Final Report

# USB Data Logger

*Authors:*
Spencer Julian
David Kauer
John Wyant
Jintao Zhang

*Course:* ECE 337
*Lab:* Tuesday 2:30 - 5:30
*TA:* Tom Pollard

April 28, 2012

# Contents

## 1 Executive Summary

The team for this project successfully created an ASIC design that reads incoming data from a USB interface and stores the data onto a microSD card. The purpose of this device would be to gain potentially valuable information such as passwords, webcam discussions, typed documents, or debugging information. This makes the device particularly useful for certain tasks such as espionage, debugging a USB device, or ensuring that critical components properly communicate via USB. An important part of this device is that the connections are subtle and transparent in order to avoid disrupting the user and calling unwanted attention upon itself.

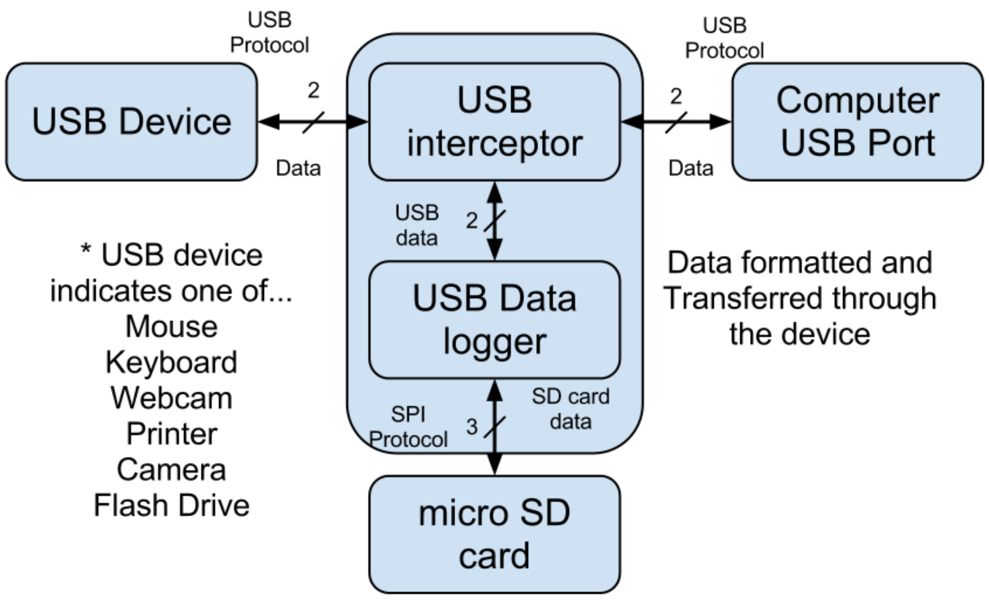
It also records all intercepted data to a microSD card, which allows the device to store a massive amount of data. The device records to a microSD card because it is a fast and easy method to collect data, and it allows for new data to be stored onto another microSD card when the original one is pulled out to examine data. The USB logger is designed to intercept any USB handshakes and convert them from USB 2.0 or USB 3.0 to a USB 1.1 connection. After converting the handshake, the data logger becomes transparent in the communication across the USB bus and begins to write the data passing through it onto the microSD card.

The remainder of this document describes the design specifications and the final architecture of the chip, which includes a functional block diagram. It will show the input and output pins on the chip, interfacing specifications, operational characteristics, and specific requirements necessary to run the device. This report will also describe the USB and microSD protocols involved in designing and testing, as well as the timing and area budgets of the USB logger. Finally, there is a description of the tests that are performed on the design as well as the results that can be taken from those tests, and a description of the chip layout.

# 2 Design Specifications

## 2.1 Interfacing Specifications

Figure 2.1: The USB Data Logger Usage Digram. The central box is the device.



The protocols used, as defined in figure 2.1, above, are:

**USB 1.1**
Transmission rate of 12 Mbps.

**SPI**
Transmission rate of 50Mbps after initalization, 400kbps during initalization.

## 2.2 Operational Characteristics

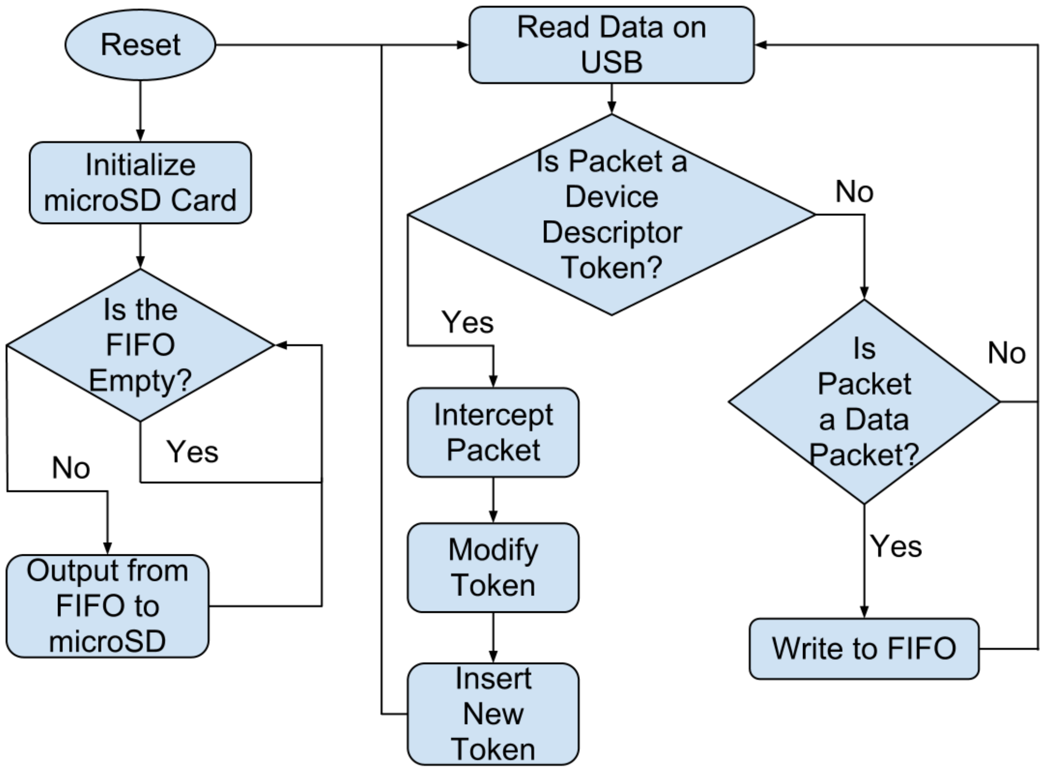
### 2.2.1 Pin-out

Table 2.1: The USB Data Logger Pin-Out table.

| **Signal Name** | **Type** | **Number of Bits** | **Data Format** | **Description and Timing Constraints** |
|---|---|---|---|---|
| clk | In | 1 | System Clock | Clock signal for the chip |
| rst | In | 1 | Active Low | Asynchronous reset signal for the device's memory elements. |
| PWR | In | 1 | Power | Power Input |
| GND | In | 1 | Ground | Ground Line |
| dataPlusUsb | Bidirectional | 1 | LSB NRZI | USB Data+ Line |
| dataMinusUsb | Bidirectional | 1 | LSB NRZI, inverted Data+ Line, above | USB Data- Line |
| dataPlusComputer | Bidirectional | 1 | LSB NRZI | Computer Data+ Line |
| dataMinusComputer | Bidirectional | 1 | LSB NRZI, inverted Data+ Line, above | Computer Data- Line |
| scl | Out | 1 | Clock | Clock output to microSD Card |
| sd_enable | Out | 1 | Active Low | Chip Select for microSD Card |
| dataOut | Out | 1 | Active High Binary | MOSI Data Output to microSD Card |

### 2.2.2 Chip Operation Sequence

Figure 2.2: A flowchart of the USB Data Logger's operation



#### General Operation

As shown in figure 2.2 above, the operation has two concurrent timelines operating after reset. The first timeline, shown as the left branch (going down from the reset block), is for the microSD card, and the second timeline (from the right of the reset block) is for USB communication.

#### USB Operation

As soon as the device resets, it begins listening on the USB line for any packet coming through. If the packet is a device descriptor token, identified by a specific PID, then it will intercept the packet and insert our modified description. It will only intercept the two relevant bytes to change the USB version number to USB 1.1, identified in hexidecimal, most significant bit first, as 0x0110. After it modifies the description, it injects it into the line, and the computer will read our modified description instead of the one sent by the device.

This process delays the entire line slightly, but the delay is consistent, and remains even when the packet is not a device descriptor, so it will cause no errors in communication.

If the packet is a data packet, then it will write the data to an internal FIFO block where it will wait to be written to the microSD card. The device will only write data from the data packets, the rest of the packet will be ignored and discarded by the device for writing, though it will still pass through the device without issue to ensure proper operation of both connected devices.

### SD Operation

When the device resets, the microSD card must be initialized. This is a relatively long process, as it takes one half of a millisecond. The card is initialized by waiting 80 clock cycles at 400 kHz, followed by outputting a command to switch it into SPI mode from SD mode and prepare for writing. Once the card is initialized we can speed up to the full 50 MHz for writing.

Unfortunately, due to a limitation in the inital design of our device, we cannot listen for card responses via MISO. However, all tested cards have been shown to respond favorably within a limited period of time, so our commands wait for this period of time before continuing writes.

After the microSD card is initialized, we see if the FIFO is empty. If it is empty, the SD clock will not be running, and we will simply wait for data to enter the FIFO. If the FIFO is not empty, we will output each byte that is inside the FIFO until it is empty, where we will stop the SD clock and wait for more data.

After a CMD25, which is the continous write command, the SD protocol usually defines a stop bit to indicate when we are finished writing to the device. However, due to the continuous operation of our device with no appropriate ”stop” signal, other than unplugging the device thus ceasing all operation, we have not implemented this stop signal. It should not be a concern as we are performing a bit-by-bit write via SPI instead of writing using the SD protocol.

## 2.3 Requirements

As the USB logger tricks the device that is plugged into it to use the USB 1.1 protocol, I/O speed and internal clock speed are not a big concern, as USB 1.1 runs at a speed of 12 Mbps. The devices clock speed can run at 140 MHz. Power is also not a big concern, since 5+ volts and ground are leeched from the USB, which should provide more than enough power for both our device as well as the device that is plugged into it. Therefore, area is our most critical design restriction.

The area of the chip must be small, as the entire device must not be larger than the size of a microSD card. If external storage was not necessary, it would be ideal to have a chip that could fit on a USB connector so that it would be very difficult to detect. As the primary use of this device will be for secretive data collecting, the harder it is to find, the better.

An added bonus of decreasing the size is decreasing power. While our power constraints are minimal, if we decrease power that will make it more difficult for someone to notice the device, should it be plugged into something power-hungry, such as a hard drive. Although the benefits of this could be moot, as with any USB 2.0 or 3.0 device the transfer speeds will be significantly less than expected due to the degradation to the USB 1.1 protocol.

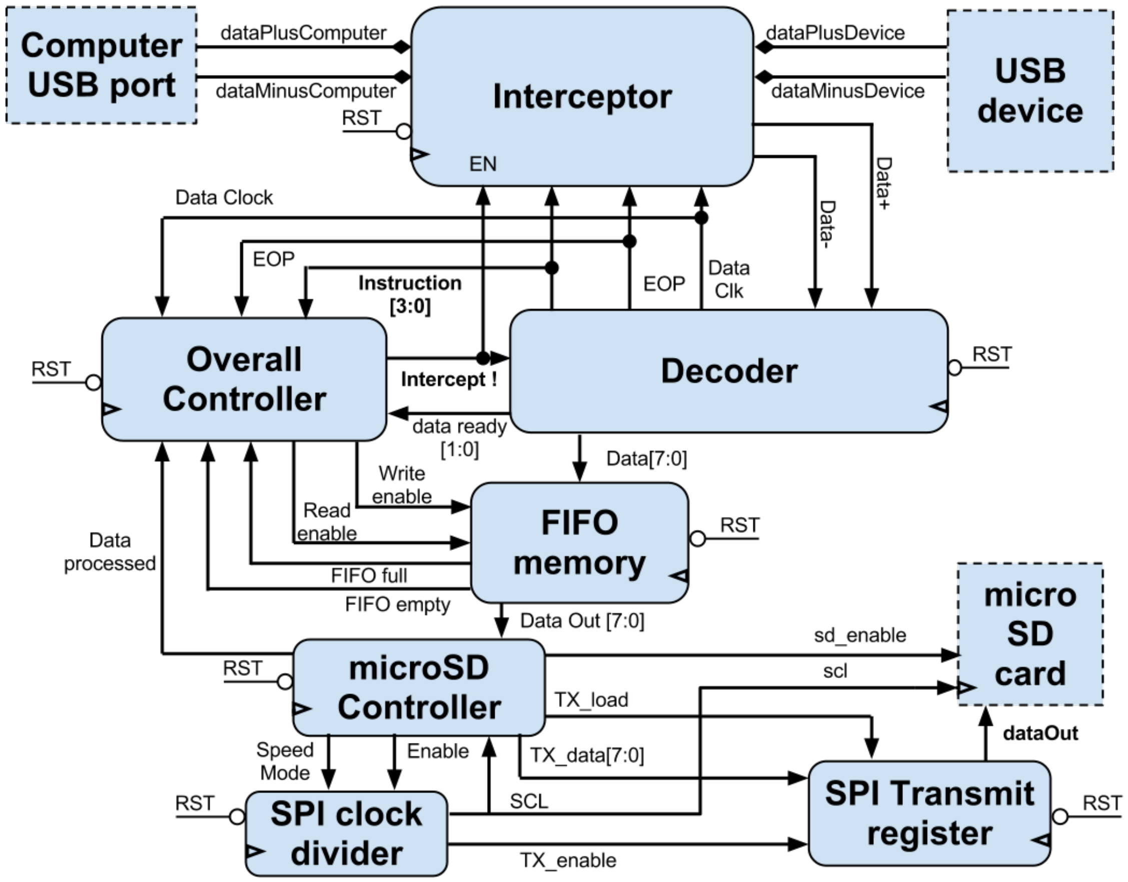
In the end, our specifications are a throughput of 12 Mbps via USB, 50 Mbps to the microSD card, a system clock speed of 140 MHz, a pin count of 2 USB input pins + 2 USB output pins + 3 MicroSD I/O pins + vcc/gnd + rst + sysclk = 11 pins, and a total area of 2.25 $\text{mm}^2$.

A speed of 140 MHz was a reasonable speed to achieve since we did not have a large number of registers, combinational blocks, or other high-delay items. The size is not easily obtained since the core of the chip required more components than originally expected. However, after including the pads, the total area of the chip hits the target.

# 3 Final Design

## 3.1 Design Architecture

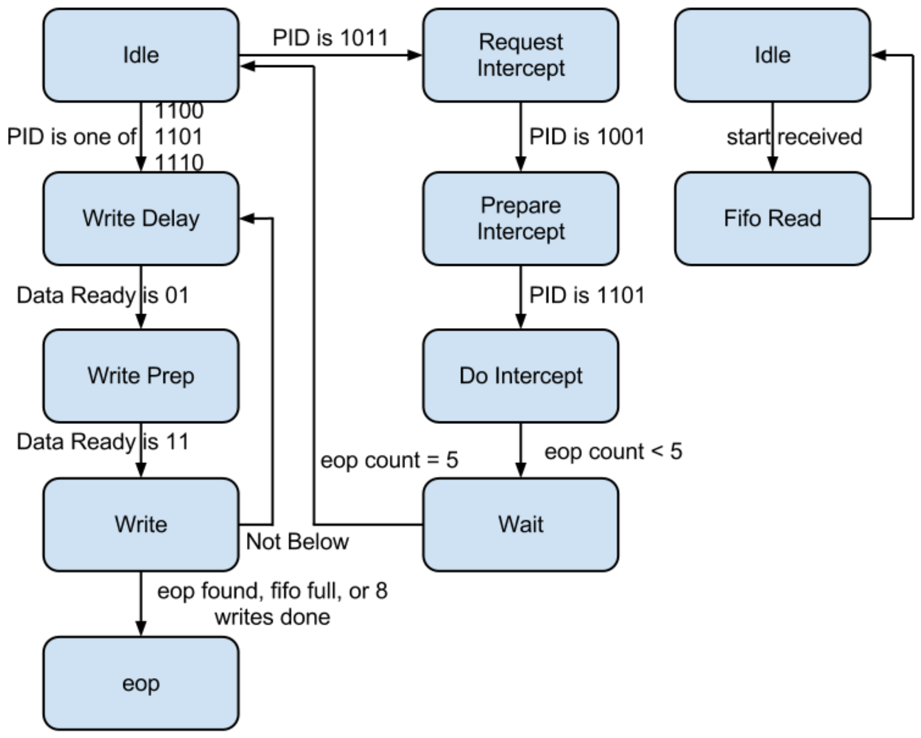
Figure 3.1: The USB Data Logger's top level design



## 3.2 Functional Block Diagrams

### 3.2.1 Controller

Figure 3.2: The controller's state transition diagram



Here, the controller's state machines are featured. The Controller has two seperate state machines, one for controlling the USB read and one for controlling the microSD write. The USB data state machine is primarily a two-part machine: intercept, and read data. If the appropriate PID is found, it will tell the interceptor to do it's job, at which point the device descriptor packet will be appropriately modified. However, if a different set of PIDs is found, it will wait for the decoder to get a full byte and output it to the FIFO, as we are receiving data packets. It ignores all other PIDs, as we won't need to intercept or record in those cases.

### 3.2.2 Decoder

Figure 3.3: The decoder's block diagram

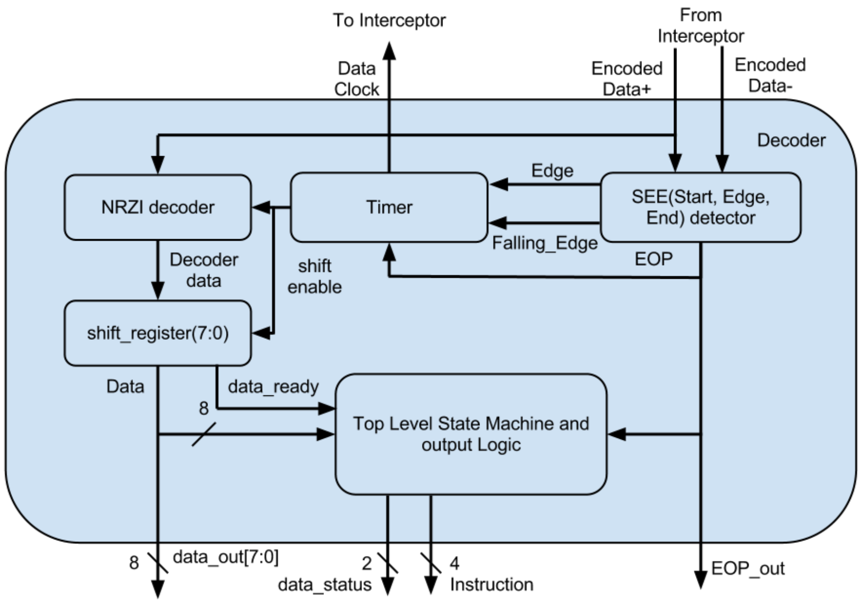
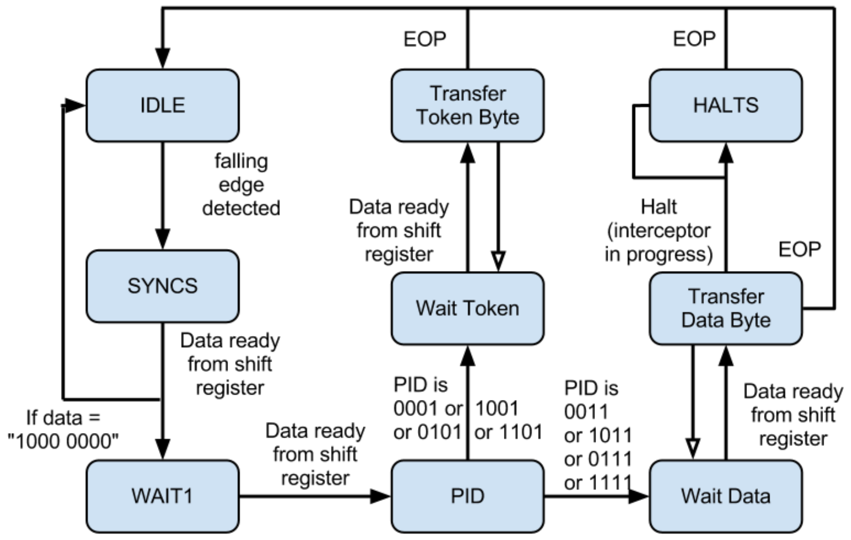


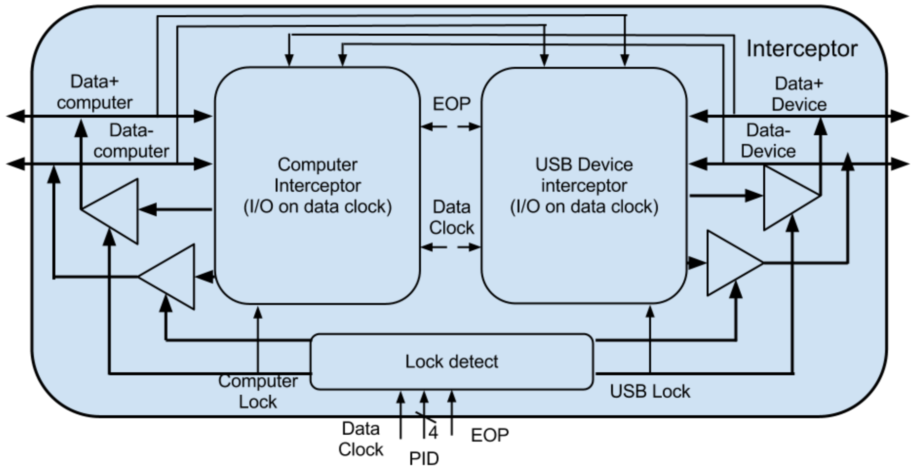
Figure 3.4: The decoder's state transition diagram


IDLE
falling edge detected
SYNCS
Data ready from shift register
If data = "1000 0000"
WAIT1
Data ready from shift register
PID
EOP
Transfer Token Byte
Data ready from shift register
Wait Token
PID is 0001 or 0101
1001 or 1101
PID is 0011 or 1011 or 0111 or 1111
EOP
HALTS
Halt (interceptor in progress)
EOP
Transfer Data Byte
Data ready from shift register
Wait Data

As you can see in figures 3.3 and 3.4, the decoder is primarily a state machine. The decoder also contains the timer, which is our data clock for USB, along with the SEE detector, which lets the other blocks know when an end of packet is on the line. The NRZI decoder recognizes and decodes the USB interface's non-return to zero, inverted signal and shifts in the appropriate bit to an internal shift register. When a full byte is loaded into the register, it outputs to the FIFO.

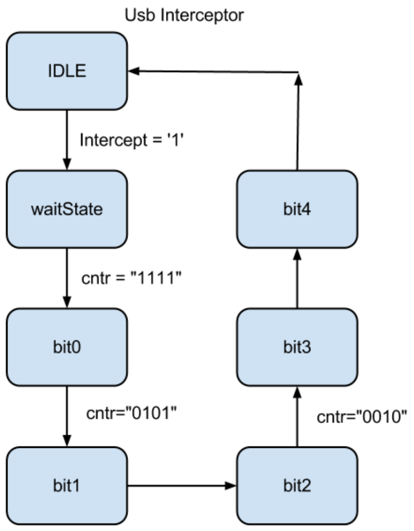
### 3.2.3 Interceptor

Figure 3.5: The interceptor's block diagram



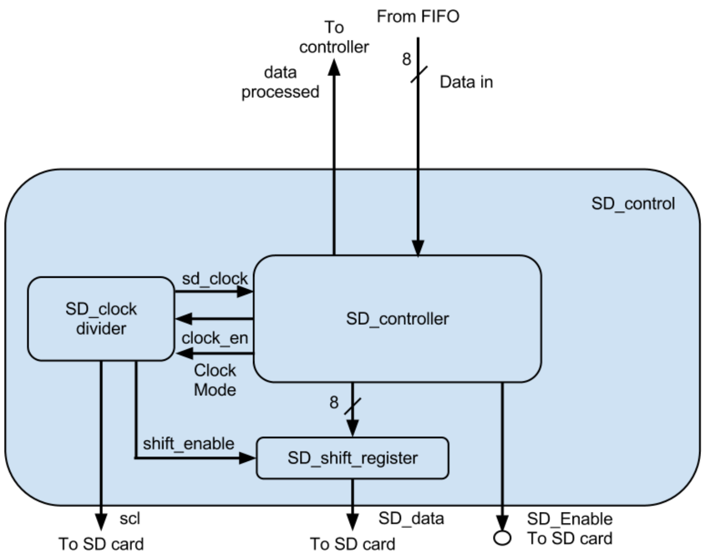
In figures 3.5 and 3.6, we see the interceptor's block diagram and it's state machine. Again, this block is primarily a state machine. Normally, this block will simply be passing data through from one device to the other, with a slight delay. The "Lock detect" sub-block simply uses the PID from the token sent by the computer to decide which way data will be coming from. The "Computer Interceptor" sub-block simply passes through data, while the "USB Device Interceptor" sub-block actually does the intercepting, once it receives an intercept signal from the controller.

Figure 3.6: The USB interceptor's state transition diagram



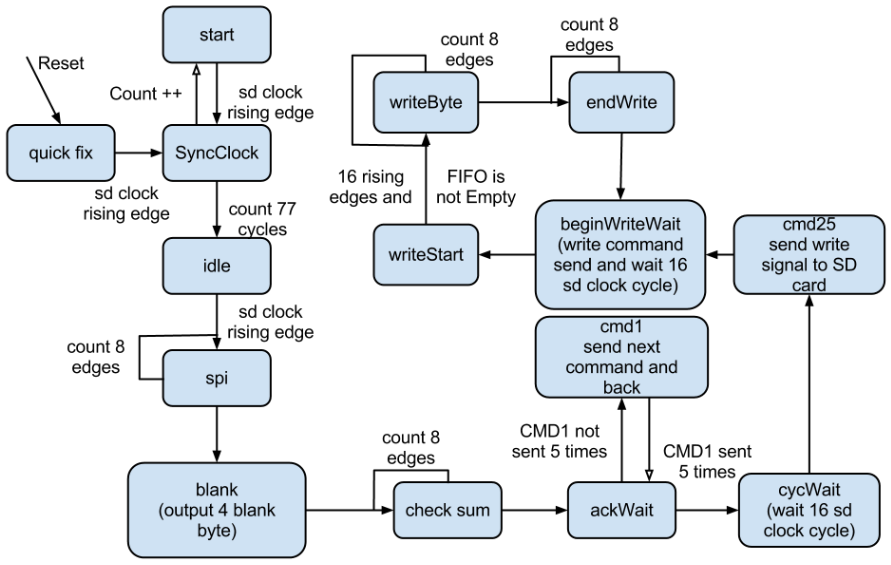
### 3.2.4 microSD Controller

Figure 3.7: The microSD controller's block diagram



Above, the SD controller block is shown. This block is relatively simple, despite what the state machine may have you believe. All this block does is initalize the SD card for writing, followed by writing data from the FIFO to the sd card. When the FIFO is empty, it will simply sit and wait for the FIFO to become non-empty, when it will begin it's outputting once more. It will continue in this loop infinitely.

Figure 3.8: The microSD controller's state transition diagram



## 3.3 Standards and Protocols

The USB Data Logger follows two protocols, USB 1.1 and SD via SPI. USB was chosen as it is currently the most widespread protocol used for communication between external devices and computers. USB 1.1 was specifically chosen due to it's significantly slower speed against USB 2.0 and USB 3.0. The .5 micrometer technology we are using is too slow to reach speeds faster than 400 MHz. We would like to be at least 10x faster than the protocol we are following to allow for sufficient time to perform detection and manipulation of packets. As USB 2.0 is 480 Mbps, which is faster than even our fastest reasonably possible clocking speed, we were unable to use it as our protocol and decided on USB 1.1. [1]

We chose microSD cards as an output device due to their small size and sufficient storage. After doing some research, the team was able to find microSD cards as large as 16 GB, which would be more than enough data to log keystrokes, mouse movements, or other low-data devices for several days, perhaps several weeks. [2] They could be used for some larger-data devices for a day or so. The size is sufficiently small such that, when the full device is

[1] USB Protocol datasheet

[2] microSD datasheet 2

manufactured, it would not need to be much bigger than the data logger ASIC. This allows for inconspicuous placing of the device with a low likelihood of being discovered.

Finally, we chose to communicate with the microSD card using SPI due to the simplicity and low pin count. SPI affords sufficient speed at a maximum of 50 Mbps, significantly faster than our clock and incredibly faster than USB 1.1, while still allowing us to write to the device using 3 lines for one-way communication and a simplified version of the SD protocol, allowing us to ignore CRC bits and checksums.

## 3.4 Timing and Area Budgets

Table 3.1: Timing and Area table

| | **Critical Path** | **Core Area** | **Total Area** |
|---|---|---|---|
| **Success Critera** | 7 ns | Not Applicable | 2.25 mm$^2$ |
| **Budget Assesment** | 4.3 ns<br>Start: Sync Register<br>End: Controller<br>State Register | 0.257 mm$^2$<br>IO Pad Frame<br>Dimensions:<br>870x870 µm$^2$<br>IO Bounded | 1.97 mm$^2$ |
| **Design Compiler** | 1.78 ns<br>Start: SD Control Register<br>End: SD Enable Output | 0.490 mm$^2$ | Not Applicable |
| **Encounter** | 6.628 ns | 0.778 mm$^2$ | 2.25 mm$^2$ |

According to table 3.1 on the previous page, the critical path delays and the area reports from the success criteria, budget assessment, design compiler, and encounter vary to a certain to degree. The reason for this could be caused by a number of factors, which may include changes to the design or the algorithms that encounter and design compiler use to calculate the paths and area.

Starting with the critical path delays, the chosen clock period was 7 ns (140 MHz clock speed) because the design needed to be clocked at least ten times faster than the USB clock. The total area, on the other hand, needed to meet a relatively small target area of 2.25 mm$^2$ because the ASIC device must not be larger than the microSD.

When it came to the budget assessment, the largest combinational path had to be found in order to determine the critical path. It was discovered that the largest path started from a sync register, through combinational logic in the decoder block, then through the next state logic in the controller, and finally ended at the state registers in the controller. Given the timing estimates for 0.5 micron technology, the calculated critical path for the budget was 4.3 ns. The budgeted core area was calculated to be 0.257 mm$^2$, which was found by estimating the number of registers and combinational logic in each block. The budgeted total area was calculated to be 1.97 mm$^2$.

Once the design was working in source simulations, the mapped version of our top-level was created through Design Compiler. The reports after synthesis yielded a 1.78 ns critical path delay that started from SD Control Register, and ended at the SD enable output.The reason why the budgeted path was lower than the Design Compiler critical path might be because the budgeted estimate was generous and the design had been modified while coding and testing the source. It may be possible that a register was coded in between the expected critical path in the budget, thus reducing the delay. The estimated core area was reported to be 0.49 $\text{mm}^2$, which is larger than the budgeted estimate. The reason for this is that the design needed more components than what was originally expected.

A layout was then created in Encounter after the mapped simulations were working. The timing report from Encounter showed that the critical path delay was 6.628 ns, which is higher than the budgeted and Design Compiler estimates. The reason for this change is probably due to how Encounter calculates the paths since more factors are included in the calculation such as wiring and fan in/out delays. Even though 6.628 ns is the largest critical path estimate and probably the most accurate, it still meets the timing requirement of 7 ns clock period. On the other hand, the core area from Encounter reported 0.788 $\text{mm}^2$, which is higher than the budgeted and Design Compiler estimates. In addition, the total area estimate from Encounter was 2.25 $\text{mm}^2$, which precisely matches the maximum target area. The total area, however, is slightly larger than the calculated total area from the budget report, which was 1.97 $\text{mm}^2$. The reason for this might be caused by the number of components that were added to the design. Despite this, the design is able to meet the total target area of 2.25 $\text{mm}^2$.

The critical path delay of our design was able to meet its target of 7 ns with a 6.628 ns delay. The core area, however, was larger than what was originally anticipated even though our total area was able to meet its target. Attempts were made to try and reduce the core area from 0.788 $\text{mm}^2$ down to 0.257 $\text{mm}^2$. The number of registers in the FIFO was one of the major causes of the problem. The design originally had twelve registers in the FIFO, and each register stored eight bits of data, which took up large amounts of area according to Design Compiler. Therefore, the FIFO size was decreased to a reasonable size with a reasonable amount of area. In addition, an effort was made to make the source code smaller and more efficient such as reducing the number of states in a FSM or removing signals that were not needed in the design.

## 4 Testing

The USB Data Logger was primarily tested against the design criteria, so additional tests could be performed to ensure proper functionality. Each primary sub-component of the top-level device was tested using it's own, independent test bench, however most of the sub-components were modified slightly when integrated with the full design. With that said, some of these sub-component test benches no longer work as well as they were originally intended.

The ASIC was designed to handle two different operations that can occur separately or simultaneously. One part of the ASIC is the interceptor, decoder, and overall controller, which handles the USB protocol injection and data extraction. The other half of the ASIC is the SD Controller, FIFO, clock divider, and transmit shift register, which handles the clock, enable, and data signals that go to the micro SD. The test bench had to be made to test both of these processes working separately and at the same time. The test bench for the design utilizes a USB packet record, a NRZI encoding function, and several procedures to accurately simulate the transactions on a USB bus, and to make the design testing process simpler.

The record is composed of the different fields that any packet sent on the bus can have. These fields include synchronize, PID, address, end point, data, and cycle redundancy check (5 and 16 bits depending on the packet). An end of packet field could have been part of the record, however both data lines should be pulled low during an end of packet. So, having an end of packet field did not seem necessary. In addition, test vectors were created to store data for each packet sent on the bus. Each field in the test vector corresponds to the attributes in the record.

The NRZI encoding function was used to encode the testing data bits in the test bench such that the bits on the USB bus would be NRZI encoded in the simulation. The function accepts the decoded data that is meant to be transmitted on the bus, and the value on the data line. If the decoded data is zero then the value on the data line is flipped, otherwise if the decoded data is one then the value on the data line stays the same. This function was simply meant for convience so that the test data would not have to be encoded, and can therefore be easily read and created.

Lastly, there were several procedures used in the test bench that are meant to send different USB packets on the bus. Each procedure uses the NRZI encoding function to encode the test data from the test bench, and if either the device or the host is driving the bus, then the D+ and D- lines of the component that is receiving the bits is pulled high and low, respectively. This is was done because tristates on a bi-directional bus require either pull-up or pull-down resistors, so that the bus never goes into a high impedance state. The first procedure simulates the token packet that is sent from the host. It is flexible enough to support any kind of token (Setup, In, Out, SOF), which is determined by the test vectors. This procedure sends the sync byte first followed by the PID, address, end point, crc5, and then the end of packet. After the end of packet is transmitted, the bus will go idle. The next two procedures simulate data transmissions from the device or the host. Both of these procedures send the sync byte first followed by the PID, data, crc16, and then transitions from the end of packet back to idle. The last two procedures simulate the ack from the device or the host. Both of these procedures send the sync byte first followed by the PID, and then transitions from the end of packet back to idle. The last procedure that was created simulates the transmission check packet. This packet is only used after the device description has been sent from the device. If the transaction was successful then the host will send a zero length packet, which basically means there is no data field. So, this procedure is similar to the data procedure mentioned earlier, however there the data field is left out.

After creating the record, NRZI function, and procedures, the commands to run the test bench lie in the main process. This process was created to send packets using only one line of code with the help of the procedures. So, if the host wanted to know the device description of the device, then the test bench would be able to simulate those transactions in only a few lines of code. The necessary procedures would be called upon to send the appropriate packets.

The full design was tested using a small series of packets over a simulated USB data line. Among the seven tests, five of them were standard data packets to ensure proper reading and writing, with the remaining two used to ensure proper intercepting of USB 3.0 and USB 2.0 device descriptors. All seven tests were deemed successful. Each test case is listed below:

1. Intercept a USB 3.0 descriptor and replace with USB 1.1
2. Intercept a USB 2.0 descriptor and replace with USB 1.1
3. Data "0xdeadbeeffeedbeef" is sent from the device to the computer
4. Data "0xaaabbbcccdddeeef" is sent from the device to the computer
5. Data "0xfff9998887776665" is sent from the device to the computer
6. Data "0x1010101010f00730" is sent from the device to the computer
7. Data "0x1337933244881122" is sent from the computer to the device

The first test case sends several packets between the host and the device, which can ultimately be divided into three steps and each step can be divided into three packets. The first step is for the host to request the USB description of the device. This step starts with the host sending the setup token, then the request description packet is sent, and then it ends with an ack fro the device. The second step is when the USB sends the device description, which starts with the host sending an in token, then the USB 3.0 device description is sent from the USB, and then it ends with an ack from the host. The third and final step is the transaction check, which starts with the host sending an out token, then the host sends a zero length data packet, and then it ends with an ack from the device. The second test case is very similar to this.

Test cases three through six are data packets that are sent from the device to the host. The first packet is the in token which is sent from the host, then the device sends data to the host, and then it ends with an ack from the host. The last test case works in a similar way, however data is now being sent from the host to the device. The first packet is the out token which is sent from the host, then the host sends data to the device, and then it ends with an ack from the device.

# 5 Layout

Figure 5.1: The layout of the USB Data Logger, as generated by Virtuoso and Encounter, respectively

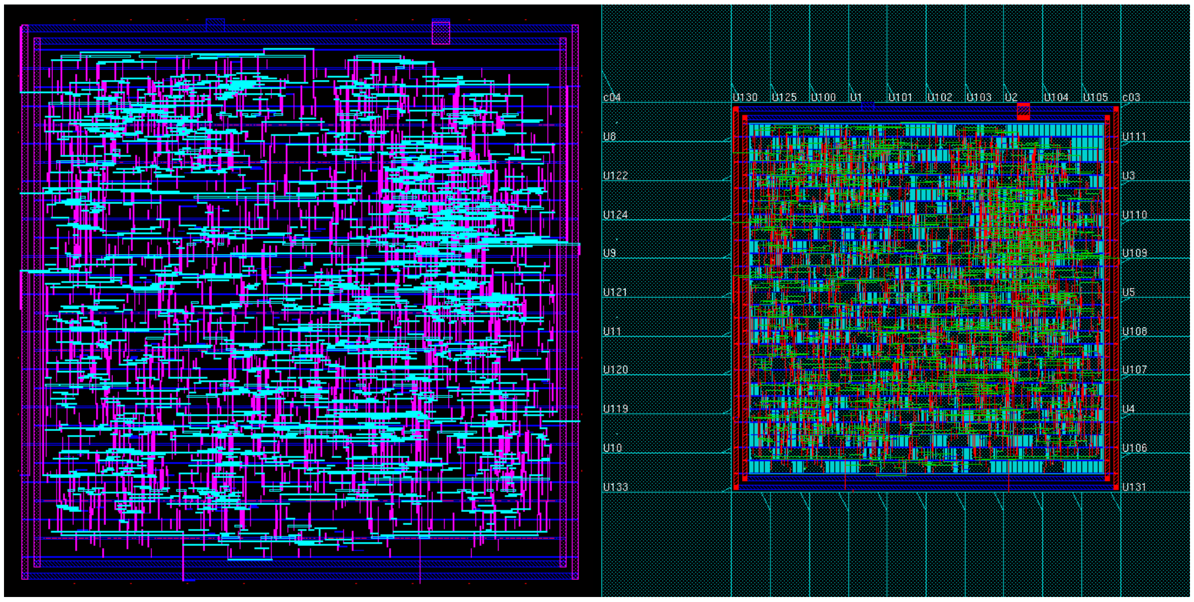


Table 5.1: Layout information provided by Encounter and Virtuoso

| | |
|---|---|
| **Aspect Ratio** | 1.0 |
| **Row Utilization** | 1.8 |
| **Surrounding Area around Core** (µm) | 40 |
| **Core Dimensions**(µm) | 890 x 885 |
| **Core Area**(mm$^2$) | 0.787 |
| **Total Dimensions**(µm) | 1500 x 1500 |
| **Total Area** (mm$^2$) | 2.25 |
| **Number of Gates** | 2337 |

# 6 Results

## 6.1 Fixed Success Critera

1. Test benches exist for all top level components and the entire design. The test benches for the entire design can be demonstrated or documented to cover all of the functional requirements given in the design specific success criteria. **Result: Success**

2. Entire design synthesizes completely, without any inferred latches, timing arcs, and, sensitivity list warnings. **Result: Success**

3. Source and mapped version of the complete design behave the same for all test cases. The mapped version simulates without timing errors except at time zero.
**Result: Success**

4. A complete IC layout is produced that passes all geometry and connectivity checks.
**Result: Success**

5. The entire design complies with targets for area, pin count, throughput (if applicable), and clock rate. **Result: Success**

## 6.2 Design Specific Success Critera

1. Demonstrate by simulation of VHDL test benches that the complete design is able to correctly write data to the external SD card by attaching the SPI Master from the chip to an SPI Slave. **Result: Success**

2. Demonstrate by simulation of VHDL test benches that the complete design is able to successfully remain transparent when intercepting data bits from the bus.
**Result: Success**

3. Demonstrate by simulation of VHDL test benches that the complete design is able to correctly recognizes the default USB protocol handshake. **Result: Success**

4. Demonstrate by simulation of VHDL test benches that the complete design is able to successfully modify the default USB protocol handshake into the USB 1.1 protocol and inject it into the USB bus **Result: Success**

## A Data Sheets and Guide to Design Data

**Account and directory where all of the files are located:** mg56/ECE337/Project

| | |
|---|---|
| **Top level structural VHDL code** | source/sniffer_top.vhd |
| **Computer Interceptor sub-component** | source/computerInterceptor.vhd |
| **Overall Controller** | source/controller.vhd |
| **Top Level Decoder** | source/decoder.vhd |
| **Fifo Ram sub-component** | source/FifoRam.vhd |
| **Fifo Read Pointer sub-component** | source/FifoRead.vhd |
| **Top Level Fifo** | source/FifoTop.vhd |
| **Fifo Write Pointer sub-component** | source/FifoWrite.vhd |
| **Top Level Interceptor** | source/interceptor.vhd |
| **Locking Detector sub-component** | source/lockingDetector.vhd |
| **NRZI decoder sub-component -** | source/NRZIdecode.vhd |
| **SD Controller** | source/sd_control.vhd |
| **Edge detector sub-component** | source/SEE_det.vhd |
| **Shift Register sub-component** | source/shift_greg.vhd |
| **Spi Clock Divider** | source/SpiClkDivide.vhd |
| **Spi Shift Register** | source/SpiXmitSR.vhd |
| **Timer sub-component** | source/timer.vhd |
| **Tristate sub-component** | source/tristate.vhd |
| **USB Interceptor sub-component** | source/usbInterceptor.vhd |

**Test Benches**

| | |
|---|---|
| **Sniffer Top Level test bench** | source/tb_sniffer_top.vhd |
| **Computer Interceptor test bench** | source/tb_computerInterceptor.vhd |
| **Overall Controller test bench** | source/tb_controller.vhd |
| **Fifo test bench** | source/tb_FifoTop.vhd |
| **Interceptor test bench** | source/tb_interceptor.vhd |
| **SD Controller test bench** | source/tb_sd_control.vhd |
| **Spi Clock Divide test bench** | source/tb_SpiClkDivide.vhd |
| **Spi Transmit Shift Register test bench** | source/tb_SpiXmitSR.vhd |
| **Tristate test bench** | source/tb_tristate.vhd |
| **USB Interceptor test bench** | source/tb_usbInterceptor.vhd |

**Report Files**

| | |
|---|---|
| **Sniffer Top Level Design Compiler Report** | reports/sniffer_top.rep |
| **Sniffer Top Level Encounter Report** | timingReports/sniffer_top_postRoute_all.tarpt |

**Scripts**

| | |
|---|---|
| **Sniffer Top Level Script** | scripts/sniffer_top.fcr |

**Presentation**

| | |
|---|---|
| **ECE 337 Final Presentation** | docs/337FinalPresentation.pdf |

**Final Report**

| | |
|---|---|
| **Final Report Document** | docs/finalReport/finalReport.pdf |
| **Final Report LaTeX File** | docs/finalReport/finalReport.tex |
| **Title Page LaTeX File** | docs/finalReport/titlePage.tex |
| **Appendix A LaTeX File** | docs/finalReport/appendixa.tex |
| **Table of Contents** | docs/finalReport/finalReport.toc |
| **LaTeX Log File** | docs/finalReport/finalReport.log |
| **LaTeX Aux File** | docs/finalReport/finalReport.aux |
| **Controller Transition Diagram** | docs/finalReport/controllerTrans.png |
| **Decoder Block Diagram** | docs/finalReport/decoderBlock.png |
| **Decoder Transition Diagram** | docs/finalReport/decoderTrans.png |
| **Encounter-generated Layout** | docs/finalReport/encounter.png |
| **Interceptor Block Diagram** | docs/finalReport/interceptorBlock.png |
| **Interceptor Transition Diagram** | docs/finalReport/interceptorTrans.png |
| **Operational Timeline** | docs/finalReport/OperationalTimeline.png |
| **microSD Output Testbench Diagram** | docs/finalReport/outputToMicroSD.png |
| **microSD Controller Block Diagram** | docs/finalReport/sdBlock.png |
| **microSD Controller Transition Diagram** | docs/finalReport/sdTrans.png |
| **Token Recognition Testbench Diagram** | docs/finalReport/tokenRecognition.png |
| **Design Architecture Diagram** | docs/finalReport/topLevel.png |
| **Transparency Testbench Diagram** | docs/finalReport/transparency.png |
| **System Usage Diagram** | docs/finalReport/usageDiagram.png |

|  |  |
|---|---|
| **USB 2.0 to USB 1.1 Testbench Diagram** | docs/finalReport/usb2.png |
| **USB 3.0 to USB 1.1 Testbench Diagram** | docs/finalReport/usb3.png |
| **Virtuoso and Encounter layout** | docs/finalReport/virtuoso.png |

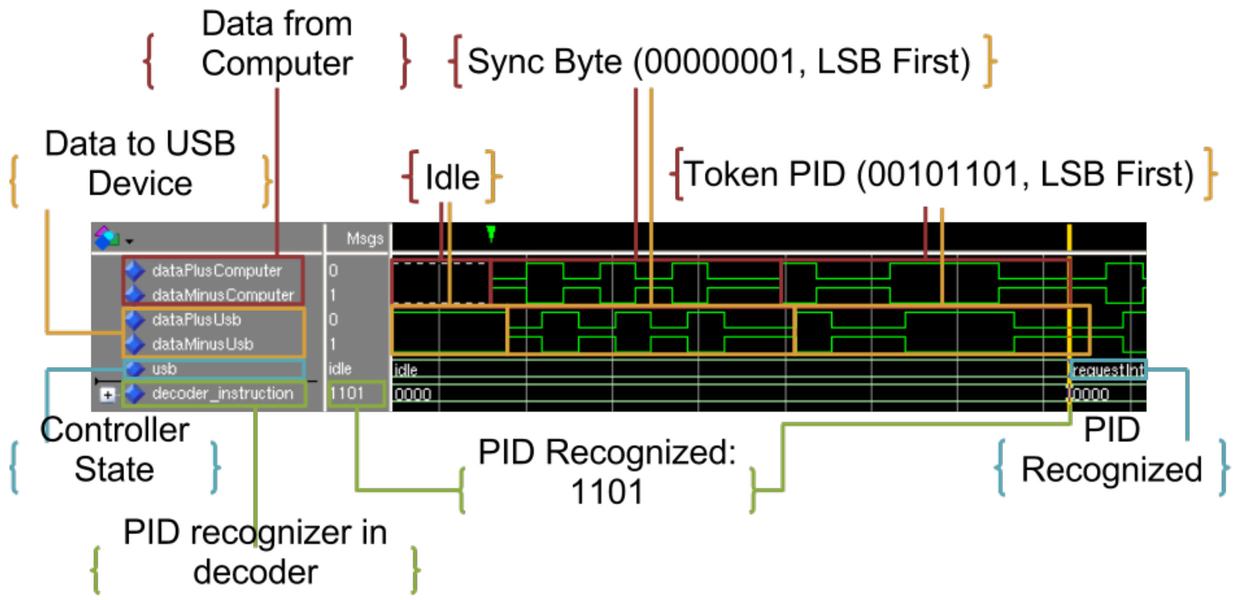
**Datasheets**

|  |  |
|---|---|
| **ECE 337 USB Lab Manual** | doc/lab6_sp09.pdf |
| **ECE 337 USB Slides** | doc/Lab6_USB.pdf |
| **microSD datasheet 1** | doc/Lattice_microSD_specs.pdf |
| **microSD datasheet 2** | doc/sandisk_secure_digital_card.pdf |
| **SPI Protocol datasheet** | doc/SPI_Protocol.pdf |
| **USB Protocol datasheet** | doc/USBProtocol.pdf |

# B Simulation Results

## B.1 Token Identification

Figure B.1: Testbench showing identification of a USB Data Descriptor Token

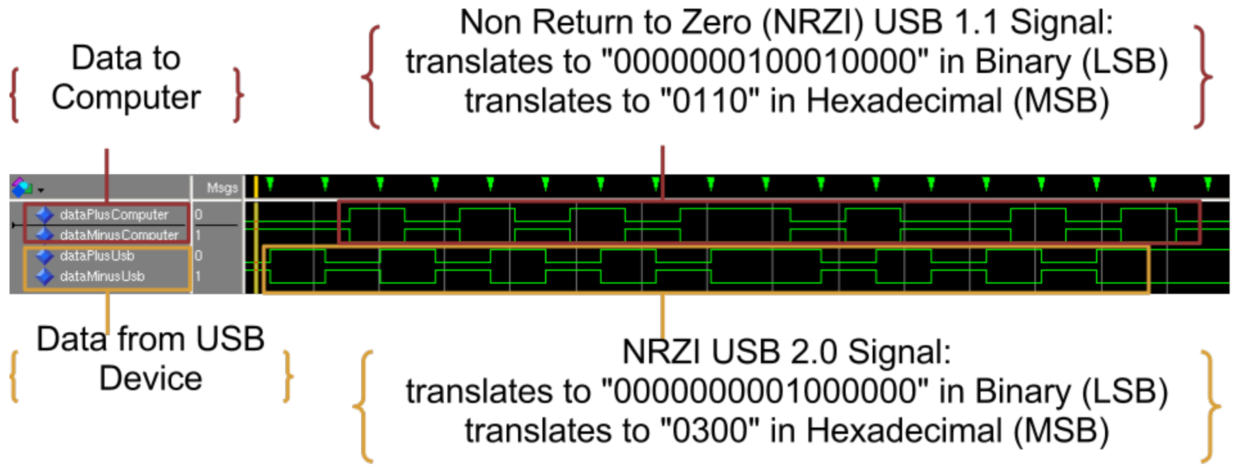


In figure B.1, the depicted testbench shows more than just the output signals. This must be done in order to show proper identification of a USB Data Descriptor Token. The top four lines are the data lines, identified using red and yellow, and are effectively identical for this test. After coming off of idle, the lines send a sync, followed by a PID, and ultimately followed by the appropriate data. You can see in the green decoder_instruction line when the PID is recognized by the decoder, followed by when it is recognized by the controller on the blue USB line.

## B.2 Data Descriptor Modifications

### B.2.1 USB 2.0 to USB 1.1

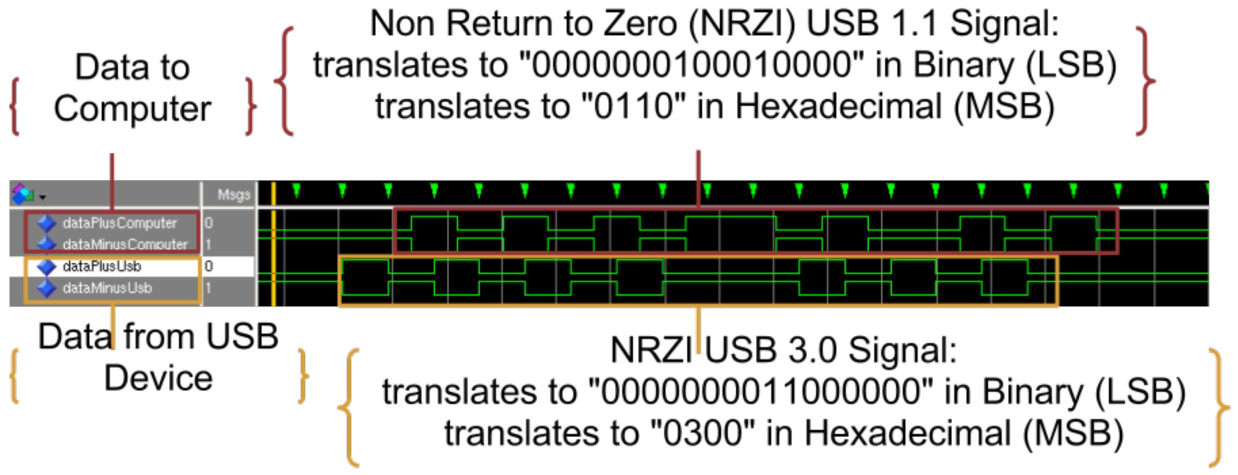
Figure B.2: USB 2.0 Data Descriptor Modification Testbench


Data to Computer
Non Return to Zero (NRZI) USB 1.1 Signal:
translates to "0000000100010000" in Binary (LSB)
translates to "0110" in Hexadecimal (MSB)
Msgs
dataPlusComputer 0
dataMinusComputer 1
dataPlusUsb 0
dataMinusUsb 1
Data from USB Device
NRZI USB 2.0 Signal:
translates to "0000000001000000" in Binary (LSB)
translates to "0300" in Hexadecimal (MSB)

In figure B.2, we see the Data Descriptor being modified from USB 2.0 to USB 1.1. As you can see in the lower pair of lines, the data coming from the USB device is USB 2.0, 0x0200 (remember it is read least significant bit first). However, in the upper pair of lines, the data going to the USB device is USB 1.1, 0x0110.

### B.2.2 USB 3.0 to USB 1.1

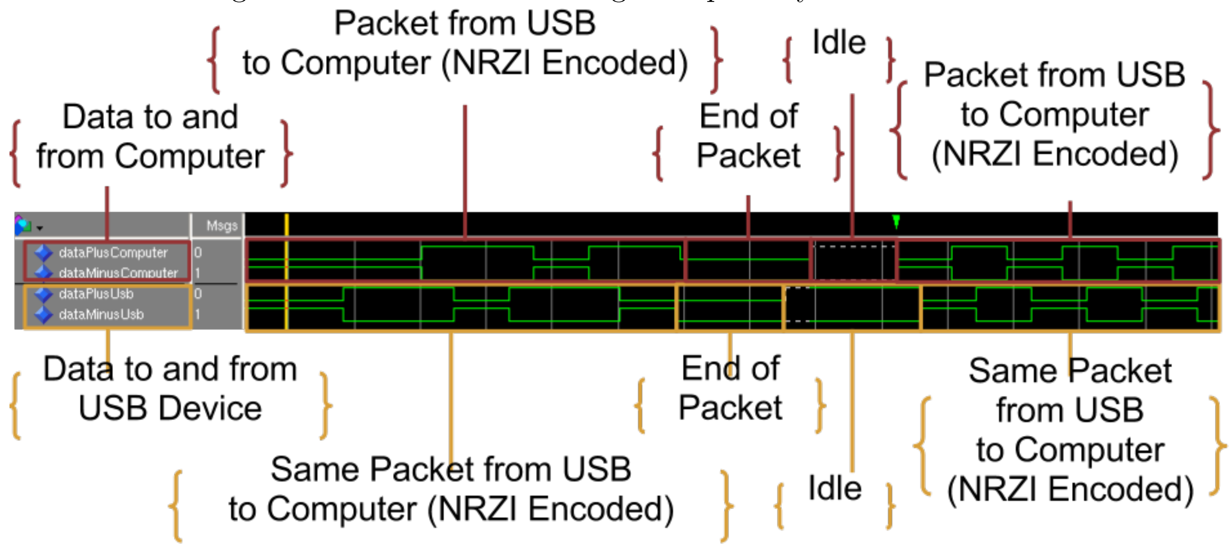
Figure B.3: USB 3.0 Data Descriptor Modification Testbench


Data to Computer
Non Return to Zero (NRZI) USB 1.1 Signal: translates to "0000000100010000" in Binary (LSB) translates to "0110" in Hexadecimal (MSB)
Msgs
dataPlusComputer 0
dataMinusComputer 1
dataPlusUsb 0
dataMinusUsb 1
Data from USB Device
NRZI USB 3.0 Signal: translates to "0000000011000000" in Binary (LSB) translates to "0300" in Hexadecimal (MSB)

In figure B.3, we see a nearly identical test case to figure B.1, except that the modifcation is from USB 3.0 to USB 1.1. Notice that in the 2.0 test bench the NRZI encoding is reversed between lines until the end of packet, but in the 3.0 test bench this is not the case. That is because the binary reprsentations of the USB 3.0 and USB 1.1 descriptors (0x0300 and 0x0110 in hexadecimal) both have two ones, while the USB 2.0 descriptor (0x0200) only has one. After modifying this bit, we can't simply "follow" the bus directly anymore, but rather we have to look for changes between the lines and make our output match.

## B.3 Transparency

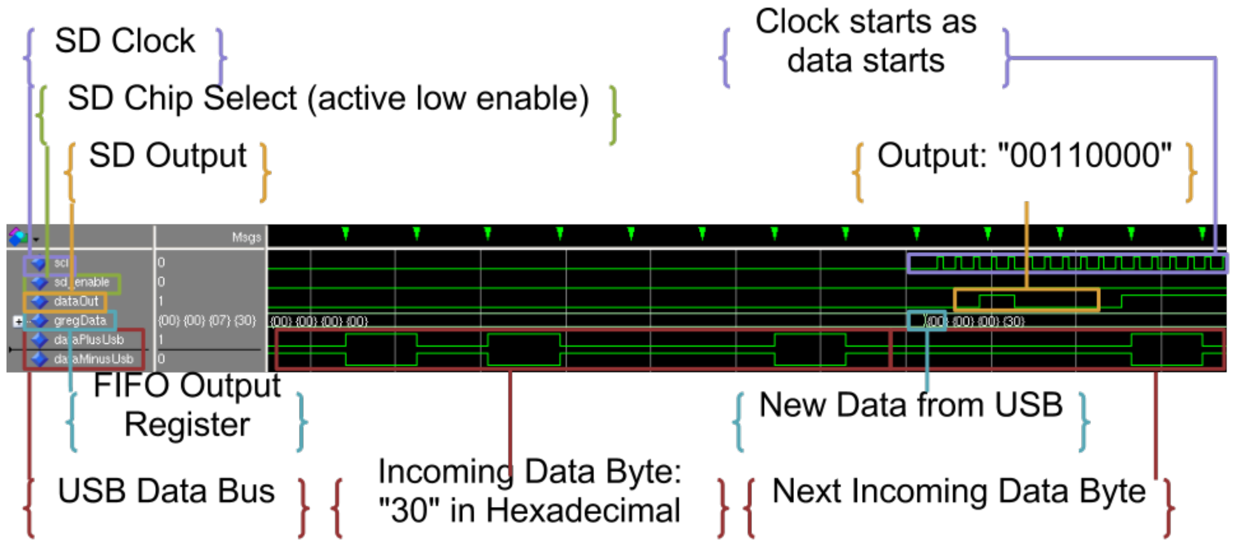
Figure B.4: Testbench showing transparency between devices


Packet from USB to Computer (NRZI Encoded)
Idle
Data to and from Computer
End of Packet
Packet from USB to Computer (NRZI Encoded)
Msgs
dataPlusComputer 0
dataMinusComputer 1
dataPlusUsb 0
dataMinusUsb 1
Data to and from USB Device
End of Packet
Same Packet from USB to Computer (NRZI Encoded)
Same Packet from USB to Computer (NRZI Encoded)
Idle

In figure B.4, we can see that data coming in from the USB device is identical to data going out to the computer before the end of packet. After the idle state, we can see the data coming in from the computer is identical to the data going out to the USB device.

## B.4 microSD Output

Figure B.5: microSD card output



In figure B.5, we can see the input from the USB device is 0x30. We see that this changes on the FIFO in the blue-labeled line, directly above it. When this happens, the SD Clock starts ticking, and it outputs the data from the device to the SD card. It continues to tick after the fact as it has to write a few commands to the SD card before it can continue.