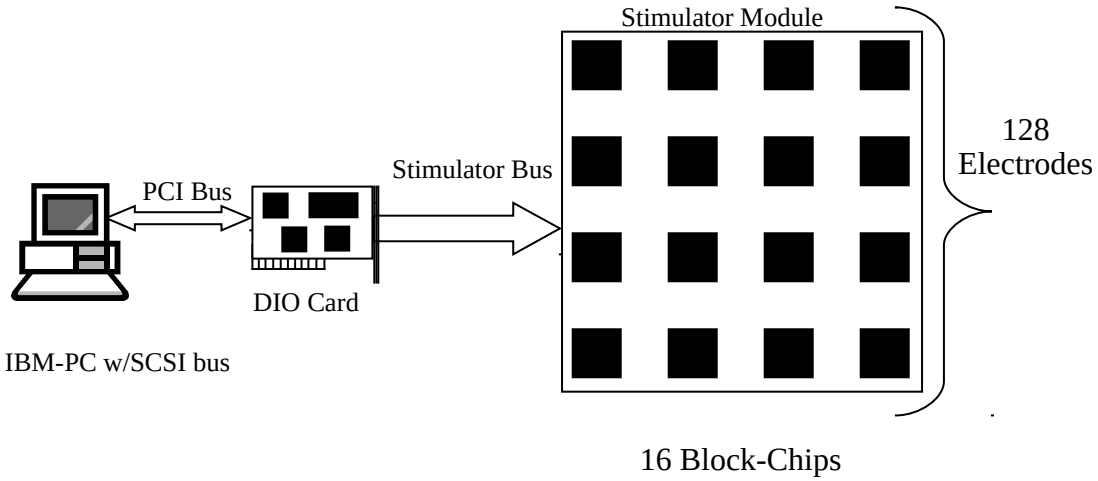


Figure 3.4. Top Level Stimulator Architecture.

Figure 3.5 shows how the bus is connected to the stimulator device. The signals that are common to each block chip, are as follows: DIN, DCLK, TCLK, RESB and SIM. The unique signals are the LE signals, i.e., each block chip has its own unique LE signal to let the user control which block chip is stimulated at what point in time.

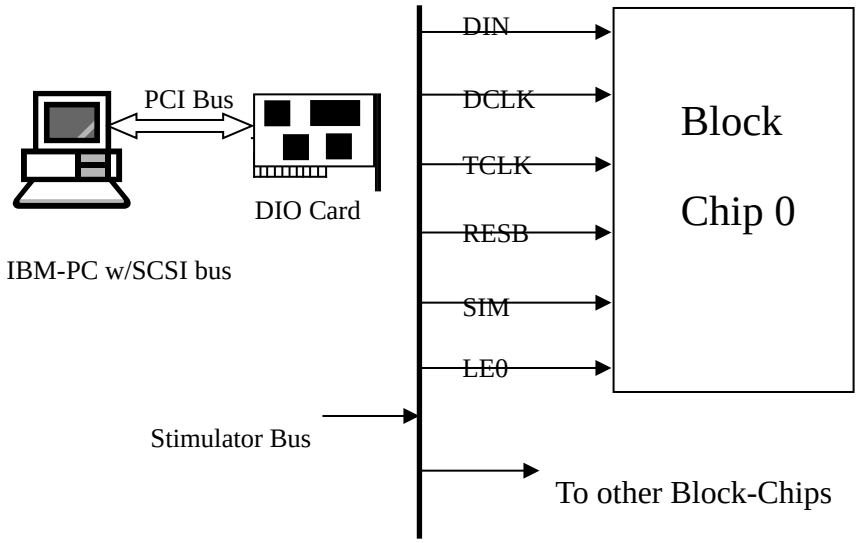


Figure 3.5.Stimulator Bus –Block Chip Interface.

Even though different electrodes in different chips may require waveforms of different stimulation parameters, the architecture uses a common serial data line DIN. Doing so is safe from a stimulation standpoint because of the fact that the respective LE lines that will actually trigger the stimulation as describe above. Hence for all the 16 block chips we need a bus that is 21 bits wide, five lines for DIN, DCLK, TCLK, RESB and SIM and 16 lines for LE0 through LE15.

IV. **Limitations of this architecture.** The DIO card used for the stimulator system has 32 input/output lines. Since the stimulator system used 21 lines, using the DIO card to control the stimulator system was a good solution. Future generations of stimulator systems will require more than 128 electrodes. The next generation stimulator systems as proposed in the visual prosthesis project and discussed above, have already been theoretically devised, however, due to unfavorable electrode density, implanting 1024 electrodes is years away. However, planning for the interface to communicate with a 256 channel system is presently underway. Since for 256 channels there would be 32 block chips, we would require 32 digital output lines in addition to the 5 common lines for the whole system. This makes the total number of digital lines to 37 lines. The current choice of the DIO card is unable to satisfy these requirements. The solution lies in using 2 of these cards, using a card with more digital lines, or modifying the way in which LE signals are transmitted, since LE’s require most number of lines. We now look at each of these solutions. Using two DIO cards is certainly possible, but

given the cost of each card ($900 each) and the amount to computing resources used for using such a system and the complexity of programming each card, a simpler solution is required. In addition, use of the DIO card does not allow for easy portability between computers, especially for notebook computers,

Using a DIO card with more number of digital lines is delaying the inevitable. Eventually, stimulator systems would require more and more number of electrodes. When that happens, a similar problem would arise.

The other solution is to generate the LE in a different fashion. The LE's could be transmitted serially on one digital line. The modification requires the design and fabrication of another chip with a 32-bit shift register. Additional logic would decide which block chip needs to be sent an LE signal. Even if we divide the LE's into two groups and have two 16-bit shift registers, it would require an extra digital line and almost certainly more chip area. The chip would also require 32 output lines for the LE's going to 32 different block chips. As chip area increases so does its cost to fabricate. Another reason a big chip area or even an additional chip is a negative quality is that these chips will eventually be implanted into the visual cortex, and in case of implants, the smaller they are the more invisible they are to the user. Hence designing the next generation stimulator requires investigating new PC communication technologies, some of which are discussed in section 3.2.

3.2 ALTERNATIVE COMMUNICATION LINKS

It is clear that to design a stimulator system that is easy to use and program is the basis on which it will be accepted widely or not. To do so, the PCI bus based systems

need to be re-evaluated. One important factor of doing so, besides the above-mentioned technological hurdles, is the cost of these commercial off-the-shelf solutions. To reduce cost, one must consider the latest communication links that are standard with PC's today. These are the USB, Firewire, parallel port and the serial port. These solutions are not expensive as they are already proven communication techniques that have are a number of compatible devices on the market. The parallel port and the serial port are not good solutions for our problem, as they are slow and new generation neuroprosthetic chips require a lot of information at high speed to generate complex waveforms for stimulation. The other two solutions are firewire and the USB. Firewire is a good solution with up to 400 Mbps of data transfer speed. Firewire was rejected for our solution due to the fact that there are no freely available IP cores. A freely available USB IP core was found and was the main motivation of using the USB communication link instead of the firewire link. A brief description of the advantages of the USB is discussed in section 3.3

3.3 ADVANTAGES OF THE USB

One of the most important reason the USB was chosen for this research was the fact that all PC's today have USB ports available. The other important reason is that a readily available, open source implementation of USB in Verilog RTL was available for use. Using an open source IP core for the USB, reduced the amount of time required for initial development, as compared to designing a USB IP core from the ground up. It also provided with a previously used and tested core, which increased our confidence of success.

The other technology related advantages are:

I. **Speed:** It supports three speeds, 1.5MBps, 12MBps and 480MBps. The highest speed is 80MBps more than that of firewire.

II. **Reliability:** The hardware specification for drivers, receivers and cables eliminate most noise, in addition to the specification requiring CRC checks.

III. **Low cost:** Even though USB is more complex compared to older interfaces, the cables and connecters required are less expensive.

IV. **Availability:** All PC's developed today have USB compatible connectors.

V. **Flexibility:** USB provides for different kinds of data transfers, enabling its use for different kinds of peripherals.

VI. **Support:** Good support for developers, both software and hardware. It is also extensively supported by almost all major operating systems.

CHAPTER 4

THE UNIVERSAL SERIAL BUS

The initial development of the USB was seeded by three motivations, namely, connection of the PC to the telephone, ease-of-use and port expansion. Before the USB, it was well understood that the next generation of technology lay in the merger of

computing and telecommunication. The traffic of human centric and machine centric data depends on inexpensive communication links. Such a link already exists in the form of the Internet. Due to the fact that computing and telecommunication technologies developed in isolation to one another, an easy-to-use link between the two was needed. The USB was devised as the answer. To make it easy to use, the USB was designed to be plug-and-play. This was made possible by developing a large number of application and systems software for everyday electronic equipment like digital cameras, mice, keyboards etc. The extensive availability of an additional PC port enabled the explosive rise in USB compatible computer peripherals.

Due to recent advances in computer processing power, PC's are now capable of processing a lot of data. This led to the development of USB 2.0. User applications like digital imaging and video have demanded a higher bandwidth communication link with the PC, thus USB 2.0 was designed to transfer data at up to 480 Mb/s.

4.1 USB ARCHITECTURAL OVERVIEW

The USB connects devices with a host. The USB interconnect is a tiered star topology. A hub is at the center of each star and each wire segment is a point-to-point connection between the host and hub, or a hub and a function. This topology is shown in figure 4.1.

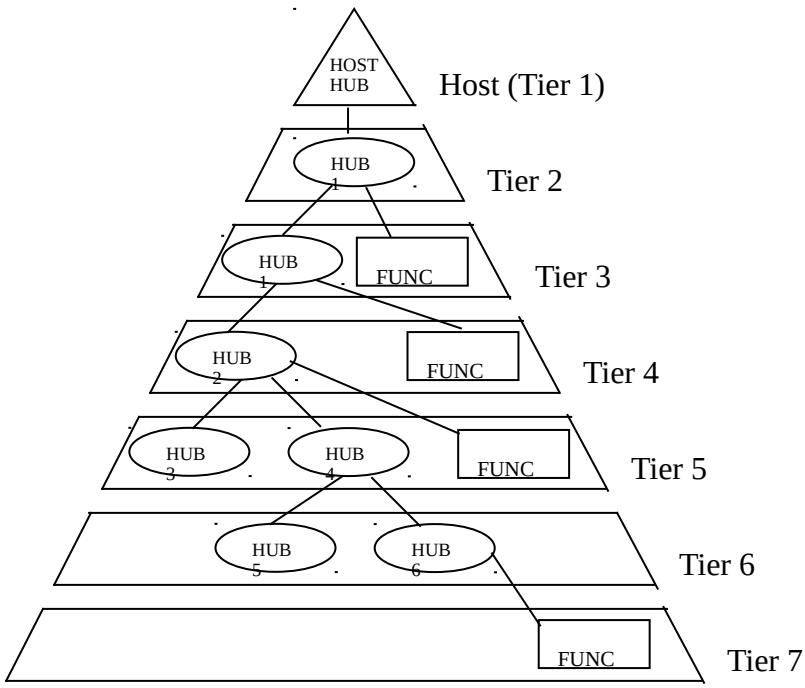


Figure 4.1. USB Topology.

Due to the timing constraints allowed to hub's and cable propagation times, a maximum of 7 tiers are allowed. In these 7 tiers, a maximum of 5 non-root hubs are allowed. In tier 7 only functions can be allowed. In a USB system there can only be one host. The USB interface in the host computer is called the host controller. A root hub is integrated within the host system to provide one or more USB port attachment points. USB devices fall into two categories: hub's that provide additional attachment points for other USB devices, and functions, which provide capabilities to the computer system.

USB devices conform to certain standards defined by the USB specification, namely their comprehension of the USB protocol, their response to standard USB operations such as configuration and reset. USB transactions take place on a 4-wire cable. The cable wire specification is shown in figure 4.2.

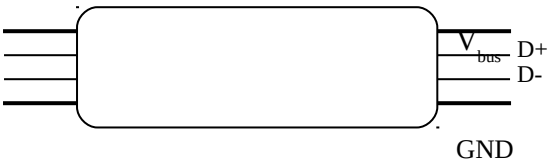


Figure 4.2. USB Cable.

There are three data rates available for USB transfers. High signaling rate is 480 Mb/s, full speed rate is 12 Mb/p and a limited capability low-speed signaling mode is also defined at 1.5 Mb/s. USB 2.0 host controllers and hubs provide capabilities so that full-speed and low-speed data is transmitted at high speed between the host controller and the hub, but transmitted at full or low-speed between the hub and the full-speed or low-speed device. Having this capability minimizes the impact that full-speed and low-speed devices have on the bandwidth available for high-speed devices. This is obvious from the fact that data travels at full or low speed on when absolutely necessary.

There are two ways in which USB devices can be powered. USB devices can be bus powered or self-powered. Bus-powered devices draw power from the power lines provided in the USB cable, as shown in figure 4.2. The power available for each USB port is limited to about 500mA, thus all the bus-powered devices cannot draw more than 500mA of current. If more devices need to be connected, they need to be self-powered. Such devices usually are wall-plug-in voltage regulators or battery powered systems.

The USB is a polled bus, i.e., the host controller initiates all data transfers. Most bus transfers comprise of up to three packets. The host controller sends a USB packet describing the type and direction of the transaction, the USB device address and endpoint number. This packet is called the token packet. The addressed USB device decodes the information in the setup packet and either waits for the next stage or initiates the next

stage itself, depending on the direction of the data transfer. This stage is also known as the data stage. The source of the data, as specified by the setup packet starts the transfer. The destination responds to the data stage by sending an acknowledgement, stating whether or not the transfer was successful.

The USB data transfer model between a source and destination on the host and an endpoint on a device is referred to as a pipe. There are two types of pipes: stream and message. Pipes have associated data bandwidth, transfer type and endpoint characteristics like directionality and buffer sizes. An endpoint can be considered as a source or a user of information/data. For instance, a simple USB device like a flash drive has a minimum of 2 endpoints. One endpoint acts as a sink, i.e., it accepts data from the host computer and stores it on flash memory. Another endpoint acts as a source, i.e., it sends data to the host computer. Each endpoint has its own pipe associated with it. Most pipes come into existence when a device is connected to a host computer. One message pipe called the Default Control Pipe always exists once a device is powered, in order to provide access to the device's configuration, status and control information.

4.2 USB TRANSFER TYPES

There are four data transfer types supported by the USB: control, bulk, interrupt and isochronous.

I. **Control Transfers:** Control transfers have two uses. They are used to facilitate transfers specified by the USB specification and used by the host to learn about and configure devices, and to carry requests defined by a class or vendor for any other purpose.

   Every device is required by the USB specification to support control transfers over the default pipe at endpoint 0. As discussed above, each transfer consists of three stages: Setup, Data (optional) and the Status stages. A stage could

consist of one or more than one transactions. At minimum every control transfer must have a Setup and Status stage. The use of the data stage depends on the kind of requests by the host or the device. All control transfers require that all information flow in both directions, the control pipe used both IN and OUT addresses of the endpoint 0. An IN transaction means that information travels from the device to the host, and an OUT transaction means that information travels from the host to the device. On other words all transactions are looked at from the host's perspective. In control read transfer the data in the Data stage travels from the device to the host and in a control write transfer, the data in the Data stage travels from the host to the device.

The token packet contains a PID that identifies the transfer as a control transfer. The data transfer contains information about the request, including request number. The USB specification defines 11 standard requests. Successful enumeration requires specific responses to these requests. Enumeration is discussed in detail in section 4.3.

The data size for control transfers can vary according to speed. Low-speed devices can have a maximum data size of 8-bytes. For full-speed the size could be 8, 16, 32 or 64 bytes. For high-speed the maximum data size must be 64-bytes.

The host makes its best effort to ensure that all control transfers get through as quickly as possible. The host reserves a portion of bandwidth specifically for control transfers: 10% for low and full-speed and 20% for high-speed transfers.

If a device does not return an expected handshake packet during a control transfer, the host tries again two times. If after a total of three tries no

response is received, the host notifies the software that requested the transfer and stops communicating with the endpoint until the problem is solved.

II. **Bulk Transfers:** Bulk transfers are used for transferring data when timing is not critical. Such a transfer can send large amounts of data without clogging the bus, because transfers defer to other transfer types and wait for bulk transfers until bandwidth is available. However, if there are no other pending transfer types, bulk transfers are the fastest.

Only full and high-speed devices can do bulk transfers. A bulk transfer consists of one or more IN or OUT transaction. Since a transfer's transaction must be all IN or all OUT, transferring data in both directions requires a separate pipe and transfer for each direction.

A full-speed bulk transfer can have a maximum packet size of 8, 16, 32, or 64-bytes. For high-speed a maximum of 512 bytes is allowed. The host reads the maximum supported size during enumeration. If the amount of data is more than the maximum allowed, then the data transaction is broken down into multiple data packets.

The host controller guarantees that bulk transfers will eventually complete, but does not reserve any bandwidth for them. Control transfers are guaranteed 10% bandwidth for low and full speeds and 20% for high-speed transfers. Interrupt and isochronous transfers use the rest of the bandwidth. Hence if a bus is very busy, a bulk transfer may take very long. If there are no devices that use interrupt or isochronous transfers, connected to the bus, bulk transfers are completed very quickly.

Bulk transfers use error detection, hence they are used in applications where transfer of correct data is required. If the device does not return an

expected handshake packet, the host tries up to two times more. Bulk transfers are designed in such a way that ensures that no data is lost. While they do not have any error correcting facility, they require that erroneous data be transmitted again till there are no errors.

III. **Interrupt Transfers:** Interrupt transfers are useful when data has to transfer within a specific amount of time. Typical applications are keyboards, mice and other human interface devices. The bandwidth available for interrupt transfers is limited for low and full-speed devices, but high-speed increases the limits and enables an interrupt endpoint almost 400 times as much data as full-speed. For low-speed devices, the maximum packet size can be any value from 1 to 8 bytes. For full-speed, the maximum packet size can range from 1 to 64-bytes. For high-speed the range is 1 to 1,024-bytes.

IV. **Isochronous Transfers:** Isochronous transfers are streaming, real-time transfers that are useful when data must arrive at a constant rate, or by a specific time and occasional errors can be tolerated. At full-speed, isochronous transfers can transfer more data than interrupt transfers, but there is not provision for re-transmission of data received with errors.

Examples of uses for isochronous transfers include encoded voice and music to be played in real-time. Unlike with bulk transfers, once an isochronous transfer begins, the host guarantees that the time will be available to send data at a constant rate, so the completion time is predictable.

Only full and high-speed devices can do isochronous transfers. Devices are not required to support isochronous transfers, but certain device classes do require isochronous data transfers.

For full-speed endpoints, the maximum packet size can range from 0 to 1,023-bytes. High-speed endpoints can have a maximum packet size up to 1, 024-bytes.

4.3 USB ENUMERATION

Before applications can communicate with a device, the host needs to learn about a device and assign it a device driver. Enumeration is the initial exchange of information between the host and the device, and is the process by which the host learns more about the device. The enumeration process includes, assigning an address to the device, reading data structures from the device, assigning and loading a device driver, selecting a configuration from the available options. Once this is done, the device is configured and ready to transfer data using any of the endpoints specified in its configuration descriptors.

During the enumeration process a device moves through different device states as defined by the specification: Powered, Default, Address and Configured. The steps described below are a typical sequence of events that occur during enumeration.

1. **The user plugs a device into a USB port:** This puts the device in the powered state.
2. **The hub detects the device:** The hub monitors the voltages on the signal lines of each port. Every USB device has a 15Kohm pull down resistor on either the D+ or the D- line. If the hub detects a pull down resistor on the D+ line, the device is full-speed device, if the pull down resistor is on line D- then the device is configured as a low-speed device.
3. **The host learns of the new device:** Each hub reports events on its ports to the host. When the host learns of an event, it sends the hub a Get_port_status request to find out more about the port.

4. **The hub detects the speed of the device:** The hub looks at the signal lines to determine what the speed of the device is, as described above.
5. **The hub resets the device:** After the speed is detected, the host asks the hub to reset the device. This is done by placing the D+ and D- lines at logic low. Normally the logic lines have opposite states, they are required to be placed at logic low to reset the device.
6. **The host learns if a full-speed device supports high-speed:** Detecting whether a device supports high-speed uses two special signal states. In the chirp J state, the D+ line is driven and in the chirp K state, the D- line only is driven. During reset a device that supports high speed sends a chirp K. A high-speed hub responds with alternating chirp K's and J's. When the device detects the pattern KJKJKJ, it removes its full-speed pull-up and performs all transfers at high-speed. If the hub does not respond to the device's chirp K, the device communicates at full-speed. Due to this reason all high-speed devices must be able to communicate at full-speed. For instance, if the hub does not support high-speed, a high-speed device should still be able to communicate with the host PC in full-speed mode.
7. **The hub establishes a signal path between the device and the bus:** The host asks the hub to remove the device from the reset state to the default state. The USB registers are in their reset states and are ready to accept control transfers over the default pipe at endpoint 0. The device can now communicate with the host using the default address 00h.
8. **The host sends a Get_Descriptor request:** The host requests the device to send it the device descriptor. This basically tells the host about how the host must communicate with the device, for the duration of the enumeration process. All types of descriptors are discussed in section 4.4.

9. **The host assigns an address:** The host assigns a unique address to the device. The address assigned earlier is a temporary address given to every new device being enumerated.
10. **The host learns about the device's abilities:** The host now requests all the other descriptors that are stored in the device. This enables the host to know how to communicate with the device and what the device capabilities are.
11. **The host assigns and loads a device driver:** Once the host goes through the descriptors, it tries to match the information in there to the information stored in its driver information files (for Windows). Once there is a match, the operating system, dynamically loads the driver.
12. **The host's device driver selects a configuration:** Once the device driver is loaded, it requests a configuration by sending a Set_Configuration command to the device. The device sets the configuration provided by the command, and the device is ready for use.

4.4 USB DESCRIPTOR TYPES

Descriptors are data structures of information that enable the host to learn more about the device. Each descriptor contains information about either the device as a whole, or about a specific functionality. To be compatible with the USB specification, all devices must respond to requests for standard USB descriptors. The device, hence, must store the information and respond to requests for the same in an expected format.

The first descriptor that is requested by the host is the device descriptor. It contains information about the device as a whole, and specifies the number of configurations available in the device. The configuration descriptor contains information about the device's use of power and the number of interfaces supported by the device. Each interface descriptor has associated with it zero or more endpoint descriptors. After the

device descriptor is sent to the host, the device receives a request for the configuration descriptor. After the host receives the configuration descriptor, it also gets to know the total number of bytes in all the descriptors except the device descriptor. The host then requests the configuration descriptor again, but this time it requests the device to send all the other descriptors associated with it. Hence, all the interface and corresponding endpoint descriptors are also sent in one request. Each descriptor type is described below.

I. Device Descriptor

Table 4.1. Device Descriptor.

Offset (Decimal)	Field	Size (Bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	Constant DEVICE (01h)
2	bcdUSB	2	USB spec. Rel number.
4	bDeviceClass	1	Class code
5	bDeviceSubClass	1	Subclass code
6	bDeviceProtocol	1	Protocol code
7	bPacketMaxSize	1	Max packet size for EP0
8	idVendor	2	Vendor ID
10	idProduct	2	Product ID
12	bcdDevice	2	Device release number
14	iManufacturer	1	Manufacturer string descriptor index.
15	iProduct	1	Product string descriptor index.
16	iSerialNumber	1	Serial number string descriptor index.
17	bNumConfigurations	1	Number of possible configuration.

bLength: The length in bytes of the descriptor.

bDescriptorType: The constant DEVICE (01h), used for the device descriptor.

bcdUSB: The USB specification number to which the device and its descriptors are compatible. For version 1.1, this value will be 0110h.

bDeviceClass: This field is for devices that belong to a class. Values from 1h to FEh are reserved for USB defined classes. Not all devices must belong to a class.

bDeviceSubClass: This field specifies the subclass for the device.

bDeviceProtocol: This field specifies the protocol that the device complies with.

bMaxPacketSize: This is the maximum packet size of the endpoint 0.

idVendor: This is a unique ID for a particular vendor. Vendors who pay a fee, are given a unique ID that they can use for their products. This value is also stored in the INF file for the device. The operating system matches this value with the value received from the device and it knows which device driver to load.

idProduct: This ID number is assigned by the device manufacturer to distinguish between different products.

bcdDevice: This is the device release number in bcd format. This value can also be used to decide which driver to load.

iManufacturer: This is an optional field. It points to a location, which stores string information about the manufacturer.

iProduct: This is also an optional field. It points to a location, which stored string information about a product.

iSerialNumber: This is the index of a string that points to the device serial number.

bNumConfigurations: This is the number of configurations the device supports. A particular configuration defines the device's capabilities and features. For instance, a digital video camera may be designed to function in two modes. One mode might be recording mode, and another might be playback mode. The camera can only be in one mode at a particular time. Thus the features of such a camera can be divided into two

different configurations, which can be loaded as the user so chooses. The software can thus function depending on which mode the camera is in.

II. Configuration Descriptor

Table 4.2. Configuration Descriptor.

Offset (Decimal)	Field	Size (bytes)	Offset (Decimal)
0	bLength	1	Descriptor size in bytes.
1	bDescriptorType	1	Constant CONFIGURATION (02h)
2	wTotalLength	2	Size of all data returned for this config in bytes
4	bNumInterfaces	1	Number of interfaces the configuration supports.
5	bConfigurationValue	1	Identifier for Set_Configuration and Get_Configuration requests.
6	iConfiguration	1	Index of string descriptor for configuration.
7	bmAttributes	1	Self power/bus power settings.
8	MaxPower	1	Bus power requirements.

bLength: The length in bytes of the descriptor.

bDescriptorType: The constant configuration (02h).

wTotalLength: The number of data bytes that the descriptor returns, including all the interface and associated endpoint descriptors.

bNumInterfaces: The number of interfaces the configuration supports. The minimum number is 1.

bConfigurationValue: Identifies the configurations for configuration requests. Should be more than 0.

iConfiguration: Index to a string that describes the configuration. This field is optional.

bmAttributes: Bit 6 = 1 if a device is self-powered. Bit 5 is 1 if the device supports remote wakeup feature. Bits 0 through 4 should be 0 and bit 7 must be 1.

MaxPower: Specifies how much power a device requires from the USB.

III. Interface Descriptor

Table 4.3. Interface Descriptor.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes.
1	bDescriptorType	1	The constant Interface (04h)
2	bInterfaceNumber	1	Number identifying this interface.
3	bAlternateSetting	1	Value used to select alternate setting.
4	bNumEndpoints	1	Number of endpoints supported, except 0.
5	bInterfaceClass	1	Class code
6	bInterfaceSubClass	1	Subclass code
7	bInterfaceProtocol	1	Protocol code
8	bInterface	1	Index of string descriptor for the interface.

bLength: The number of bytes in tl iptor.

bDescriptorType: The constant int 04h).

bInterfaceNumber: This field identifies the interface. This value should be unique for every interface. An interface controls and specifies device resources for every feature.

bAlternateSetting: When a configuration supports multiple, mutually exclusive interfaces, each interface must have a descriptor with the same value in bInterfaceNumber but a unique value in bAlternateSetting. Default value is 0.

bNumEndpoints: This is the number of endpoints supported by each interface.

bInterfaceClass: This field is similar to the field DeviceClass in the device descriptor.

bInterfaceSubClass: This field is similar to the field bDeviceSubClass in the device descriptor. The value is a code defined by the USB specification, if the device conforms to a particular subclass of devices.

bInterfaceProtocol: This is similar to bDeviceProtocol. Its value should be either user defined or should be pre-defined by the USB specification.

iInterface: This is an index to a string that describes the interface.

IV. Endpoint Descriptor

Table 4.4. Endpoint Descriptor.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes.
1	bDescriptorType	1	The constant Endpoint (05h)
2	bEndpointAddress	1	Endpoint number and direction.
3	bmAttributes	1	Transfer type supported
4	wMaxPacketSize	2	Maximum packet size supported.
6	bInterval	1	Maximum latency/polling interval/NAK rate.

bLength: The number of bytes in the descriptor.

bDescriptorType: The constant INTERFACE (04h).

bEndpointAddress: This includes the endpoint number and direction. Bits 0 through 3 are the endpoint number. Low-speed devices can have a maximum of 3 endpoints (numbered 0 through 2), full and high-speed devices can have 10 (0 through 15)

endpoints. Bit 7 is the direction: OUT = 0, IN = 1. Bits 4, 5, and 6 are unused and must be 0.

bmAttributes: Bits 1 and 0 specify the type of transfer the endpoint supports. 00 = Control, 01 = Isochronous, 10 = Bulk and 11 = Interrupt. For endpoint 0 control transfer type is assumed. Bits 6 and 7 must be 0. Bits 3 and 2 specify synchronization type, 00 = no synchronization, 01 = asynchronous, 10 = adaptive and 11 = synchronous. In most cases bits 3 and 2 are 00. Bits 5 and 4 indicate usage type: 00 = data endpoint, 01 = feedback endpoint, 10 = implicit feedback data endpoint, 11 = reserved. For non-isochronous endpoints bits 2 through 5 should be 0.

wMaxPacketSize: The maximum number of data bytes the endpoint can transfer in a transaction. Bits 10 through 0 are the maximum packet size from 0 to 1024. All other bits are set to 0.

bInterval: For full-speed bulk transfers this value is ignored. It is usually used for interrupt and control endpoints.

The above-described descriptors are necessary and sufficient for a simple USB device. The information in these descriptors tells the host everything there is to know about how a device functions and how it should communicate with it. The descriptor field values used in this project are shown in appendix B. A detailed description of choice of transfer type based on the application specification is also provided. In addition to this, an analysis of bandwidth usage is also done while keeping in mind the latency requirements of the application device.

CHAPTER 5

THE USB STIMULATOR DEVICE

The USB stimulator device was designed to investigate the feasibility of using the USB as a communication link between the PC and the block-chip. The system was designed and tested successfully first in an FPGA environment. A detailed description of the hardware prototype is given in chapter 6. This chapter provides an overview of the logical and architectural design and implementation of the stimulator system.

The main component of the system is the USB core. This core is an open source core and is freely available for use in research or commercial projects. It has been successfully used in different research and commercial projects all over the world.

Most USB cores available require a direct connection with a microprocessor or a microcontroller. This is done so that the descriptor database can be stored in ROM and changed as needed. Embedded software is responsible for accepting and responding to descriptor requests from the host. Using such a core increases the cost of the whole project due to the microprocessor, in terms of economic and labor cost.

Some cores include state machines that automatically perform the function of the microprocessor. These state machines recognize, accept and reply to descriptor requests from the host. Such a core was used for this system. As described in section 5.1, the USB core reproduces data transferred by the host into a parallel interface of 8-bit packets. This

parallel interface can then be connected to a microprocessor or custom logic to perform the required function. A detailed architectural description of the USB core and the two block chip interfaces is done in the following sections.

5.1 THE USB CORE

The USB core can be divided into different functional blocks. A block diagram of the blocks and their interconnections is shown in figure 5.1.

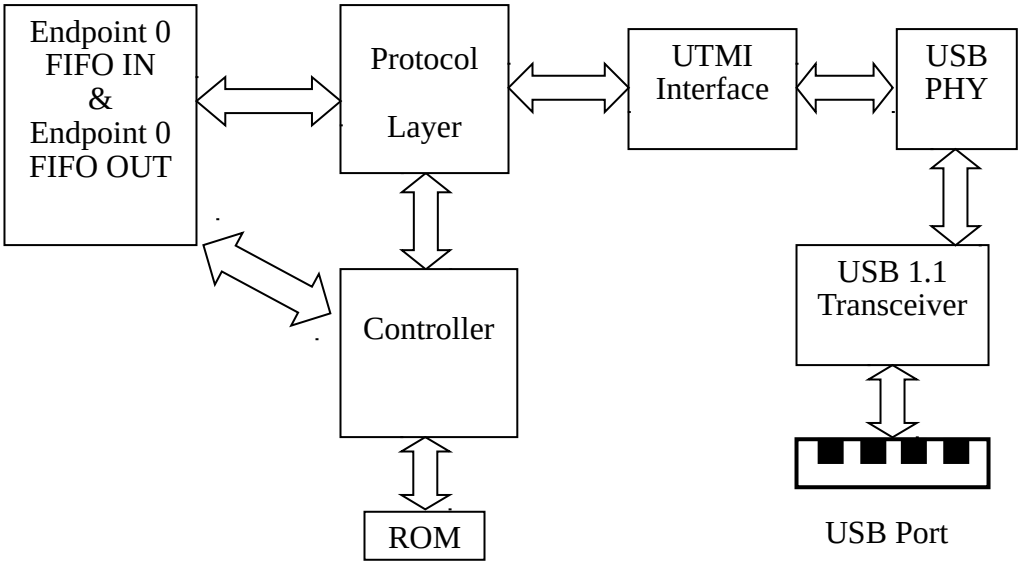


Figure 5.1. The USB Core Functional Blocks.

The USB 1.1 transceiver used for the system was a commercially manufactured IC chip. The USB port used was a standard type A connector. All the other blocks have been implemented in Verilog HDL. Some of the above blocks are top-level blocks, i.e., they consist of more than one lower level Verilog modules. This section also describes the overall functioning of the USB core from a data-flow perspective.

During enumeration, the host communicates with the USB device and receives the stored descriptors. Prior to compilation of the Verilog code for the core, the ROM module is initialized with descriptor attributes as described in chapter 4. The actual attribute

values that were used are stated and explained later in this section. When the device receives a descriptor request from the host, the protocol layer verifies protocol compliance. Once the data has been decoded, it is forwarded to the controller module. This module responds to the descriptor request, by reading the descriptors from the ROM. Once all the descriptors have been read and the operating system has assigned a driver, the device is ready to receive data from the host. When the host sends the device a data packet, the protocol layer decodes the data packet and also decodes the destination information. If the data packet is meant for endpoint 1, the data is forwarded to the endpoint 1 FIFO. Once the packet is stored in the endpoint FIFO, it can then be read by the function. The signals that facilitate the writing of data packets to the FIFO are the 8 data lines, 1 write enable line and 1 FIFO full line. Data is received in 1-byte words. The timing diagram of a data transfer is shown in figure 5.2.

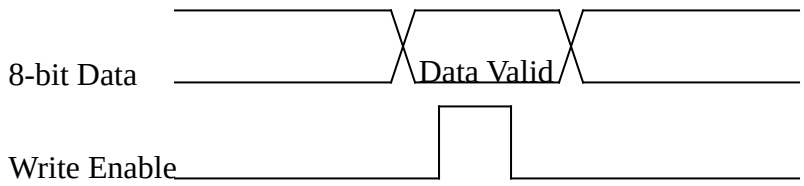


Figure 5.2. Endpoint Data Transfer Timing Diagram.

For a data packet with 8-bytes of data, the USB core outputs 8-bits of data eight times to completely store the 8-byte data packet into the FIFO. The timing diagram for an 8-byte packet is shown in figure 5.3.

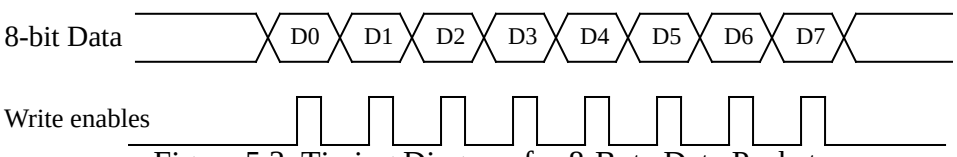


Figure 5.3. Timing Diagram for 8-Byte Data Packet.

Figure 5.4 shows the actual waveform that is seen in a logic analyzer. The signal lines shown are for Data, endpoint 1 read enable, clock, endpoint 1 write enable, empty and full signals.

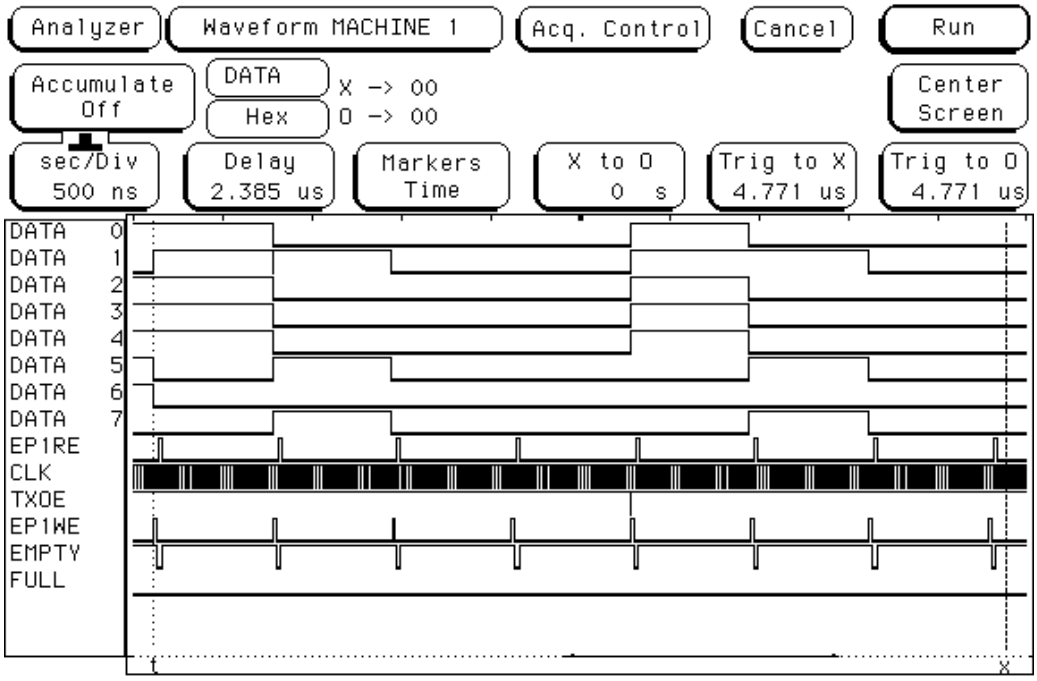


Figure 5.4. Logic Analyzer Waveform.

The system was initially implemented and tested on a customized FPGA development board. The electrical specifications and diagrams are discussed in chapter 6.

5.2 USB-BLOCK CHIP INTERFACE

This interface was designed to be compatible with the timing diagram discussed in section 3.1, and shown in figure 3.3. The interface has four lines: DCLK, DIN, TCLK and LE, as described by the block chip specification in section 3.1. A partial listing of the Verilog HDL code for this interface is given in appendix A.

The interface was designed to control a single block chip. Using a single block chip simplified the implementation and testing of the initial version. The entire interface was downloaded to an FPGA with the four block chip compatible lines. A block diagram of the overall system is shown in figure 6.1.

The system used 2 FPGA’s for its implementation. One FPGA was configured with the USB core. The second FPGA was configured as the USB-Block chip interface. This system consisted of only one endpoint. This endpoint was in the form of an 8-byte FIFO. The timing and functional specification of this endpoint 1 FIFO is the same as the endpoint 0 FIFO described in section 5.1. As shown in figure 5.4, there are 8 writes to a FIFO in a data packet. After each write, the interface controller is designed to read in the written byte. Due to a design limitation in the USB core, the controller was designed not to let the FIFO become full. If the FIFO becomes full, the USB core goes into an infinite loop, i.e., it writes the same 8-byte data to the FIFO. This sequence of writes and reads can be seen in figure 5.4.

The data format for the packet is set based on whether the system supports little-endian or big-endian. Little-endian" means that the low-order byte of the data packet is stored in memory at the lowest address, and the high-order byte at the highest address. Big-endian" means that the high-order byte of the number is stored in memory at the lowest address, and the low-order byte at the highest address. The USB system as implemented is a little-endian system. The data, as sent by the PC to the USB core is shown in figure 5.5.

Word 1 (2-bytes)					0000				0000				0000			
	AD2	**AD1**	**AD0**	**A6**	**A5**	**A4**	**A3**	**A2**	**A1**	**A0**	**T3**	**T2**	**T1**	**T0**	**P**	**H**
Word 2 (2-bytes)	0	0	0	1	1	1	1	1	1	0	1	0	0	0	1	0

Data sent to the USB device: $(0000\ 0000\ 0000\ 0000\ 1010\ 0010\ 0001\ 1111)_2 = (41{,}503)_{10}$

0 0 0 0 A 2 1 F

Figure 5.5. Interface Data Format.

In figure 5.5, the lower byte 1F is output first on the USB, and subsequently A2 and then 00 and 00. Once the interface controller receives these bytes, the first 2 bytes are loaded into a shift-register. Once the 16-bits of block chip data is loaded, it is shifted out. Although the block chip requires only 2-bytes, the FIFO size for the USB core is set to 8-bytes. There are two reasons for doing so. The first reason is to provide for enough address space for more than 1 block chips. The stimulator system can be designed to incorporate 16 or 32 block chips. An extra 16-bit space can be used as LE activation fields for the block chips. For instance, if for a particular stimulation, block chips 1, 4, and 7 are required to be stimulated with the same parameters. The first two bytes will have the block chip data, the next 16 bits will be address mapped from LE0 to LE15. Thus for block chip 1, 4 and 7, the LE1, LE4 and LE7 bits will be asserted. The controller then would output an LE pulse on each of the above lines at the same time. This feature, in addition with the hold-off bit will enable simultaneous stimulation of any electrode with the same stimulation parameters. The other advantage of having an 8-byte FIFO is discussed in section 5.3.

5.3 USB-NEUROTALK INTERFACE

The NeuroTalk family of integrated chips is a new family of chips being designed at the Illinois Institute of Technology keeping in consideration the specific needs of the neuroscience community. The neuroscience researcher has traditionally used commercially available integrated circuits (IC). Since these IC's have a variety of different applications, they are manufactured in bulk, thus providing a low cost solution for neuroscience and other applications, both commercial and research.

Neuroscience is currently proceeding towards the study of a population of neurons, rather than single neurons. Arrays of electrodes are being used to stimulate and record neural signals. It is well recognized that significant advances in the field of Neuroprosthesis will come about with the understanding of how to use large number of electrode arrays as a two-way informational link with the brain, or with neurons anywhere else. Due to the large number of interface channels required, the circuit requirements for neuroscience devices are growing with respect to commercially available components. Due to these reasons many researchers, either have to adapt, or to make a compromise in order to use commercially available components. To solve this problem, the NeuroTalk interface was designed and the first block chip that supports such an interface was fabricated. For this research, a USB-NeuroTalk interface was designed that converts USB data received, into NeuroTalk compatible instruction stream. This instruction stream is then sent to a NeuroTalk compatible block chip for processing and execution. A description of the NeuroTalk interface is described and a timing diagram is shown in figure 5.6.

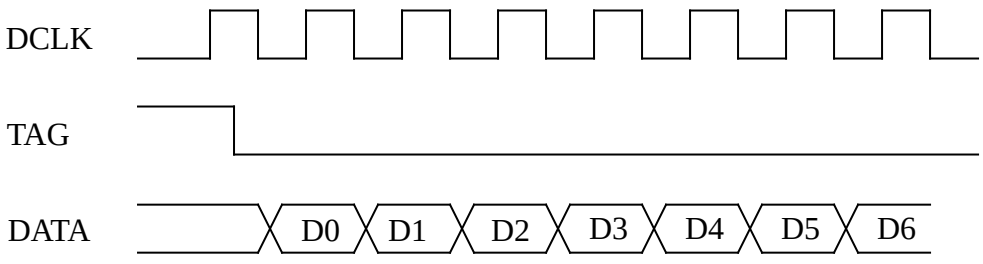


Figure 5.6. NeuroTalk Bus Timing.

The NeuroTalk interface consists of three signals, Dclk, Tag and the Data signal. The Dclk is the data clock using which the data is shifted into the block chip. The Tag signal is used as a reset signal. In the above timing diagram, we can see that the tag signal is initially high. The next rising edge of Dclk, after the Tag goes low, clocks in the first data bit on the 1-bit Data signal. All subsequent Data bits are read in by the block chip, as they are transmitted from the USB-NeuroTalk interface. When the Tag signal goes high, the shift register in the block chip is reset. Hence, Tag can be used to cancel the execution of a particular instruction stream, once it has been loaded into the shift register.

The NeuroTalk specification calls for variable length instruction commands. Doing so provides for multiple instructions and maximizes functionality of the block chip. The older version of the block chip accepted only fixed-length 16-bit long instructions. Each instruction is addressed to a single electrode channel. Having more than one instruction can add different and advantageous functionality to the block chip. For instance, in addition to the stimulation instruction of the old block chip, one instruction can be designed to send the same waveform parameters to more than one channel. If we consider the example of visual prosthesis, another kind of block chip architecture can be considered. Biological vision is very similar to computer vision; both cases include frames of external objects. For artificial vision, a similar two-dimensional frame buffer can be implemented in hardware. This buffer can be designed to hold a certain number of instructions. Different instructions can then be implemented to update and flush the buffer, start or stop the stimulation. A number of variations can be implemented for each of the mentioned instructions. Due to the variations and different kinds of instructions, a large storage space is needed in the endpoint of the USB core. Using an 8-byte FIFO can

adequately provide for the instruction storage for the first few versions of the NeuroTalk block chips.

5.4 USB BANDWIDTH ANALYSIS

To accurately determine the speed with which our stimulator devices can accept data, a bandwidth analysis needs to be performed. Even though the USB core runs at 48MHz, it does not provide data at that same rate. To guard against over- and under-flow of data, appropriate measures must be taken. For this very reason a FIFO is used. This section discusses the full-speed bulk transaction limits that every bulk connection adheres to. Table 5.1 below shows the table 5-9 shown in the USB 2.0 specification document in section 5.8.4. Note: Each frame in full-speed mode is 1ms long.

Table 5.1. Full-speed Bulk Transaction Limits.

Data Payload	Max Bandwidth (Bytes/second)	Frame Bandwidth Per Transfer	Max Transfers	Bytes Remaining	Bytes/frame Useful Data
1	107,000	1%	107	2	107
2	200,000	1%	100	0	200
4	352,000	1%	88	4	352
8	568,000	1%	71	9	568
16	816,000	2%	51	21	816
32	1,056,000	3%	33	15	1,056
64	1,216,000	5%	19	37	1,216
Max	1,500,000				1,500

For a payload of 8-bytes, the maximum transfers allowed are 71. Thus the total number of bytes that are transferred per frame are 71 x 8 = 568-bytes. 568,000 bytes are transferred every second (568 x 1,000 = 568,000). If only 2 out of the 8-bytes are used, about 71 x 2 = 142 useful bytes are transferred per frame (1ms), and 142,000 useful bytes

are transferred per second. On a per-second timeline, only about (142/568 x 100 = 25%) 25% of the bandwidth is used.

For this research, a FIFO of 8-bytes was used, as described above. The extra 6-bytes are used as an extra buffer for the USB-NeuroTalk interface which has a largest instruction size of 26-bits. The extra 6-bytes can also be useful for instruction set upgrades in future versions of the NeuroTalk interface.

CHAPTER 6

FPGA PROTOTYPE DESIGN

The prototype was initially verified and validated on a custom designed FPGA board. The FPGA board was designed to accommodate two ALTERA FLEX 10K FPGA chips. One FPGA was configured with the USB core, after being configured with the required number and sizes of the endpoints. The ROM, which was also a part of the USB core, was initialized with the appropriate descriptor values. A USB 1.1 transceiver chip from Fairchild Semiconductors was used as the bus front-end. It ensured electrical compatibility with the USB standard. The FPGA's were programmed using ALTERA's Quartus II software. Programming circuits were designed for both FPGA's on the prototype boards to facilitate re-programming at any stage in development. The second FPGA was used to program either the USB-Block chip interface or the USB-NeuroTalk interface. This FPGA has two sets of I/O's; one set is responsible for accepting data packets from the USB core FPGA. The second set of signals is either block chip, or NeuroTalk interface compatible.

FPGA programming circuits were designed and added to the prototype board to program the FPGA's. FPGA's can be programmed in two different ways. One way is to

use a programming cable for each re-programming; the second option is to store the programming information in an EEPROM. The option of using EEPROM to store programming information was not used since developing the prototype required constantly modifying the design and re-programming the FPGA's. Hence during initial development, to simplify the hardware, the cable programming method was employed. A block diagram of the prototype board is shown in figure 6.1.

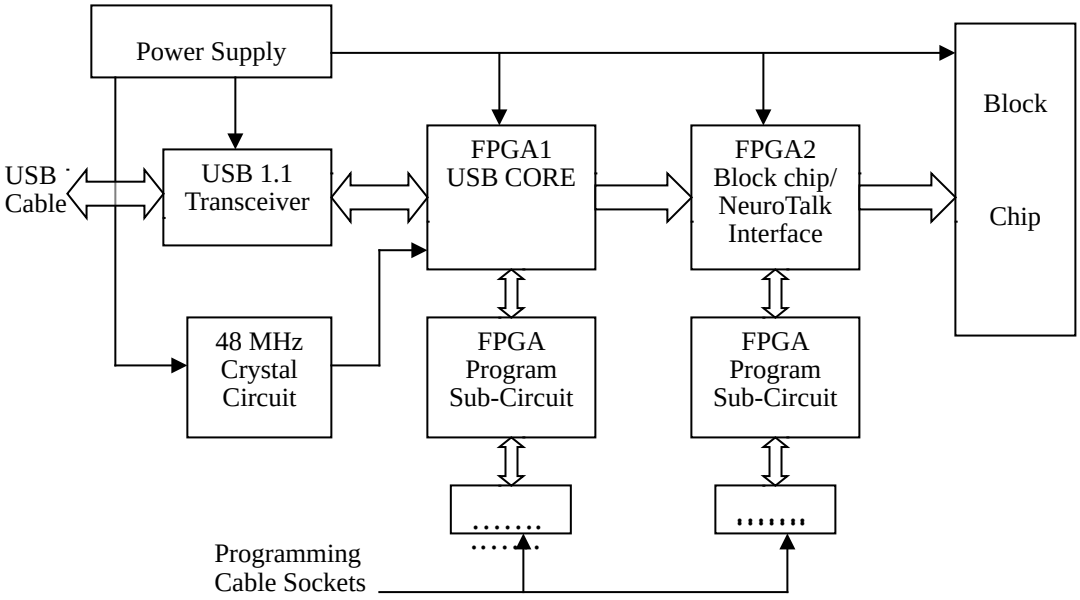


Figure 6.1. Prototype Board Block Diagram.

The figure in 6.1 provides an overview of the prototype system developed. A brief description of each block is provided below.

Power Supply: The power supply was designed to supply power to the USB transceiver; the two FPGA's and programming sub-circuits, the crystal circuit, and the Block chip. The transceiver and the FPGA programming circuits required a power supply of 3.3

volts. This was achieved using the LM317 adjustable voltage regulator. The circuit diagram and component values are shown in figure 6.2.

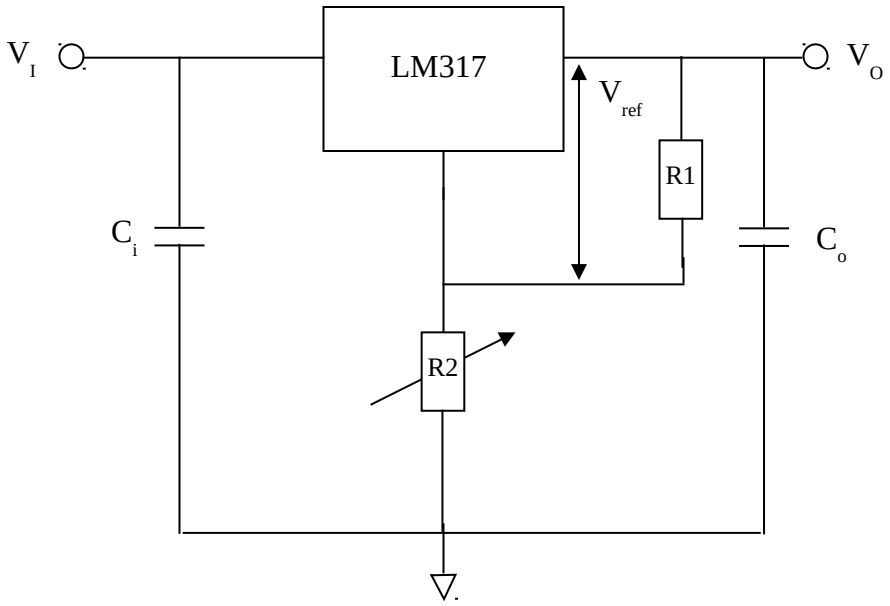


Figure 6.2. Adjustable Voltage Regulator.

The equation used to set the output voltage by varying R2 is shown below. Resistor R1 is set to 240 Ω . The equation for V_o is (V_{ref} is 1.25 Volts):

$$V_o = V_{ref}(1 + \frac{R_2}{R_1})$$

For a V_o of 3.3 volts, R2 = 393.6 Ohms or approximately 400 Ohms. The FPGA's, along with an I/O voltage supply of 3.3 V, require a core voltage of 2.5 Volts. To achieve

2.5 Volts the value of R2 is required to be 240 Ohms. The block chip requires two different voltage levels. It requires 5 Volts for V_{cc} and for V_{INDIF}, and 10 Volts for the high voltage supply $V_{HV.}$ The V_{HV} was derived using a 10 Volt zener diode and the two 5 Volt supplies were powered by one 7805, 5 Volt regulated power supply.

The crystal oscillator was powered using another 7805-voltage regulator to supply the required 5 Volts.

48 MHz Crystal Circuit: The clock generation circuit was designed using a 48 MHz crystal, a 5K-Ohm resistor and a 10pF and 15pF capacitor. The clock was generated using a HEX inverter IC. The circuit is shown in figure 6.3.

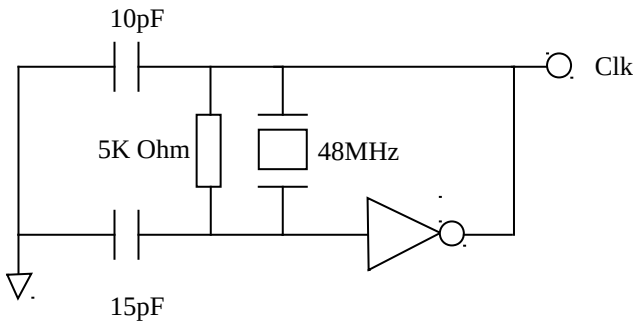


Figure 6.3. Clock Generation Circuit.

The circuit designed above, generated a clock of 48MHz that was required by the USB core to sample and decode the USB signals coming from the PC.

FPGA1 (USB Core): The FPGA's have a total of 144 pins, including power, configuration and I/O pins. The six front-end pins of the USB core were connected to the transceiver. These pins conformed to the standard USB transceiver specification. Eight

data signals, one write enable (Wen) signal and one FIFO full signal were the outputs to the interface FPGA2. Figure 6.4 shows the various connections.

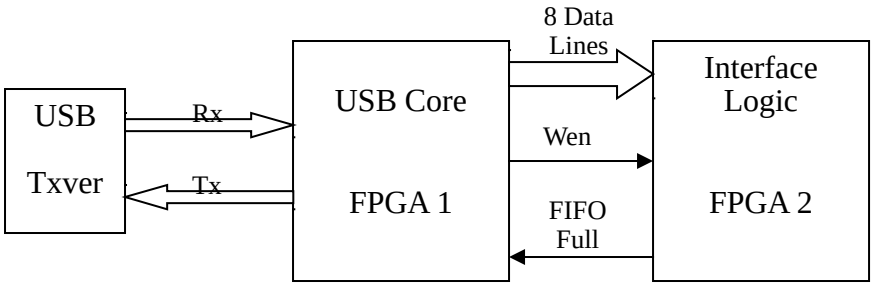


Figure 6.4. USB Core Input/Output Connections.

FPGA2 (Block chip/NeuroTalk Interface): The Interface FPGA, as shown in figure 6.4 had 10 I/O lines on its front-end and 4 I/O lines on its back-end (Block chip side) as shown in figure 6.5.

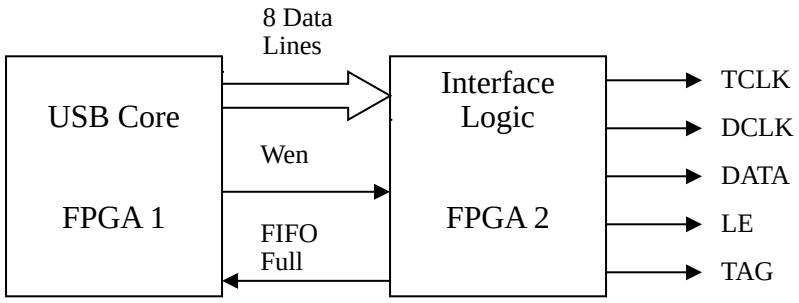


Figure 6.5. Interface Core Input/Output Connections.

FPGA Program Circuit: The FPGA's were programmed using the ByteBlaster parallel port download cable as described by the Altera programming data sheet. Table 6.1 shows the ByteBlaster 25-pin header pin-outs.

Table 6.1. ByteBlaster 25-Pin Header Pin-Outs.

Pin	Signal Name
2	DCLK
3	NCONFIG
8	DATA0
11	CONF_DONE
13	NSTATUS
15	GND
18-25	GND

The 25-pin header plugs into the LPT/Parallel port of the PC. The other end of the cable has a 10-pin female header, which has pin connections shown in table 6.2. The programming circuit must provide VCC and GND to the cable at the appropriate pins shown in table 6.2.

Table 6.2. ByteBlaster Female 10-Pin Header Pin-Outs.

Pin	Signal Name	Description
1	DCLK	Clock Signal
2	GND	Signal Ground
3	CONFIG_DONE	Configuration
4	VCC	Power Supply
5	nCONFIG	Configuration
6	NC	No Connect
7	nSTATUS	Configuration Status
8	NC	No Connect
9	DATA0	Data to Device
10	GND	Signal Ground

The ByteBlaster schematic diagram is shown in figure 6.6. It shows an octal driver IC that is required to configure the FPGA’s. The 10-pin female header plugs into the circuit board and the 25-pin end plugs into the PC and is sent configuration data by the Quartus II software.

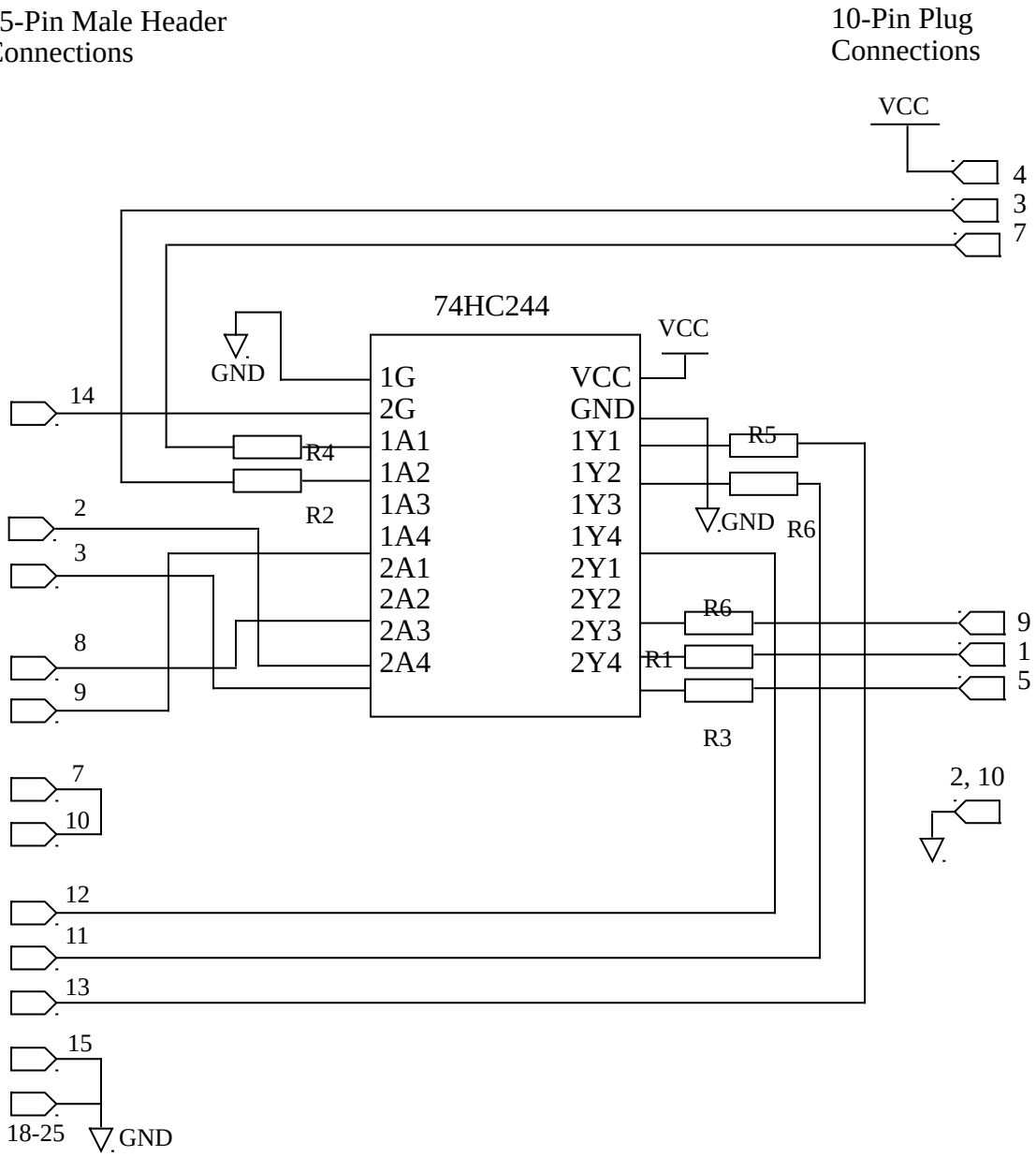
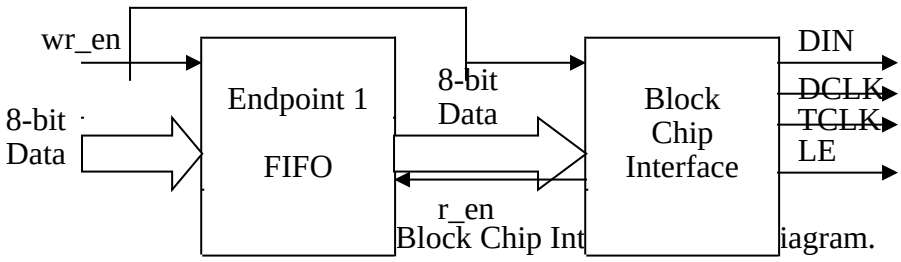


Figure 6.6. ByteBlaster Download Cable Schematic.

6.1 USB-BLOCK CHIP INTERFACE

The USB-Block chip interface was implemented using Verilog HDL. The interface has all the signals specified in the block chip specification document. The functioning of

the block chip interface depends on the DCLK. The interface shifts out the block chip instruction to the block chip for execution. This requires the interface to output each bit out of the 2-byte instruction, one at a time. Completely transferring the 16-bits requires 16 DCLK cycles as the DCLK is used by the block chip to read in the data in its shift register. After successfully shifting out the data, the interface outputs a 200ns long LE pulse, 200ns after the shifting out of the last bit. Hence the total time taken for one complete transfer to take place is 16 * (1 DCLK cycle time) + 200ns + 200ns. The DCLK frequency used for the device is about 2.4MHz. Each cycle will thus be 416.6ns long. So, one complete instruction transfer will take place in 7.066 μs. Using this time information, a verilog module that counts each DCLK cycle was designed. Depending on what the state of the interface is, the interface will decide on when an LE pulse needs to be output. Once the transfer is complete, the time counter resets itself and waits for the next instruction to be transferred.



Block Chip Interface Diagram.

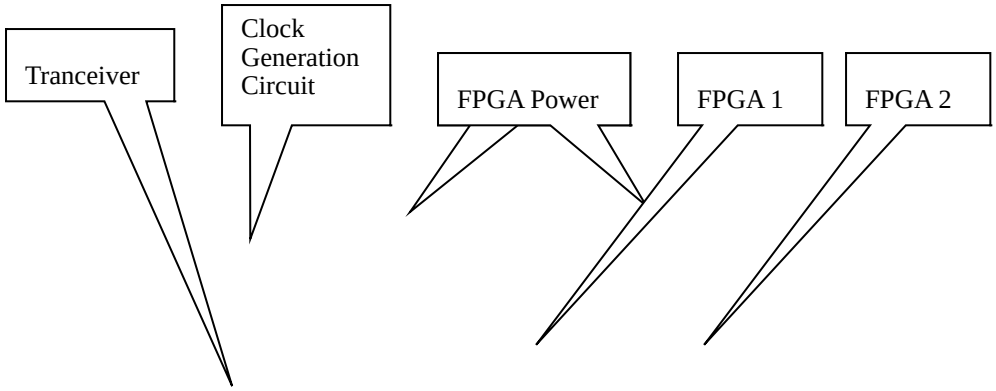
Figure 6.7 shows the block diagram for the block chip interface logic. The write enable signal is used as an input to both the FIFO and the interface block. As each byte of data comes in, it is used by the interface and prepared to be sent to the block chip serially.

6.2 USB-NEUROTALK INTERFACE

After the successful testing of the USB-Block chip interface, the neurotalk interface was designed to interface with the USB. Although it had certain similarities with the block chip interface, it took considerably less time to design due to the fact that the interface to the USB core had already been studied extensively while designing the USB-block chip interface. While the USB-block chip interface took about 16 weeks to design and implement, it took only about 1 week to design, implement, test and debug the USB-Neurotalk interface. Out of the 16 weeks for the block chip interface, it took 4 weeks to design the FPGA development board, and 12 weeks to learn the inner functioning of the USB core.

6.3 FPGA DEVELOPMENT BOARD

The FPGA development board was custom designed without the use of a printed circuit board. The general layout of the board is shown in figure 6.1. A photograph of the board is shown in figure 6.11. The board used two TQFP sockets for the two FPGA's and a QFN socket for the USB 1.1 transceiver. Each of the two FPGA's required two different power supplies of 2.5 volts and 3.3 volts. The board also has a block chip DIP-40 socket.



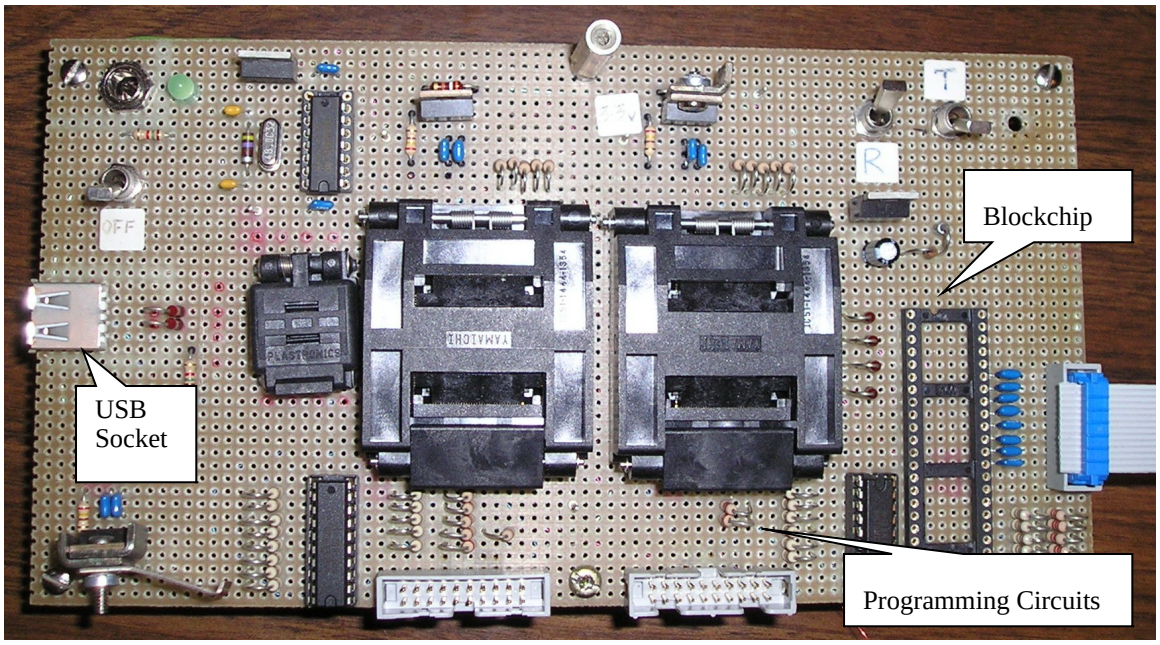


Figure 6.8. Photograph of the FPGA Development Board.

6.4 DEVICE DRIVER AND APPLICATION

The Microsoft device driver development kit has sample device drivers available for use. One of these drivers was a bulk endpoint compatible driver and was used for this research. Along with the driver, a sample application was available that was modified to send block chip compatible data to the device.

CHAPTER 7

USB-INTERFACE SYSTEM TESTING

The complete system was tested in three stages. The first stage involved testing the FPGA development board since it had five different sub-circuits. The power, two FPGA programming, clock generation and block chip circuits were tested and their nominal performance verified. The second stage involved testing the USB core and verifying that it actually functioned as stated by the designer. The third stage involved testing of the complete system including both the interfaces. Each of the three-stage testing methodologies is described in the following sections and test results of each are shown.

7.1 FPGA BOARD DESIGN AND TESTING

The FPGA board was custom designed without the use of manufactured PCB's. This method was used for the initial prototype due to the fact that it is very difficult to modify PCB traces in case if even a simple modification is required. Even though the modification is simple, implementing a complex system is tedious work due to the fact that the FPGA socket and the transceiver socket pins were small. Hence magnet wire was used to make the necessary logic line connections under a microscope. Making the necessary connections and verifying them required that the connections be viewed under the microscope. For the power circuits, a higher gauge wire was used. Once all the power, clock and programming circuits were built, an FPGA was inserted in one of the sockets and tested for programmability. Initially program configuration failed. It was later discovered that the power pins for each of the four banks of the FPGA need to be powered for the FPGA to program and function correctly, even if one or more sides are not used. Once the FPGA was fully powered, the FPGA configured successfully. Once the general circuit connections for one FPGA were verified and validated, the circuit

connections were replicated for the second FPGA. A photograph of the circuit connections for the FPGA board is shown in figure 7.1.

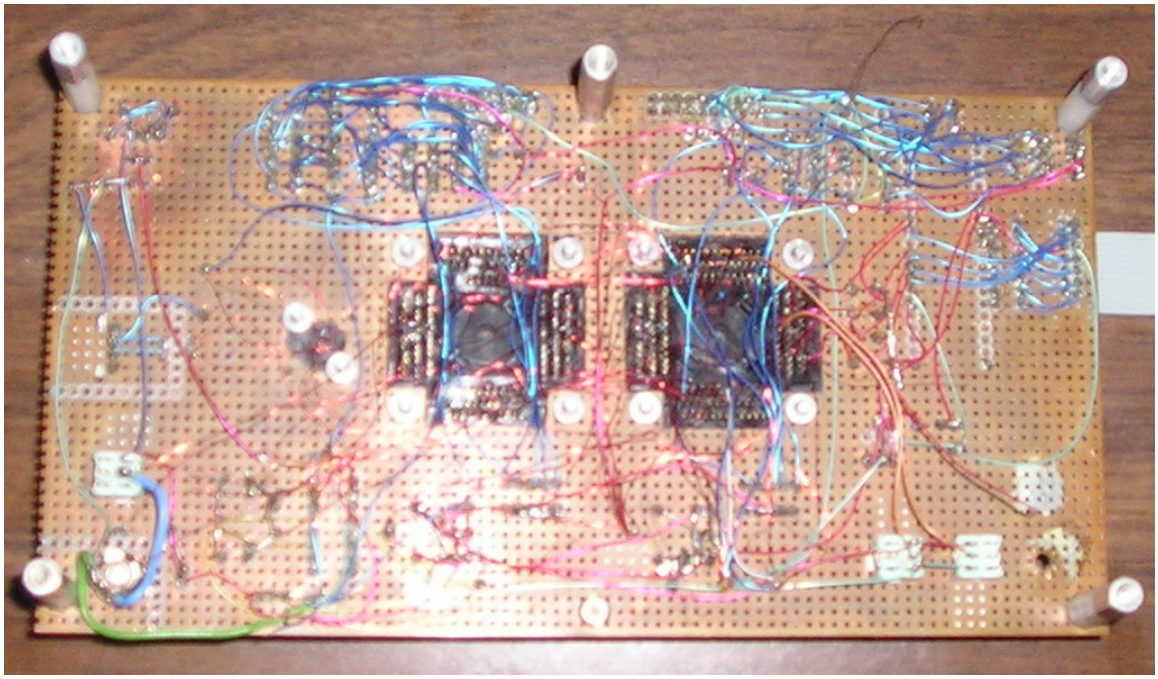
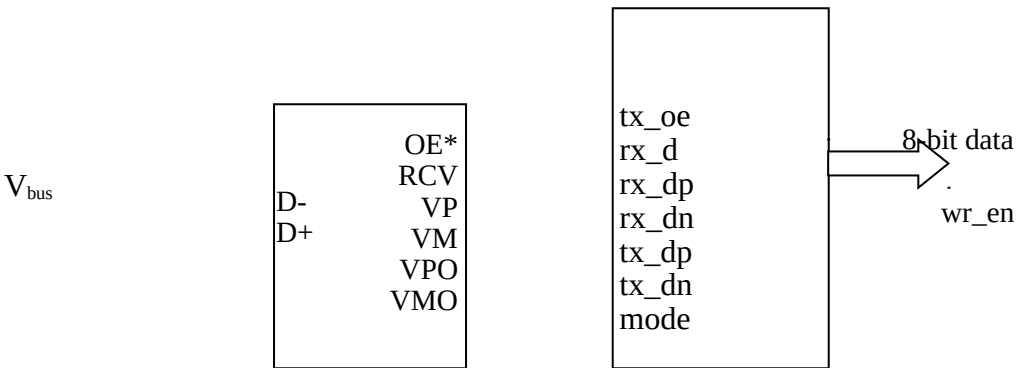


Figure 7.1. Photograph of underside of FPGA Prototype Board.

7.2 USB CORE TESTING AND VERIFICATION

The USB core chosen was obtained from www.opencores.org, and since it was obtained in open source form, it lacked detailed documentation. The top-level verilog module consists of endpoint I/O's, pins for the USB 1.1 transceiver and other status signals. The transceiver and USB core interface circuit is shown in figure 7.2.



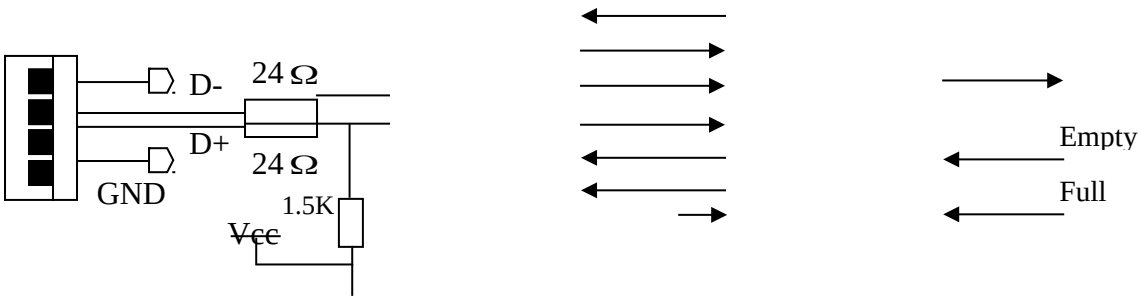


Figure 7.2. Circuit Diagram of Transceiver and USB Interface.

According to the overall design of the USB core, the 8-bit data bus shown in figure 7.2, the write enable, wr_en signal, the empty and full signals play a part in the data transmission. Once the USB receives a data packet from the PC, it is decoded according to the USB specifications and forwarded to the OUT endpoint FIFO through the 8-bit data bus. In the USB system, initially, there were two unknowns. It was not known which driver could be used for the device. The other unknown was whether or not the USB core functioned properly as it was implemented. To increase the chances of the device functioning, one unknown had to be removed. The obvious choice was the driver/application to be used. A few USB debug applications were installed to test the device. It was assumed that these devices had the requisite generic drivers that would enable the PC to successfully communicate with the USB core, provided the core was implemented correctly.

Once a data packet successfully arrives at the core, it is then placed on the data bus 8-bits at a time. When each byte is valid on the bus, a pulse is sent on the write enable signal. The byte is then written into the OUT endpoint FIFO. Once the complete packet is written, or even before it is written, the data can be used by the endpoint function. One test strategy was to monitor the eight data lines and the write enable lines to ascertain the arrival of the data packet at the USB core. As stated above, while conducting the test, it was assumed that the PC was correctly sending the data to the USB device.

Initially when the device was powered up and connected to the PC, the Windows operating system did not acknowledge the addition of the device to the USB bus. Tests were run using a USB configuration snooping software, and it was discovered that the device was not completing the enumeration process, as it should to be recognized by the PC. When a device is first connected to a USB port, the PC sends a particular sequence of commands to the device, and it expects a reply to each of the three commands it sends. If for any reason this query and reply session fails, the device enumeration fails and the PC does not recognize the device. This is what the symptoms were in the case of the USB device. Hence it was hypothesized that the device is not being recognized due to some failure in the enumeration process. To trace the problem, a logic analyzer was hooked to certain test points on the FPGA that tracked the requests and replies flowing between the USB core and the PC. Since the core had two FIFO's, one each for the PC requests and the core replies, it was fairly simple to setup the test. The data acquired by the test is reproduced below in figure 7.3.

Packet 1 from PC Host

Byte 1: A5: Start of Frame
Byte 2: EC: Frame # +
Byte 3: 9C: CRC5
Byte 4: 2D: Setup
Byte 5: 00 } Addr.+Endp.+
Byte 6: 10 } CRC5
Byte 7: C3: Data PID

Byte 8: 80: bmRequestType
Byte 9: 06: bRequest
Byte 10: 00: descriptor type
Byte 11: 01: descriptor index
Byte 12: 00: wIndex high
Byte 13: 00: wIndex low
Byte 14: 40: wLength high
Byte 15: 00: wLength low
Byte 16: DD: CRC16
Byte 17: 94: CRC16

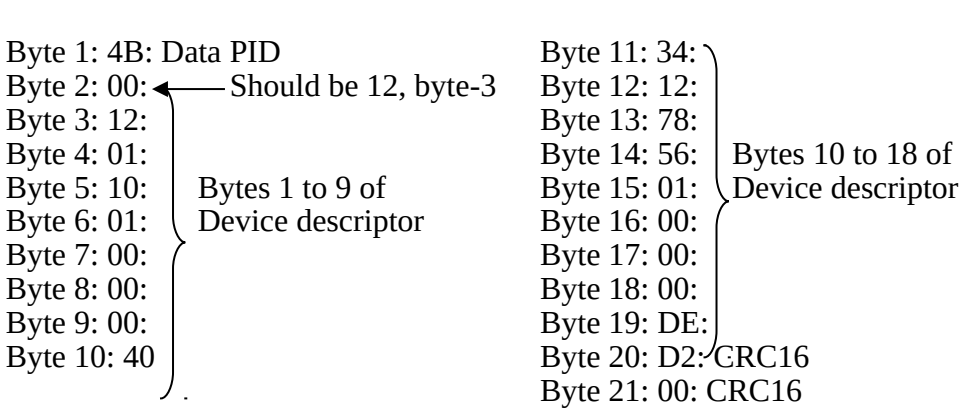


Figure 7.3. Enumeration Failure Test Results.

The test as described above was successful in locating the source of the enumeration failure. The problem was due to an extra byte that was added on the reply by the USB device. This extra byte had all bits set to 0, as can be seen in figure 7.3. Although, the problem was known, locating the source took a considerable amount of time since the inner code of the USB core needed to be looked at and studies, without any formal documentation. After about 3 weeks of intense search, the source of the problem was positively identified and eliminated.

In the initial phase of device development, Altera's MAXPLUS II software was used. In the USB core, four FIFO's were implemented. Two FIFO's were used for the control endpoint, one each for IN and OUT endpoints. Another FIFO was used for OUT endpoint 1. The forth FIFO was a smaller FIFO used as a pre-fetch FIFO for enumeration. To successfully compile the USB core in MAXPLUS, certain changes were made to the FIFO's. The endpoint FIFO's were instantiated using library SRAM modules provided by MAXPLUS. The smaller pre-fetch FIFO, however, was implemented by using the *reg* keyword available in Verilog. MAXPLUS, did not synthesize an SRAM using the *reg* keyword because this feature was not supported in that particular version. The

implementation for the smaller FIFO was then changed to make use of the library SRAM modules. After the change, the complete USB core successfully compiled without any syntax errors, and work continued on implementing the USB-block chip interface. During this initial phase, the PC on which the core development was being done had to be replaced with another. This required installing the software on the new machine. By then Altera had phased out the older MAXPLUS software and required developers to download and install an upgraded version of the software with better features and advanced options, called Quartus. Once the installation was done development went on without incident. The cause of the enumeration failure, it was discovered, was the change done in the implementation of the smaller pre-fetch FIFO. As designed, the USB core was supposed to have the FIFO functioning with unregistered outputs. In other words, as soon as the first byte is stored in the FIFO, it appears on the output pins, i.e. it did not require a positive edge of the read clock for the data to appear on the output. The default setting of the library function, however, set the FIFO to function in the registered mode, i.e., when the first byte is entered into the FIFO, it appears on the output lines only after the rising edge of the read clock. It was due to this reason, an additional 0-byte was being appended to the reply to the PC. Setting the FIFO to function with unregistered outputs then solved the problem. Apart from this there was no other problem with the USB core as it was implemented.

7.3 INTERFACE TESTING AND VERIFICATION

The only problem encountered while designing the interface what that of clock jitter. The DCLK and TCLK used by the block chip were derived from the main 48MHz clock. The main clock was observed to jitter, when viewed on a logic analyzer. This jitter was hence also induced in the DCLK and TCLK. The source of the problem on the derived clocks and a solution is presented below. The problem is shown in figure 7.4.

Figure 7.4. Clock Skew.

To better control and transmit DCLK and TCLK, a faster clock called FCLK was implemented. The DCLK was initially derived from the 48MHz clock using a counter. One cycle of the 48MHz clock is 20.83ns long. The block chip specification lists certain minimum setup time for the DIN line. To comply with the requirements the DCLK frequency was chosen to be 2.5MHz, with one cycle 0.4 μ s long. This means that there are $\frac{0.4\mu s}{20.83ns}$ = 19.2 $\approx$20 main clock cycles in each DCLK cycle. Even though there are 20 cycles in each DCLK cycle, we only need 10 cycle counts because DCLK is high or low for only 10 of these cycles, and one period consists of 20 cycles. To count 10 main clock cycles a 4-bit counter is required. Initially, DCLK is logic low and it remains low for the first 10 increments of the counter. At the next step, DCLK's state is changed to logic high and the counter is reset to 0. The counter again counts to 10 and then DCLK is set to 0. The source of the problem is shown in figure 7.5.

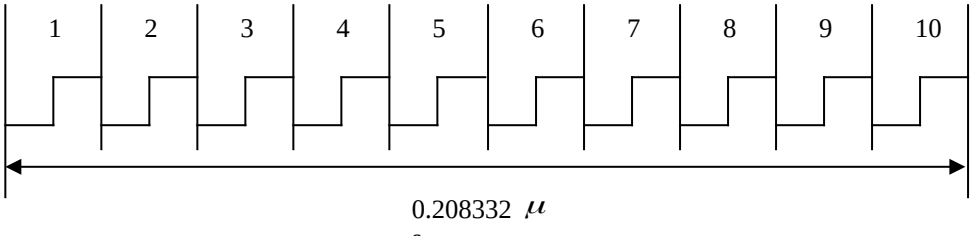


Figure 7.5. 48MHz clock with DCLK half-cycle timing.

Due to the jitter, each clock cycle, can be either more or less than 20.83 ns. We can assume for simplicity in describing the problem that because of a low resolution of measurement, each cycle of the main clock can be shorter or longer than 20.82 ns by about +/- t_s. Hence in figure above, the ten clock cycles on a worse case scenario, can be 0.208322 + (10 * t_{clk}) or 0.208332 – (10 * t_{clk}). This means that DCLK can change state from either a 0 or a 1, in the range stated above. This high difference in jitter will cause the DCLK to also jitter at a high rate. To solve the problem a 5MHz FCLK was derived from the main clock. Each cycle of FCLK will then be 0.2 μ s long. Each FCLK cycle will have $\frac{0.2\mu s}{20.83ns}$ = 9.6 ≈10 main clock cycles. To generate FCLK, a 3-bit counter is required. For the first 5 counts, FCLK will remain at logic 0; the counter will then be reset to 0. For the next 5 counts FCLK will be set at logic high. Assuming the same rate of jitter on each clock cycle, we can calculate the new range the FCLK will jitter in. According to figure 7.6 the range will be, 0.104166 + (5 * t_{clk}) to 0.104166 – (5 * t_{clk}). Since the total skew of 5 clock cycles is less than the skew of 10 clock cycles, the jitter of FCLK is less than that of the main clock cycle. DCLK is derived from FCLK and not the main clock to further reduce the jitter. Since FCLK is twice as fast as DCLK, there are two FCLK cycles per DCLK cycle.

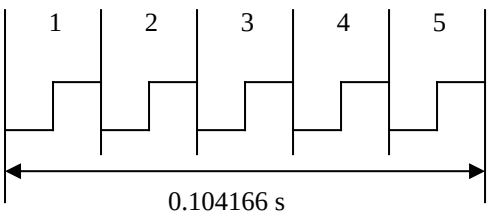


Figure 7.6. 48MHz clock with FCLK half-cycle timing.

The DCLK jitter in the first case is +/- 10* t_{clk}. Using the FCLK to derive the DCLK reduces this jitter. FCLK jitters in the range of +/- 5 * t_{clk}. Since DCLK is based on FCLK, its jitter is also reduced to +/- 5 * $t_{clk,}$ in other words, it was reduced by half. When noticed on a logic analyzer, the jitter on DCLK was not noticeable.

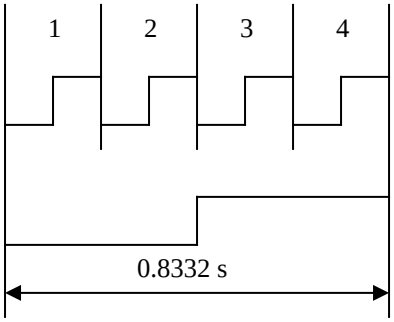


Figure 7.7. FCLK and DCLK.

CHAPTER 8

ASIC DESIGN FLOW

After the successful implementation, verification and validation of the interfaces and the USB core in FPGA, the designs were prepared for fabrication in ASIC. This chapter describes the process of preparing a proven design for layout. There are three distinct steps in the process and are described in the following sections. The first step is synthesis that converts Verilog logic code to a transistor level circuit. The second step converts the transistor level circuit to a layout while conforming to a particular technology library. The third step consists of performing design rule- and layout vs. schematic-checks. This step guarantees that the layout generated is functionally, and electrically equal to the transistor level schematic of the design.

8.1 SYSNTHESIS

This step converts the verilog source into its transistor level circuit. The software used to perform synthesis on the USB and interface cores was a widely used Synopsys design compiler. Figure 8.1 shows the steps required to convert a verilog project into its equivalent transistor level circuit.

The project information is first entered into the compile.scr file, which is a setup file for the synopsys design compiler. The information entered in the setup file includes; verilog design files, top level module, output files, etc. The dc_shell command is invoked to start the synthesis process on the design files. The screen output is written to a text file to conveniently analyze the results of the synthesis operation. The output of the synthesis operation is the creation of the .lsi file.

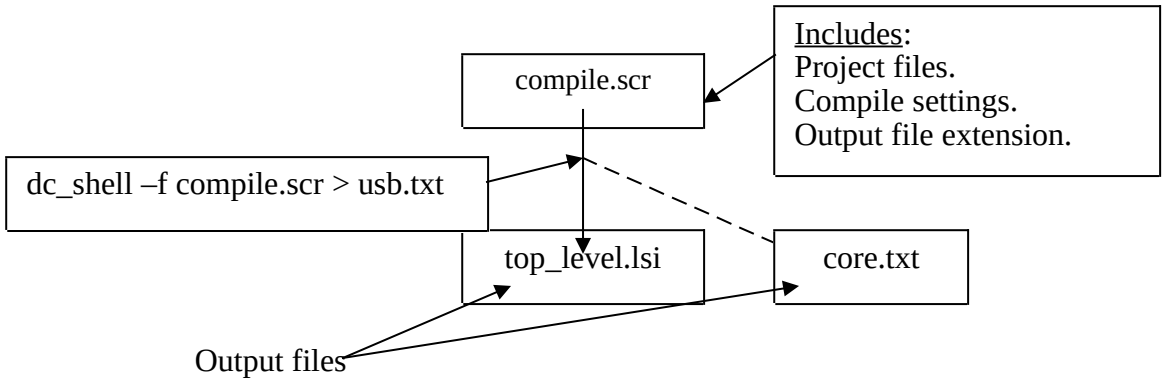


Figure 8.1. Synopsys compilation steps.

8.2 LAYOUT

The L-Edit software was used to layout the USB and interface cores. However, before L-Edit can perform layout, the source file provided to it should be in a proper format. Hence, the .lsi file obtained in the previous step is converted into the .tpr format to be readable by L-Edit. This process is depicted in figure 8.2.

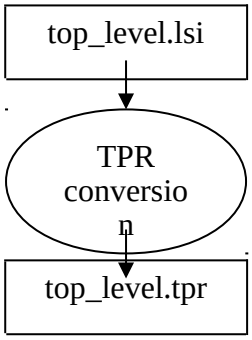


Figure 8.2. LSI to TPR conversion.

Once a tpr file is obtained, L-Edit is commanded to use that file as its input for creating the layout of the complete chip. Once the layouts of the three chips was obtained, the two interface cores were combined with a USB core, and the transceiver core to create the USB-blockchip and USB-Neurotalk chips. These two layouts are shown in figure 8.3 and 8.4.

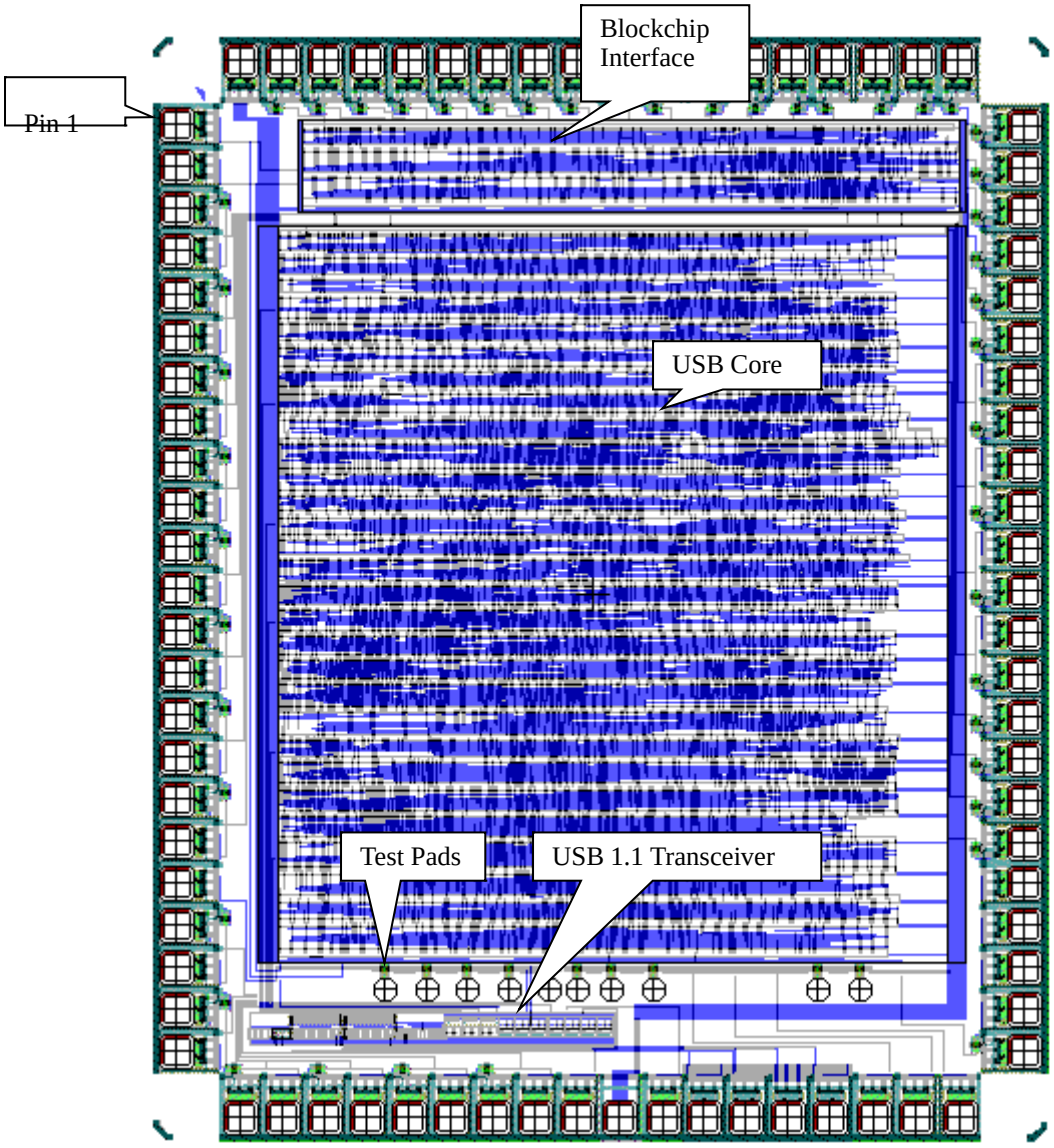


Figure 8.3. USB-Blockchip Core.

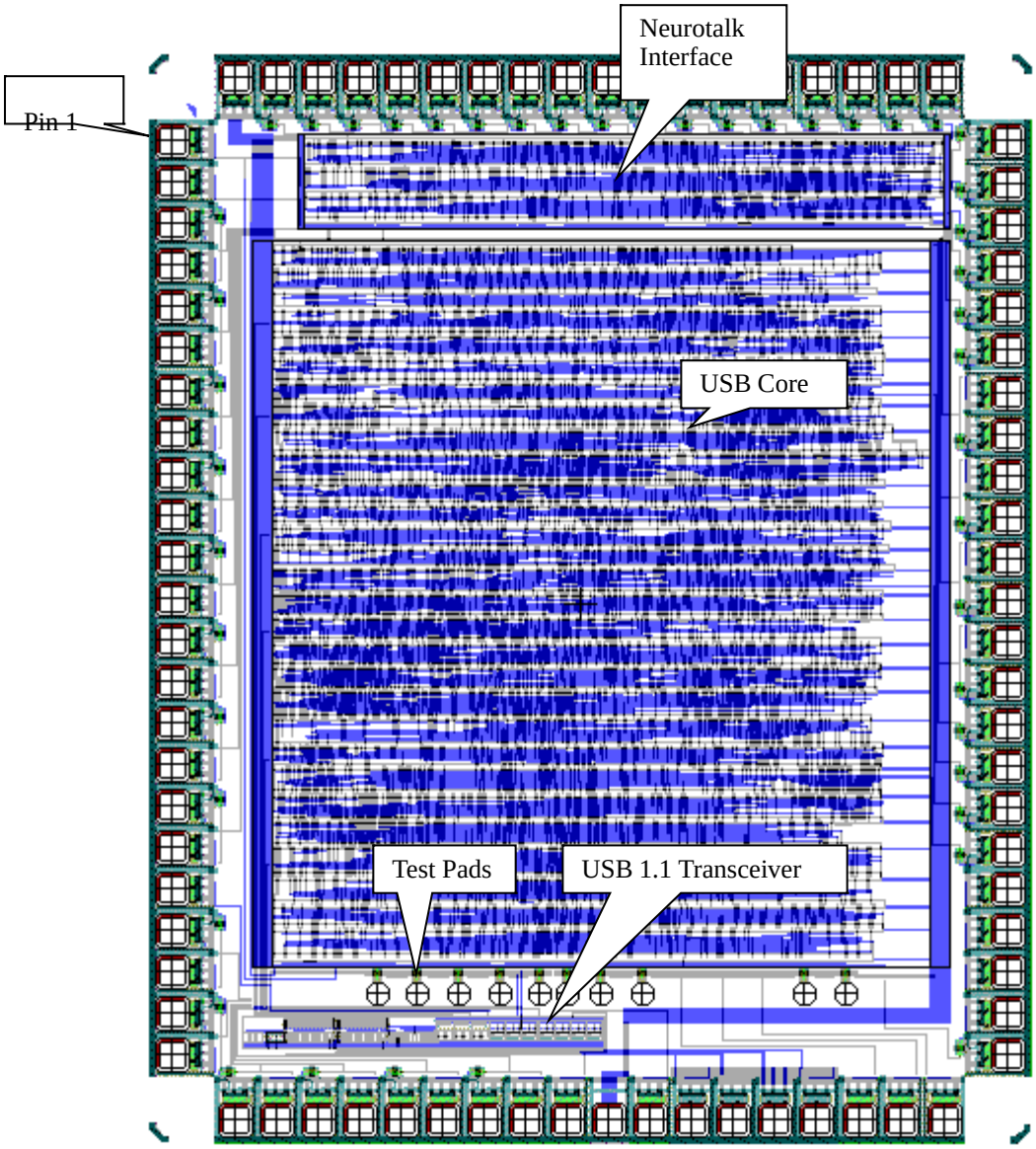
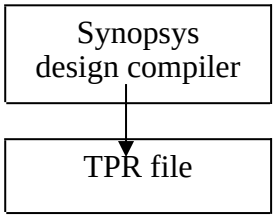


Figure 8.4. USB-Neurotalk Core.

8.3 DRC & LVS CHECKS

Once the complete cores were done, design rule and layout verses schematic checks needed to be done to ensure that the synthesized cores were translated into their respective circuit level representations without any errors. In other words, the layout process, and the process of connecting the three layouts, should not induce any errors. DRC is a function built into the L-Edit software and did not report any errors in the design rules that were used. The LVS process is a little involved and is shown in figure

8.5. L-Edit is used to extract the transistor level circuit netlist from the layout itself. This process creates an .spc file. In addition to the two complete cores, the three sub-components of each core were also used to generate separate spc files. The sub-component spc files are then used by ORCAD software to create a transistor level circuit. These circuits are then converted into blocks and connected the same way, as in the final layout, using ORCAD. Once the transistor level circuit for the USB-blockchip and the USB-neurotalk interfaces are obtained, each of these is extracted into a top-level netlist that contains not only the USB circuit, but also the interface and transceiver circuit. Once the top-level transistor level circuit netlist is obtained, it is compared to the transistor level netlist extracted from the layout, using a L-Edit software utility called LVS. If the two files match, it means that the interconnection of the three cores, in layout, i.e., USB, interface and transceiver match perfectly with the ORCAD circuit level netlist. Doing such a check increases the confidence level in the layout interconnections, as the interconnections in ORCAD are easy to do and the errors are more easily detected. Figures 8.6 and 8.7 show the ORCAD circuits for the USB-Blockchip and the USB-Neurotalk chips. These circuits were used to extract a netlist, which was compared with the layout netlist obtained by directly extracting the combined layout of the cores, shown in figure 8.3 and 8.4. Figure 8.5 shows the entire LVS process.



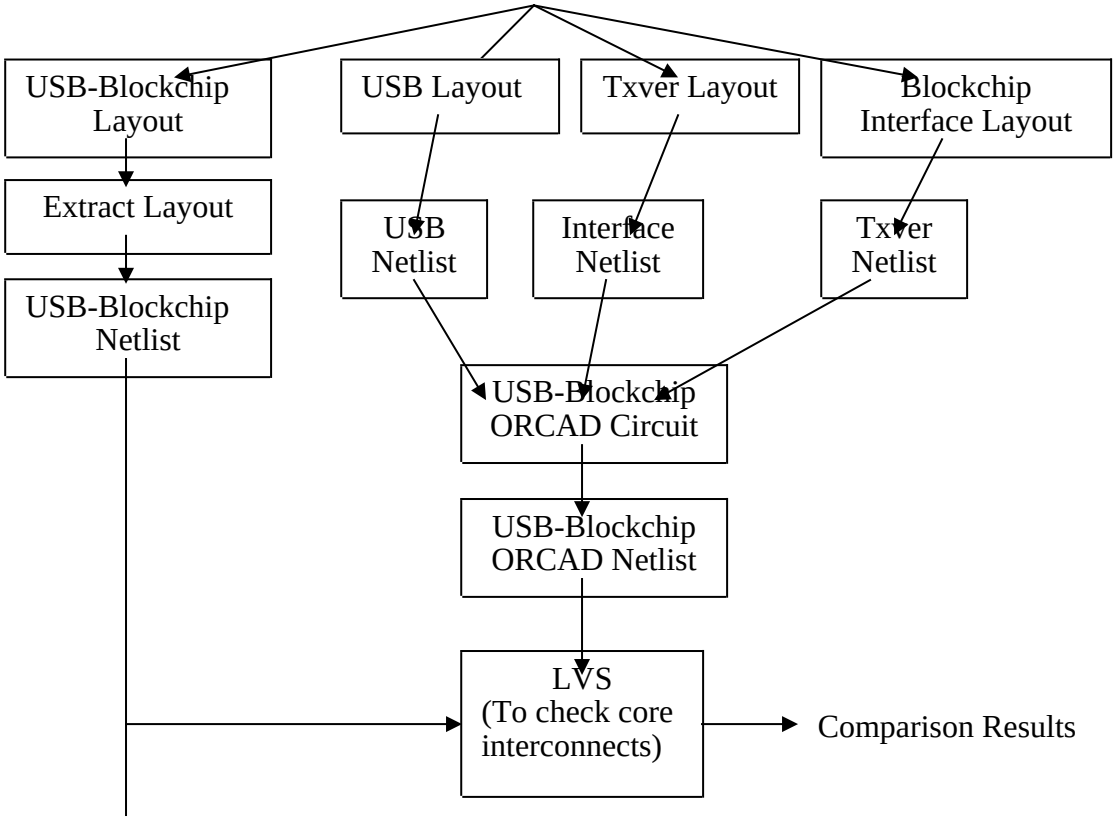


Figure 8.5. LVS Flow Chart.

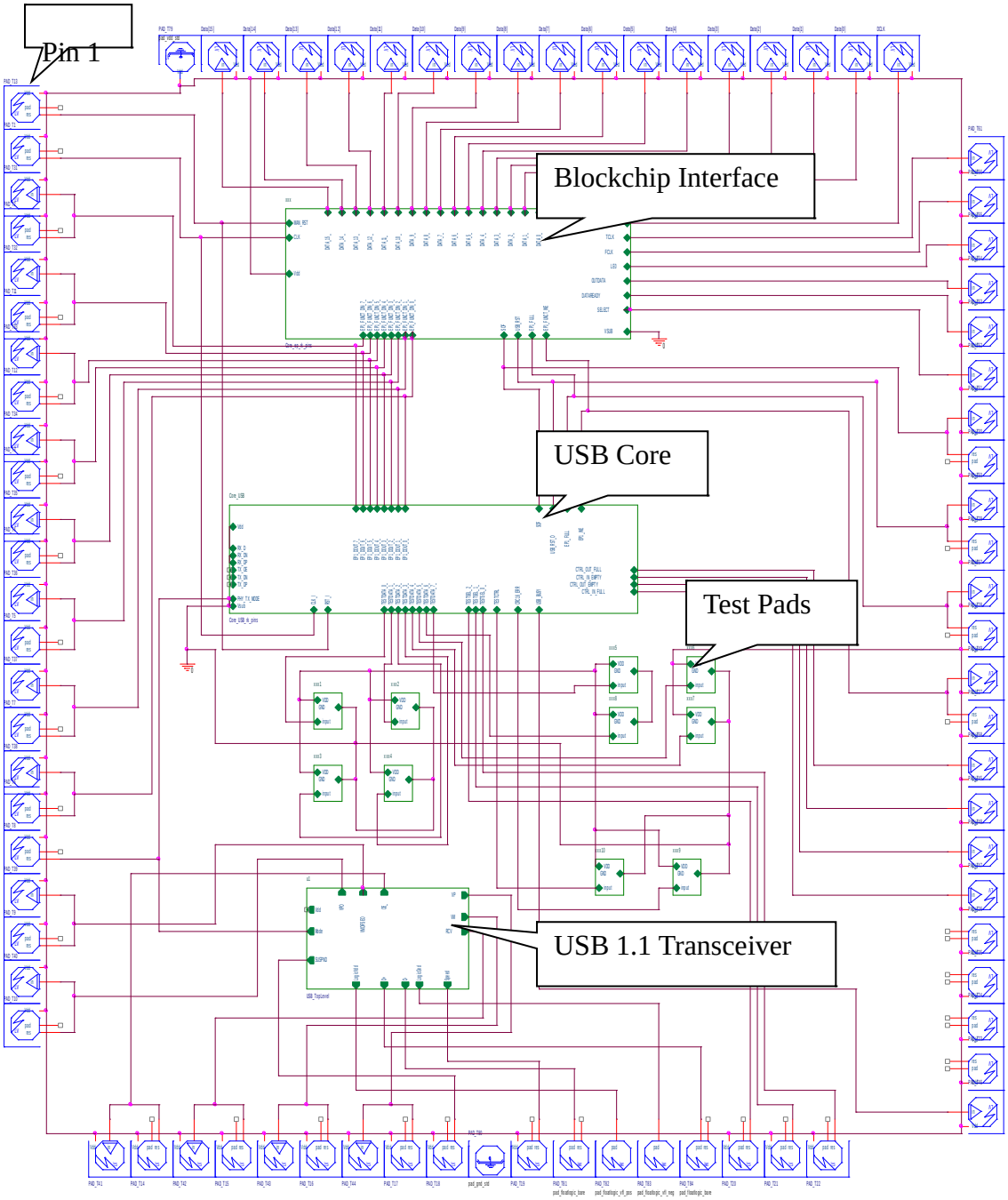


Figure 8.6. USB-Blockchip ORCAD Circuit.

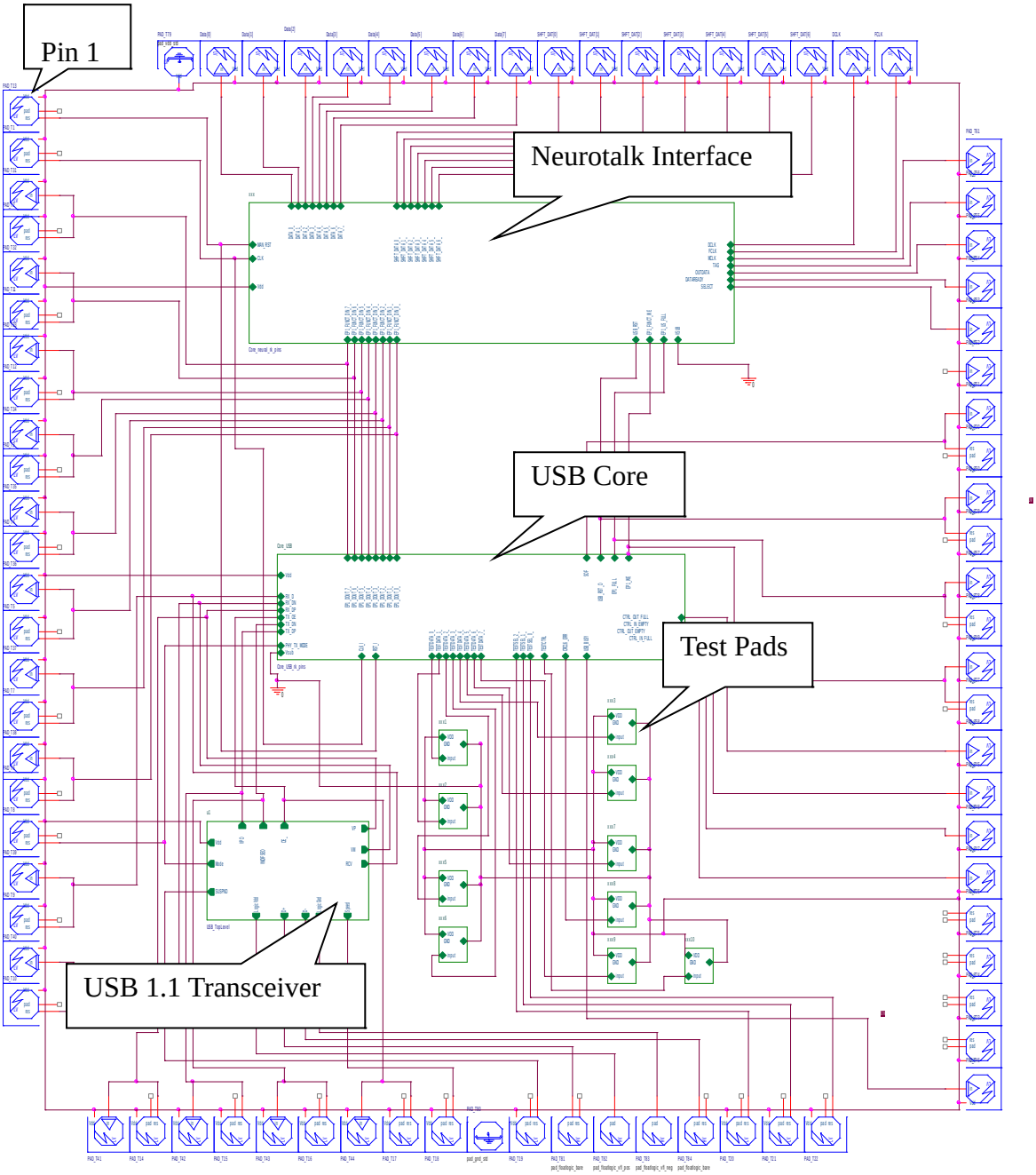


Figure 8.7. USB-Neurotalk ORCAD Circuit.

The schematics shown in figures 8.6 and 8.7 are too complex to be seen clearly. The motivation for implementing the schematics was to ensure the correctness of the interconnections when compared with the actual layout.

CHAPTER 9

DISCUSSION OF ACHIEVED RESULTS

The USB device was initially tested and validated in FPGA using the board shown in figure 6.8. A sample device driver and application provided with the Windows Device Driver Development Kit, were used. The device driver was used without being modified, however the application required minimal modification to send the correct instruction to the device.

9.1 FPGA PROTOTYPE TEST RESULTS

The blockchip and the neurotalk implementations were successfully tested on the FPGA prototype board. Figure 9.1 shows the signals on DCLK, DIN and LE. This figure closely matches figure 3.3, which shows the output waveform as described in the block chip specification document. Figure 9.2 shows a partial blockchip output waveform as it appears on an oscilloscope. The figure shows the channel output along with the DIN, DATA and LE signals. Figure 9.3 shows the complete blockchip output waveform at channel 1. The data the application sent to the USB was sent in decimal format. The 4-byte data integer was $(41{,}503)_{10}$ = $(00\ 00\ A2\ 1F)_{16}$. In binary this number is $(0000\ 0000\ 0000\ 0000\ 1010\ 0010\ 0001\ 1111)_2$. Since the USB outputs data in a little-endian format. The data that reaches the endpoint is as follows: 1^{st} byte = 0000 1111, 2^{nd} byte = 1010 0010. Hence the data received by the endpoint in binary is: $(0001\ 1111\ 1010\ 0010)_2$. The data transmitted translates into the waveform attributes shown in figure 9.1.

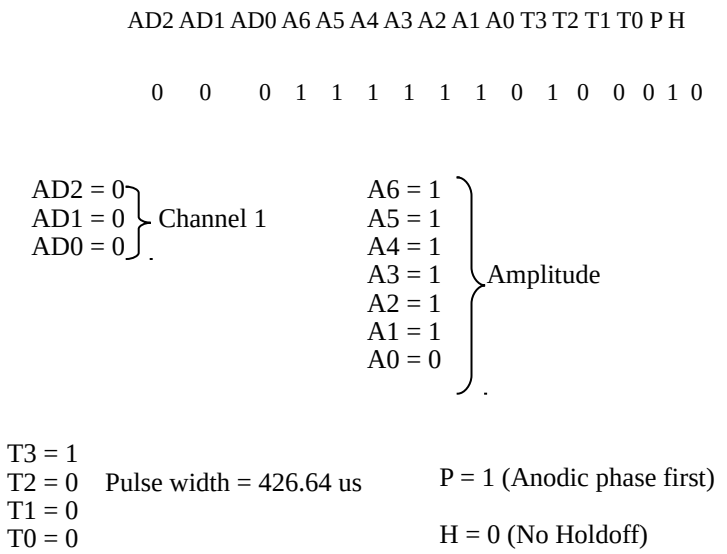


Figure 9.1. Blockchip Instruction Attributes.

Each pulse width increment starting with T3T2T1T0 = 0001 is in 53.33 us increments. Since T3T2T1T0 = 1000 the 8th increment after T3T2T1T0 = 0001, the pulse width is 53.33 us * 8 = 426.64 us. This number can be verified by looking at figure 9.4. Since the voltage obtained is the open circuit voltage across the channel, i.e. it is not loaded, the signal hits the compliance voltage, hence the amplitude cannot be verified. However the fact that the pulse width is as expected we can safely assume that the system is works correctly. The pulse polarity is ‘1’, which means the output waveform will be anodic first.

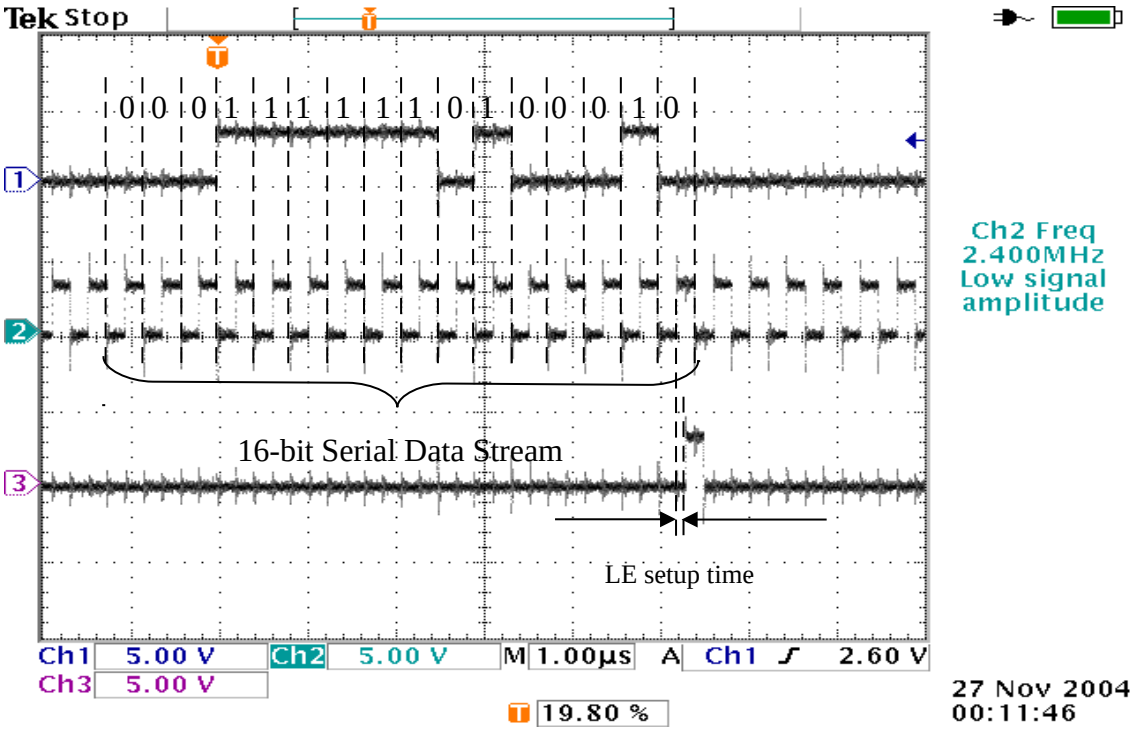


Figure 9.2. Oscilloscope Screenshot of Blockchip Signals.

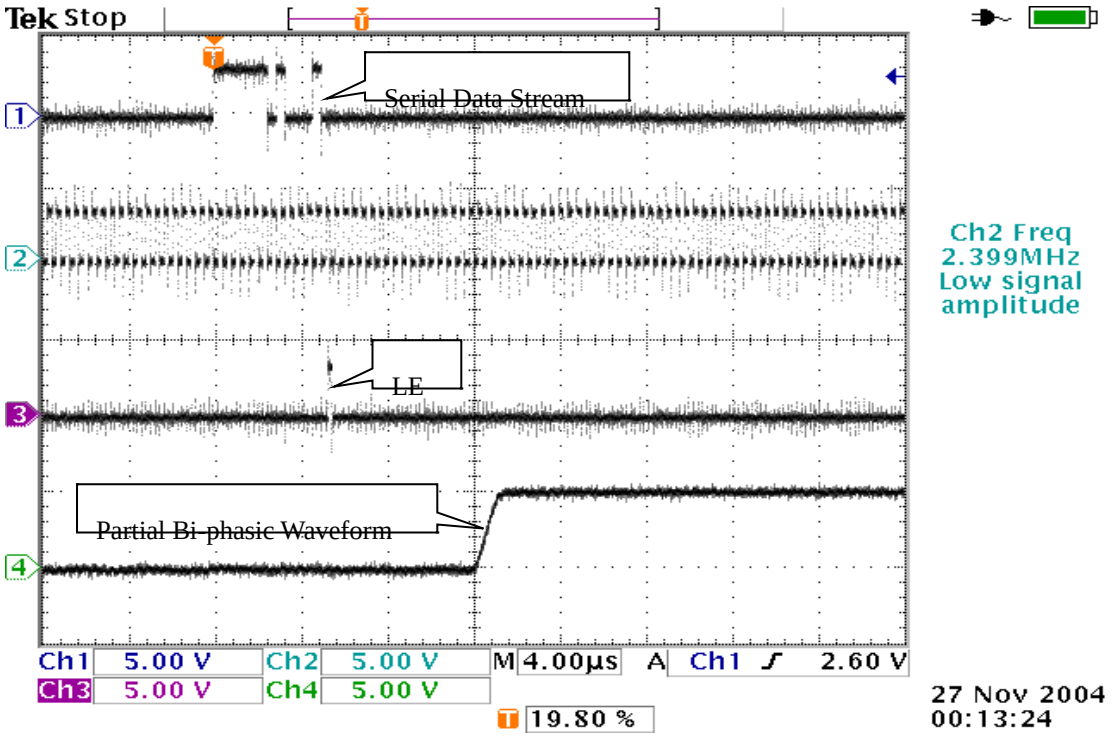


Figure 9.3. A Partial Oscilloscope Screenshot of Blockchip Channel Output.

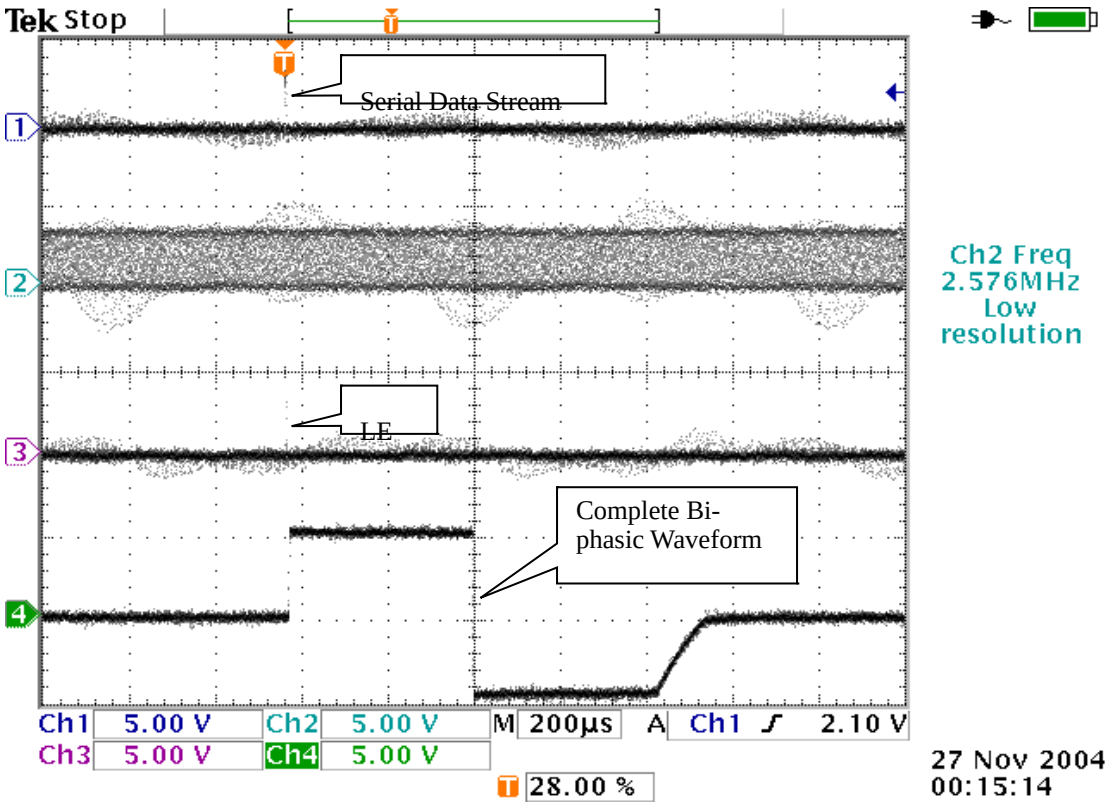


Figure 9.4. A Complete Oscilloscope Screenshot of Blockchip Channel Output.

The neurotalk version of the device was similar to that of the block chip design. The difference between them lies in the different number and kinds of signals. The neurotalk interface requires three signals: serial data output (ODATA), data clock (DCLK) and reset (TAG). Before starting any transmission, a pulse needs to be generated on the TAG signal to reset the neurotalk target device. The data output available at the first rising edge of the DCLK, after the TAG goes low, is considered valid. In addition, the most significant bit of each byte in the instruction is reserved for purposes discussed below. Figure 9.3 shows the signal lines for the neurotalk version of the device and a close up of the instruction as it is written to the endpoint FIFO.

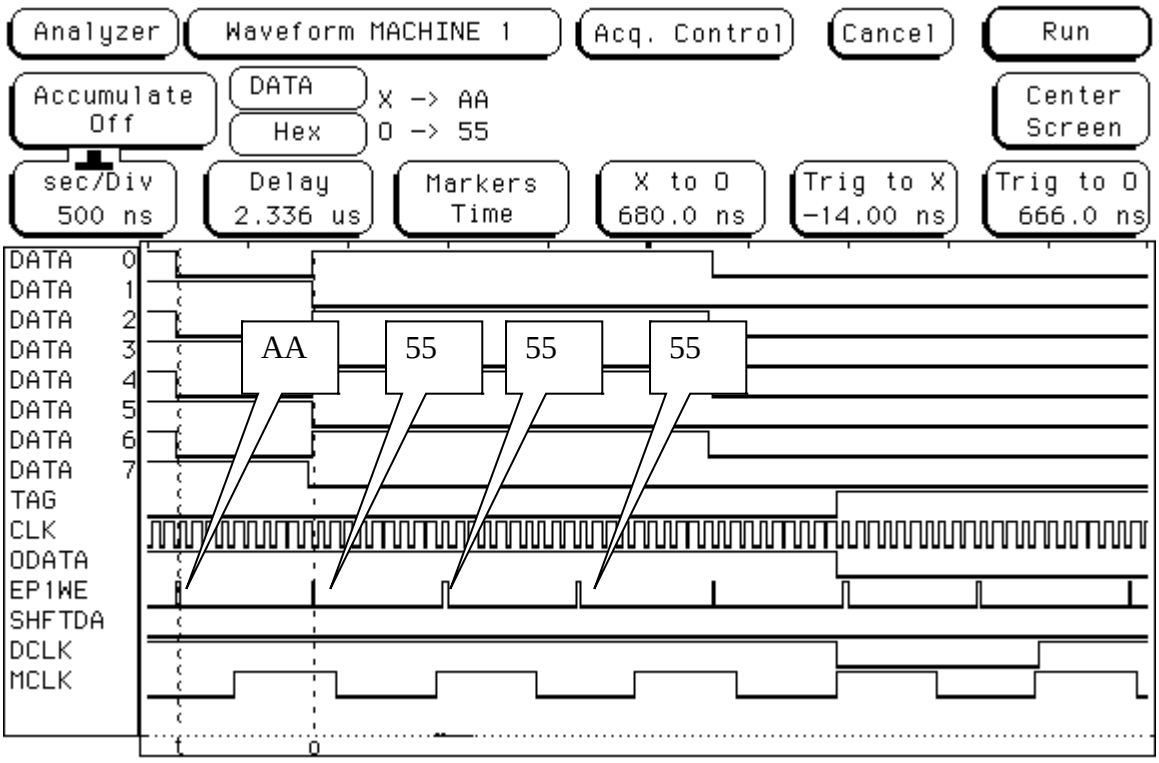


Figure 9.5. Closeup of Neurotalk Output Signals.

The data that is sent to the neurotalk interface is a 4-byte data packet as shown in figure 9.3. In binary, the data translates to: 1010 1010 0101 0101 0101 0101 0101 0101. Since the 7th bit of each byte is reserved, it is not counted in the instruction stream. The instruction stream is: 010 1010 101 0101 101 0101 101 0101. The signal output obtained from the stream is shown in figure 9.6. Figure 9.7 shows the complete output of the neurotalk interface state machine. It shows the TAG signal, the data output signal and the data clock. I can be verified visually that the data output stream of figure 9.7 matches with that of figure 9.6.

0 1 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 1 0 1 0 1 0 1

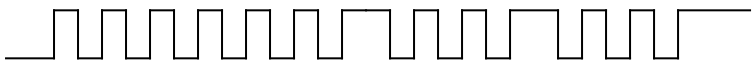


Figure 9.6. Neurotalk Data Stream.

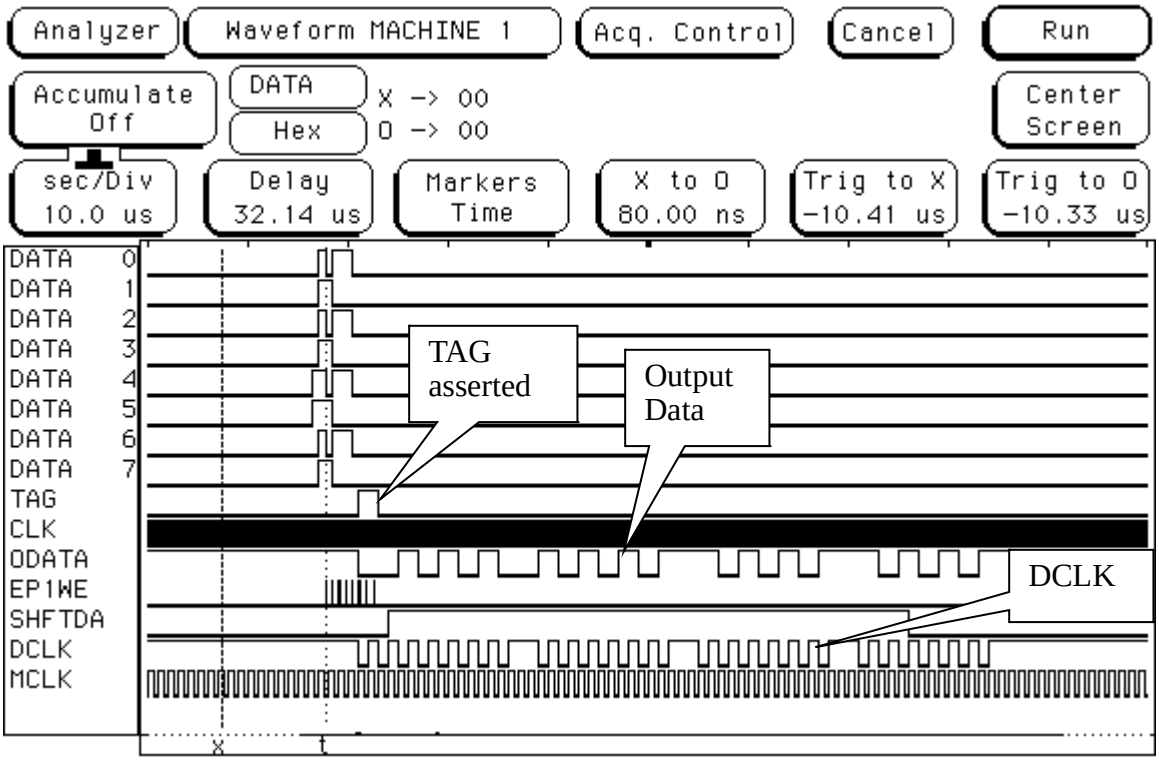
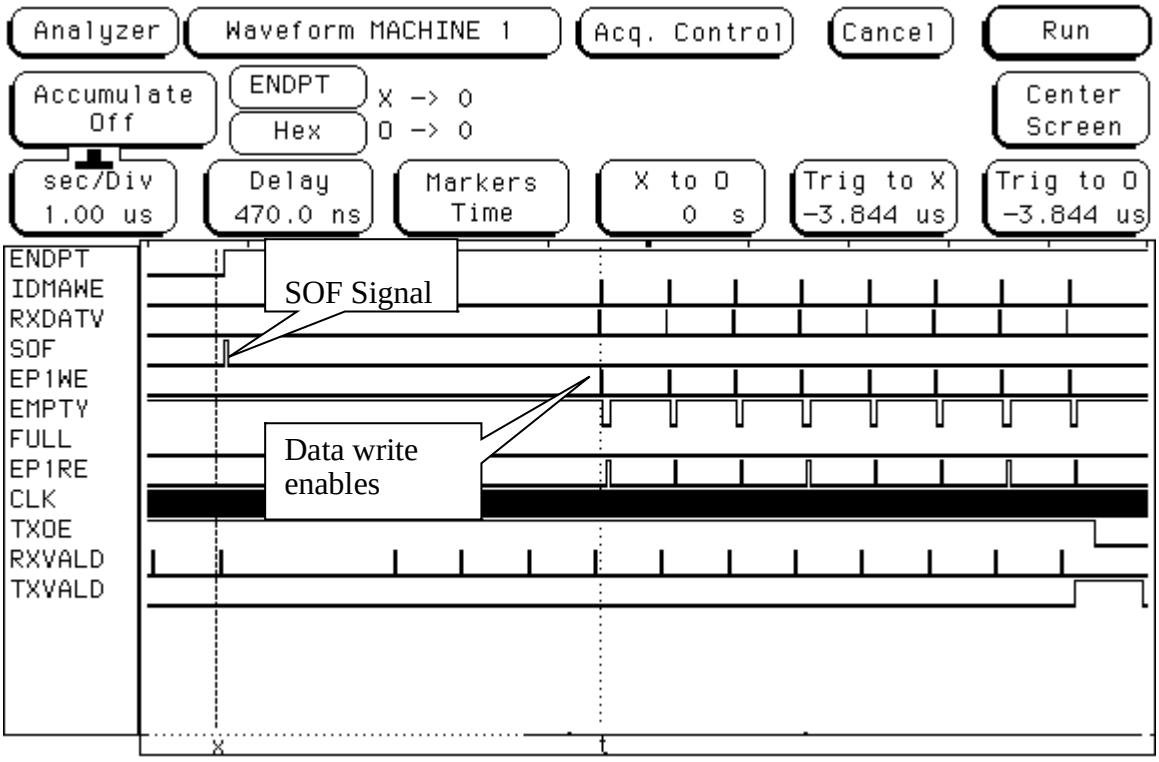


Figure 9.7. Neurotalk Output Signals.

A start-of-frame (SOF) signal was also implemented. It is designed to output a short pulse and is an output from the USB core FPGA. While writing the specification for the neurotalk interface logic, it was discovered that there was a need for the neurotalk compatible external chip to be apprised of a start of data frame. The USB core sets a bus to endpoint 1 before starting to send data to the endpoint. This bus was used to determine the exact timing of the SOF signal. It was soon discovered that using the SOF would lead to another potential problem. This problem surfaces if immediately after the very first instruction, another instruction is sent, the new instruction does not have a SOF signal

preceding it. In an ideal condition this would pose no problem. However, if any one byte were to be missed, due to a missing write enable, the wrong instruction would be read in, and the state machine will not be synchronous with the USB core. To solve this problem, the instruction format was redesigned in such a way that the most significant bit of the first byte will always be a logic ‘1’. All subsequent bytes of that particular instruction will have a logic ‘0’ as the most significant bit. Such an addition would make the interface synchronous with the USB core at every instruction. Figure 9.8 shows the SOF signal. The data packet closely follows the SOF pulse as shown.

Figure 9.8. Implementation of the SOF.



CHAPTER 10

DISCUSSION

The initially specified goals of the project have been successfully achieved. This chapter discusses the overall project, including the problems that were faced during the implementation, and recommendations for future improvements in the design and hardware implementation.

10.1 USB DEVICE IMPLEMENTATION

As stated in chapter 6 and 8 the prototype FPGA device was implemented without using PCB's. Figure 7.1 shows the bottom view of the device. It can be easily seen that the manual connections are complex and hence prone to loose connections. Such a problem was frequently faced while building and debugging the design. Often, while the device was working, one of the FPGA's would lose its configuration and would have to be programmed again. The other problem that was faced was validating the USB core that was obtained as an open source implementation. Since it did not have documentation, it took a considerable amount of time to configure and debug.

The software used for the device is the application that was supplied by the Windows Driver Development kit. It is written in C and requires a driver build environment. The driver is a generic version that is compatible with devices that support only bulk transfers. Although the driver works, it has not been optimized for use with any one specific device.

10.2 RECOMMENDATIONS

The FPGA development board that was designed required extensive soldering to function correctly. Over time due to the use of the device, the solder connections degrade

and either need to be re-done, or the connecting wires need to be replaced. To avoid such extensive maintenance and replacement, the device could be implemented as a PCB. Using a PCB however could create other problems. For instance, making a change to a PCB is more difficult compared to making a change in the manually constructed device. To overcome such a disadvantage, each major component can be soldered to it own PCB. The individual components can then be interconnected using connectors and can be made to connect any pin on one FPGA to any pin on other devices. Such a PCB system is shown in figure 10.1.

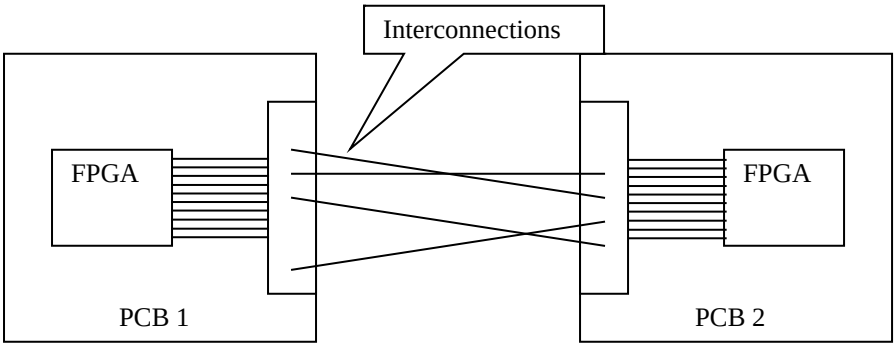


Figure 10.1. Proposed PCB based Development System.

Such a system will make sure that FPGA's don't lose configuration due to bad connections. It will enable quick re-configuration of the pin connections, provided the correct type of connector is used to connect the two PCB's.

In addition to the above-mentioned changes, the device driver used for the development board can be optimized more by implementing it at a lower level in the operating system stack. The application currently is executed using a command line console provided along with the windows driver development kit, and can be improved by adding a graphical user interface for better user interaction and control of the output waveform attributes.

10.3 TEST RESULTS

During the implementation and testing of the device a number of problems were faced. Due to the absence of accompanying documentation for USB core it was time consuming to get the device to work. In addition, due to insufficient FPGA software support an error was induced in the FIFO implementation of the USB core. This error caused the enumeration of the device to fail. It was discovered that the FIFO outputs are required to be unregistered to function as expected. The other problem faced was with the main 48MHz and the internal clocks. This problem caused the internal logic to sometimes behave erratically thereby giving an inconsistent output. It was discovered that the main clock was skewed which also skewed its derivative clocks. Generating reduced frequency clocks in small steps solved the problem. Instead of generating the 2.4 MHz DCLK directly from the 48MHz main clk, an intermediate 4.8MHz FCLK was derived and then the DCLK was derived from the FCLK. This reduced the jitter noticeably and improved the performance of the interface logic.

Subsequent to solving the problems, the FPGA development board was validated and verified for the application of interfacing with the blockchip and the proposed neurotalk chips. The ASIC implementation of the two interfaces has been implemented and is currently being tested. In addition to the ASIC version, the FPGA based development board in itself can be used effectively as a stimulator system after the development of the device driver and application optimized for this application.

CHAPTER 11

CONCLUSION

The USB device was successfully implemented and tested in FPGA in spite of facing problems. The USB, hence appears to be a good solution for use as a link between the PC and the two microchips: Blockchip and Neurotalk. To be successfully used as a neuroprosthetic stimulator system, the complete systems needs to be implemented and tested in ASIC. In addition, the device driver needs to be optimized and the application needs to be modified to add a graphical user interface.

The system was implemented manually using a copper clad perforated board. This required that the connections be made using wire. Doing so provided the flexibility to change the connections as needed, however it turned out to be tedious to design and debug. The problem of degrading solder connections was also faced in a later stage of development. To overcome these problems, implementation in PCB is suggested for future implementation. Each electronic component soldered on a separate PCB gives a reliable connection, while letting the designer to be flexible enough to make changes

easily and reliably. The system has been implemented in ASIC and is currently being tested.

APPENDIX A

BLOCKCHIP INTERFACE VERILOG RTL CODE

```
`include "usb1_tech.v"
module usb1_interface(clk, //input
                      usb_rst, //input
                      clr, //input
                      man_rst, //input
                      ep1_funct_din, // input
                      ep1_funct_re, // output
                      ep1_we,
                      ep1_funct_full,//input
                      ep1_funct_empty,//input
                      Dclk, //output
                      Tclk, //output
                      Fclk, //output
                      LE0, //output
                      outData, //output
                      select,
                      dataready,
                      Data,
                      sof);//output
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                ENDPOINT                1:                BULK,                OUT
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// This data bus inputs data from the FIFO to the device function.
// It is then used by the function. This is used by endpoint 1
input [7:0] ep1_funct_din;
wire [7:0] ep1_funct_din;
// This is the read enable signal the goes into the FIFO.
output ep1_funct_re;
wire ep1_funct_re;
// This input comes in from the FIFO, it tells the device
// whether or not the FIFO is empty.
input ep1_funct_empty;
wire ep1_funct_empty;
input ep1_funct_full;
wire ep1_funct_full;
output LE0;
output Dclk;
output Tclk;
output Fclk;
wire LE0, Dclk, Tclk, Fclk;
output [15:0] Data;
```

```
wire [15:0] Data;
output dataready;
output select;


output outData;
wire outData;

input ep1_we;
wire ep1_we;

input clk, usb_rst, clr;// usb_rst coming from the USB core
wire clk, usb_rst, clr;

input man_rst; // man_rst coming from outside the FPGA

input sof;
wire sof;


reg [7:0] reg5;
//reg re;
reg LE0;
reg select;
//reg ff_enable;
reg enable_le;
reg shift;
reg sft_count;


reg ep1_funct_re;
//wire ep1_funct_re;

reg sr_dataready;

reg dataready;

reg [15:0] Data;
reg [15:0] bitfield;



reg [3:0] count1;
reg [3:0] count2;
reg [3:0] count3;

reg [8:0] shift_count;
reg Dclk, Tclk, Fclk;
```

```
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    // CLOCKS required by the block chips.
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    // Dclk at 2.5MHz =
    always @(negedge Fclk)
    begin
        if(!usb_rst) begin
                Dclk <=1'b0;
                count1 <= 4'd0;
        end
        else if (!man_rst) begin
                Dclk <= 1'b0;
                count1 <= 4'd0;
        end
        else if(count1 == 4'd1 && Dclk == 1'b0 && usb_rst == 1'b1 && man_rst ==
1'b1) begin
                Dclk <= 1'b1;
                count1 <= 4'd1;
        end
        else if(count1 == 4'd1 && Dclk == 1'b1 && usb_rst == 1'b1 && man_rst ==
1'b1) begin
                Dclk <= 1'b0;
                count1 <= 4'd1;
        end else count1 <= count1 + 4'd1;
    end


    // Fclk = 5MHz
    always @(posedge clk)
    begin
        if(!usb_rst) begin
                Fclk <= 1'b0;
                count3 <= 4'd0;
        end
        else if(!man_rst) begin
                Fclk <= 1'b0;
                count3 <= 4'd0;
        end
        else if(count3 == 5 && Fclk == 1'b0 && usb_rst == 1'b1 && man_rst == 1'b1)
begin
                Fclk <= 1'b1;
                count3 <= 4'd1;
        end
        else if(count3 == 5 && Fclk == 1'b1 && usb_rst == 1'b1 && man_rst == 1'b1)
begin
                Fclk <= 1'b0;
                count3 <= 4'd1;
        end
        else count3 <= count3 + 4'd1;
    end
```

```
    // Tclk at 312500Hz
    always @(posedge Dclk)
    begin
        if(!usb_rst) begin
                Tclk <= 1'b0;
                count2 <= 4'd0;
        end
        else if(!man_rst) begin
                Tclk <= 1'b0;
                count2 <= 4'd0;
        end
        else if(count2 == 4'd4 && Tclk == 1'b0 && usb_rst == 1'b1 && man_rst == 1'b1) begin
                Tclk <= 1'b1;
                count2 <= 4'd1;
        end
        else if(count2 == 4'd4 && Tclk == 1'b1 && usb_rst == 1'b1 && man_rst == 1'b1) begin
                Tclk <= 1'b0;
                count2 <= 4'd1;
        end else count2 <= count2 + 4'd1;
    end
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    // Read from FIFO
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


    reg [7:0] temp;
    reg [3:0] count_re;

    reg ep1_we1, ep1_we2;

    always @(posedge clk)
        ep1_we1 <= ep1_we;

    always @(posedge clk)
        ep1_we2 <= ep1_we1;



    ////////////////////////////////////////////////
    // Start reading from the OUT FIFO
    ////////////////////////////////////////////////

    always @(posedge clk)
    begin
        if(!usb_rst) begin
                count_re <= 4'd0;
        end
        if(sof) count_re <= 4'd0;
```

```
if(ep1_we2 & count_re < 4'd8) begin
        if(count_re == 4'd0) begin
                ep1_funct_re <= 1'b1;
                Data[15:8] <= ep1_funct_din;
                dataready <= 1'b0;
                count_re <= 4'd1;
        end
        if(count_re == 4'd1) begin
                ep1_funct_re <= 1'b1;
                Data[7:0] <= ep1_funct_din;
                dataready <= 1'b1;
                count_re <= 4'd2;
        end
        if(count_re == 4'd2) begin
                ep1_funct_re <= 1'b1;
                bitfield[15:8] <= ep1_funct_din;
                dataready <= 1'b1;
                count_re <= 4'd3;
        end
        if(count_re == 4'd3) begin
                ep1_funct_re <= 1'b1;
                bitfield[7:0] <= ep1_funct_din;
                dataready <= 1'b1;
                count_re <= 4'd4;
        end
        if(count_re == 4'd4) begin
                ep1_funct_re <= 1'b1;
                temp <= ep1_funct_din;
                dataready <= 1'b1;
                count_re <= 4'd5;
        end
        if(count_re == 4'd5) begin
                ep1_funct_re <= 1'b1;
                temp <= ep1_funct_din;
                dataready <= 1'b1;
                count_re <= 4'd6;
        end
        if(count_re == 4'd6) begin
                ep1_funct_re <= 1'b1;
                temp <= ep1_funct_din;
                dataready <= 1'b1;
                count_re <= 4'd7;
        end
        if(count_re == 4'd7) begin
                ep1_funct_re <= 1'b1;
                temp <= ep1_funct_din;
                dataready <= 1'b0;
                count_re <= 4'd0;
        end
end
```

```
        else if(ep1_we2 == 1'b0) begin
                ep1_funct_re <= 1'b0;
                if(dataready == 1'b0) dataready <= 1'b0;
                else if(dataready == 1'b1) dataready <= 1'b1;
        end
    end
    //
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    // Shift out data for 16 Dclk cycles
    //
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    always @(posedge Fclk) begin

        if(!usb_rst) begin
                shift_count <= 6'd0;
                sr_dataready <= 1'b0;
                select <= 1'b1;
                sft_count <= 1'b1;
                LE0 <= 1'b0;
        end
        else if(!man_rst) begin
                shift_count <= 6'd0;
                sr_dataready <= 1'b0;
                select <= 1'b1;
                sft_count <= 1'b1;
                LE0 <= 1'b0;
        end

        if(sr_dataready == 1'b0)
                shift_count <= 6'd0;


        if(sr_dataready == 1'b1 & shift_count == 6'd0 & Dclk == 1'b0) begin
                shift_count <= 6'd1;
                select <= 1'b1;
                LE0 <= 1'b0;
        end
        else if(sr_dataready == 1'b1 & shift_count == 6'd0 & Dclk == 1'b1) begin
                shift_count <= 1'b0;
                select <= 1'b0;
                LE0 <= 1'b0;
        end
        else if(sr_dataready == 1'b1 & shift_count <6'd38) shift_count <= shift_count +
6'd1;


        if (dataready == 1'b1 & shift_count == 6'd0) begin
                sr_dataready <= 1'b1;
        end
        else if (dataready == 1'b0 & shift_count == 6'd36) begin //691
                sr_dataready <= 1'b0;
```

```
            sft_count <= 1'd0;
            select <= 1'b1;
        end


        if(sr_dataready == 1'b1 & shift_count == 6'd1) begin
            select <= 1'b1;
            LE0 <= 1'b0;
        end

        if(sr_dataready == 1'b1 & shift_count == 6'd2 & Dclk == 1'b0) begin
            select <= 1'b0;
            LE0 <= 1'b0;
        end

        if(sr_dataready == 1'b1 & shift_count == 6'd3) begin
            select <= 1'b0;
            LE0 <= 1'b0;
        end

        if(sr_dataready == 1'b1 & shift_count > 6'd3 & shift_count < 6'd34) begin
            select <= 1'b0;
            LE0 <= 1'b0;
        end

        if(sr_dataready == 1'b1 & shift_count == 6'd33 & Dclk == 1'b1) begin
            select <= 1'b1;
            LE0 <= 1'b1;
        end

        if(sr_dataready == 1'b1 & shift_count == 6'd34) begin
            select <= 1'b1;
            LE0 <= 1'b0;
        end
    end



    //
=================================================================
====
    // Synthesize in FPGA
    //
=================================================================
====

    `ifdef FPGA
        lpm_shiftreg shft_reg(.data(Data),
                                                    .clock(!Dclk),
                                                    .enable(enable),
```

```
                                                        .load(select),
                                                        .shiftout(outData));
        defparam shft_reg.lpm_width = 16;
    //
================================================================
====

    //
================================================================
====
    // Synthesize in ASIC
    //
================================================================
====
    `else `ifdef ASIC

        mux21 mux0(.in1(Data[0]), .in0(1'b0), .sel(select), .out(dff_0_i));
        dff_i df0(.data(dff_0_i), .q(dff_0_o), .clock(!Dclk), .enable(1'b1));

        mux21 mux1(.in1(Data[1]), .in0(dff_0_o ), .sel(select), .out(dff_1_i));
        dff_i df1(.data(dff_1_i), .q(dff_1_o), .clock(!Dclk), .enable(1'b1));

        mux21 mux2(.in1(Data[2]), .in0(dff_1_o), .sel(select), .out(dff_2_i));
        dff_i df2(.data(dff_2_i), .q(dff_2_o), .clock(!Dclk), .enable(1'b1));

        mux21 mux3(.in1(Data[3]), .in0(dff_2_o), .sel(select), .out(dff_3_i));
        dff_i df3(.data(dff_3_i), .q(dff_3_o), .clock(!Dclk), .enable(1'b1));

        mux21 mux4(.in1(Data[4]), .in0(dff_3_o), .sel(select), .out(dff_4_i));
        dff_i df4(.data(dff_4_i), .q(dff_4_o), .clock(!Dclk), .enable(1'b1));

        mux21 mux5(.in1(Data[5]), .in0(dff_4_o), .sel(select), .out(dff_5_i));
        dff_i df5(.data(dff_5_i), .q(dff_5_o), .clock(!Dclk), .enable(1'b1));

        mux21 mux6(.in1(Data[6]), .in0(dff_5_o), .sel(select), .out(dff_6_i));
        dff_i df6(.data(dff_6_i), .q(dff_6_o), .clock(!Dclk), .enable(1'b1));

        mux21 mux7(.in1(Data[7]), .in0(dff_6_o), .sel(select), .out(dff_7_i));
        dff_i df7(.data(dff_7_i), .q(dff_7_o), .clock(!Dclk), .enable(1'b1));

        mux21 mux8(.in1(Data[8]), .in0(dff_7_o), .sel(select), .out(dff_8_i));
        dff_i df8(.data(dff_8_i), .q(dff_8_o), .clock(!Dclk), .enable(1'b1));

        mux21 mux9(.in1(Data[9]), .in0(dff_8_o), .sel(select), .out(dff_9_i));
        dff_i df9(.data(dff_9_i), .q(dff_9_o), .clock(!Dclk), .enable(1'b1));

        mux21 mux10(.in1(Data[10]), .in0(dff_9_o), .sel(select), .out(dff_10_i));
        dff_i df10(.data(dff_10_i), .q(dff_10_o), .clock(!Dclk), .enable(1'b1));

        mux21 mux11(.in1(Data[11]), .in0(dff_10_o), .sel(select), .out(dff_11_i));
        dff_i df11(.data(dff_11_i), .q(dff_11_o), .clock(!Dclk), .enable(1'b1));
```

```
        mux21 mux12(.in1(Data[12]), .in0(dff_11_o), .sel(select), .out(dff_12_i));
        dff_i df12(.data(dff_12_i), .q(dff_12_o), .clock(!Dclk), .enable(1'b1));

        mux21 mux13(.in1(Data[13]), .in0(dff_12_o), .sel(select), .out(dff_13_i));
        dff_i df13(.data(dff_13_i), .q(dff_13_o), .clock(!Dclk), .enable(1'b1));

        mux21 mux14(.in1(Data[14]), .in0(dff_13_o), .sel(select), .out(dff_14_i));
        dff_i df14(.data(dff_14_i), .q(dff_14_o), .clock(!Dclk), .enable(1'b1));

        mux21 mux15(.in1(Data[15]), .in0(dff_14_o), .sel(select), .out(dff_15_i));
        dff_i df15(.data(dff_15_i), .q(outData), .clock(!Dclk), .enable(1'b1));

    `endif // ASIC
    `endif // FPGA
    //
//=================================================================
====
    endmodule
```

BIBLIOGRAPHY

[1] Akin T., Najafi K., Bradley R.M. A Wireless Implantable Multichannel Digital Neural Recording System for a Micromachined Sieve Electrode. IEEE Journal of Solid-State Circuits, January 1998.

[2] Ben-Haim S. A., Anuchink C.L., Dinnar U. A Computer Controller for Vest Cardiopulmonary Resuscitation (CPR). IEEE Transactions on BME, May 1988.

[3] Broberg R., Hubbard A. A Custom-Chip Based Functional Electrical Stimulation System. IEEE Transactions on BME, September 1994.

[4] Buckett J.R., Peckham P. H., Thrope G. B., Braswell S. D., Keith M. W. A flexible, Portable System for Neuromuscular Stimulation in the Paralyzed Upper Extremity. IEEE Transactions on BME, November 1988.

[5] Capelle C., Trullemans C., Arno P., Veraart C. A Real-Time Experimental Prototype for Enhancement of Vision Rehabilitation using Auditory Substitution. IEEE transactions on BME, October 1998

[6] Cheever E.A., Thompson D.R., Cmolik B.L., Santamore W.P., George D.T. A Versatile Microprocessor-Based Multichannel Stimulator for Skeletal Muscle Cardiac Assist. IEEE Transactions on BME, January 1998

[7] Connor S.B., Quill T.J., Jacobs J.R. Accuracy of Drug Infusion Pumps Under Computer Control. IEEE Transactions on BME, September 1992

[8] Hartov A., Mazzarese R.A., Reiss F.R., Kerner T.E., Osterman S., Williams D.B.,

Paulsen K.D. A Multichannel Continuously Selectable Multifrequency Electrical Impedance Spectroscopy Measurement System. IEEE Transactions on BME, January 2000.

[9] Hinterberger T., Schmidt S., Neumann N., Mellinger J., Blankertz B., Curio G., Birbaumer N. Brain-Computer Communication and slow cortical Potentials. IEEE Transactions on BME, June 2004.

[10] Ignagni A.R., Buckett J. R., Peckham P.H. A Programming and Data Retrieval System for an Upper Extremity FES Neuroprosthesis. Annual International Conference of the IEEE Engineering in Medicine and Biology Society, 1990

[11] Kaczmarek K.A., Kramer K.M., Webster J.G., Radwin R.G. A 16-channel 8-Parameter Waveform Electrotractile Stimulation System. IEEE Transactions on BME, October 1991.

[12] Kanhai J.K.K., Caspers P.J., Reinders E.G.J., Pompe J.C., Bruining H.A., Puppels G.J. A Fast digitally Controlled Flow Proportional Gas Injection System for Studies in Lung Function. IEEE Transactions on BME, November 2003.

[13] Krief B., Dye R., Tucker J.H., Brugal g., Chassery J.M. A New Approach to Man Machine Communication for Computerized Microscopy. IEEE Transactions on BME, March 1994

[14] Lawrence T.L., Schmidt R.N. Wireless In-Shoe Force System. IEEE-EMBS International Conference Proceedings, Oct-Nov 1997

[15] Mesic S., Babuska R., Hoogsteden H.C., Verbraak A.F.M. Computer Controlled Mechanical Stimulation of the Artificially Ventilated human Respiratory System. IEEE Transactions on BME, June 2003.

[16] Morris L. R, Barszczewski P. Algorithms, Hardware, and Software for a Digital Signal Processor Microcomputer-Based Speech Processor in a Multielectrode Cochlear Implant System. IEEE Transactions on BME, June 1989.

[17] Muller G.R., Neuper C., Pfurtscheller G. Implementation of a Telemonitoring System for the Control of an EEG-Based Brain-Computer Interface. IEEE Transactions on Neural Systems and Rehabilitation Engineering, March 2003

[18] Polak M., Kostov A. Development of Brain-Computer Interface: Preliminary Results. IEEE-EMBS International Conference Proceedings, Oct-Nov 1997

[19] Rollins D.L., Killingsworth C.R., Walcott G.P, Justice R.K., Ideker R.E., Smith W.M. A Telemetry System for the Study of Spontaneous Cardiac Arrhythmias. IEEE Transactions on BME, July 2000

[20] Sawan M., Duval F., Hassouna M.M., Li J., Elhilali M.M, Lachance J., Leclair M., Pourmehdi S., Mouine J. Computerized Trancutaneous Control of a Multichannel Implantable Urinary Prosthesis. IEEE Transactions on BME, June 1992

[21] Schalk G., McFarland D.J., Hinterberger T., Birbaumer N., Wolpaw J.R. BCI2000:

A general purpose Brain-Computer Interface System. IEEE Transactions on BME, June 2004

[22] Schuessler T.F., Bates J.H.T. A Computer-Controlled Research Ventilator for small Animals: Design and Evaluation. IEEE Transactions on BME, September 1995

[23] Suaning G.J., Lovell N.H. CMOS Neurostimulation ASIC with 100 channels, Scalable Output, and Bidirectional Radio-Frequency Telemetry. IEEE Transactions on Biomedical Engineering, February 2001

[24] Zeng S., Powers J.R., Hsiao H. A New Video-Synchronized Multichannel Biomedical Data Acquisition System. IEEE Transactions on BME, March 2000

[25] Zhu H., Harris G.F., Wertsch J.J., Tompkins W.J., Webster J.G. A Microprocessor-Based Data-Acquisition System for Measuring Plantar Pressure from Ambulatory Subjects. IEEE Transactions on BME, July 1991