Lab 6: USB Peripheral Receiver

In this lab you will:

- Design and test (using individual Test Benches) each of the following functional units for the USB Peripheral Receiver: DECODE, EDGE_DETECT, TIMER, SHIFT_REG, EOP_DETECT, RCU.
- Combine all of the blocks to create the Receiver (USB_RCVR).
- Generate a test bench to demonstrate the functionality of the completed receiver.
- Synthesize the receiver using Synopsis.
- Test the synthesized version of the receiver.

1. Copying Setup Files

In a UNIX terminal window, issue the following command:

```
setup6
```

This command is actually an alias to a batch/script file in the ee495d/Class directory that will copy the files that are necessary for the completion of this lab. **If you have trouble with this step, please ask for assistance from your TA.**

2. Expectations for Lab 6 (and beyond)

From the first day of class, you have been gathering the knowledge and expertise with the tools to allow you to complete this design. At this point in the course, you should know how to operate all the tools that were introduced to you in Labs 1 through 5. In addition, up to this point, you have been introduced to all the VHDL syntax and constructs that are needed to implement the USB Peripheral Receiver. That is to say that with the knowledge of VHDL that you have you should be able to complete this design.

This lab is structured to mimic what you would encounter should you choose to pursue a career as an ASIC/VLSI designer upon graduation. Essentially you (the designer) are being provided with a set of specifications by your supervisor or lead engineer for each block used in the design. You are not being instructed on how to design these blocks. You are only told the inputs to each block, the outputs from each block and what function the block is to perform. It is up to you to come up with a working solution for each block and then integrate them all to form the USB Peripheral Receiver. In practice, even this may be more information than you would be given. More likely, you would be just given a top level specification of the inputs, outputs, and functions to be performed.

Grading Policy, Deadlines, and Turnin Commands:

For Lab 6, there are three deadlines:

PHASE I: Code, test benches, and diagram due at the beginning of lab during the week of Feb. 24, 2009. Test bench demonstrations will be performed during lab.

For Phase I, you will need complete code for the Decode, Edge Detection, and Shift Register blocks as well as partially complete code for the RCU and the Timer block. You will also need to create test benches for the Decode, Edge Detection, Shift Register, and FIFO blocks, which you will electronically submit using '**submit Lab6a**', and demonstrate during lab.

You will also need to draw a state transition diagram for the RCU for the USB receiver.

PHASE II: Test bench submission and source code grading by 11:59 PM, March 1, 2, and 3rd, depending on your section.

You will have three chances to test your source code against the grading script without penalty. However, your credit on this section is entirely dependent on the completeness of the test bench you submit while testing your source code. Course staff will be looking for test cases which can be inferred from the design specifications.

For Phase II, you will use '**submit Lab6b**' for both your source code testing and for your test bench.

PHASE III: The mapped version of your design must be working and will be graded based on the submission score for your mapped design. This is due by the beginning of your lab during the week of March 3rd. The command for Phase III is '**submit Lab6c**'.

You will get 3 chances to use the automated grading script, and your grade for this phase will be based solely on your mapped score. Your score will only be reported if you earn at least 50% on the automated grading script.

For your lab to be considered complete (to pass the course) your mapped version must be working sufficiently well to pass 50% of the automated tests.

If at any time you would like to check the last set of graded results, you can use '**check LabX**', where LabX corresponds to a submission phase/lab.

COMMENTS:

- You are to work on this lab on your own. You are not to share your test benches nor your code with anyone else.
- The code for the test bench used by the grading script will not be disclosed to the student nor will the test cases be told to the student.
- The majority of the points come from successfully passing the synthesized design test bench; therefore it is highly recommended to make sure you have an error free run through Synopsys.
- You will be allowed a total maximum of 6 passes through the Lab 6 TA test bench: 3 for Phase II and 3 for Phase III.
- **For the top-level block and seven sub-blocks of code required for this lab you must name the source file with the name provided, and use the entity, signal names and signal types provided.** Failure to do this will cause you to fail the grading script and it will be counted as one attempt.
- **Make sure that your USB_RCVR.scr file is located in your scripts directory.** Failure to do this will cause you to fail the grading script and it will be counted as one attempt.

3. Introduction to USB

The Universal Serial Bus (USB) is a modern serial interface designed for high-speed and easy-to-use communications between personal computers and peripheral devices. The system is comprised of a single host, generally the PC, and up to 127 peripheral devices. USB offers many advantages over serial ports, including higher speed, the ability to share a bus with many devices, and the ability to supply power to the end devices.

The goal of this project is to design the receiver portion of a USB bus interface for a peripheral device. The design has been simplified in that much of the logic that would normally be performed in hardware has been moved into software, but the finished receiver remains essentially compliant[1] with the core USB 1.1 specifications for full-speed devices (http://www.usb.org/developers/docs/usbspec.zip).

4. Protocol Description

Data Rate

USB is an asynchronous bus, meaning that the data has no separate clock line, the clock must be reconstructed from the incoming data. The USB you are going to design and build will have a data rate of 1.5 Mbits/second, ±0.25% (actually, the full-speed data rate for USB is 12.000 Mbits/second, ±0.25% and USB 2.0 allows data rates of 480Mb/s, but these will not be covered in this project).

Like many other high-speed serial busses, USB uses differential signals, so that the data is sent over two separate wires. The D+ line can be seen as the 'real' data while the D- line contains the inverse of the D+ line, except at the end of a packet as explained below.

[1] The design to be implemented in Lab 6 has been simplified by removing bit-stuffing from the protocol. The purpose of the NRZI encoding, inverted or not (explained in Section 4), is to provide frequent transitions on the bus so that the clocks can be synchronized. However, if a long string of 1s is to be transmitted on the bus, the output would remain in the same state, not providing the necessary transitions to synchronize the clock. For this reason, the encoding also uses a technique called "bit stuffing" to provide the necessary transitions. Each time six consecutive ones are encountered in the input data stream, a zero is inserted before the data is NRZI encoded. The receiver will then ignore the incoming bit following any six consecutive ones in the decoded data. Figure 2.2 shows an example of the incoming encoded data, the decoded data with the stuffed bit highlighted, and the decoded data with the stuffed bit removed. **You are not required to handle bit stuffing for your design. In fact, you should not do so since the grading script is designed on the assumption that you are not handling bit stuffing.**

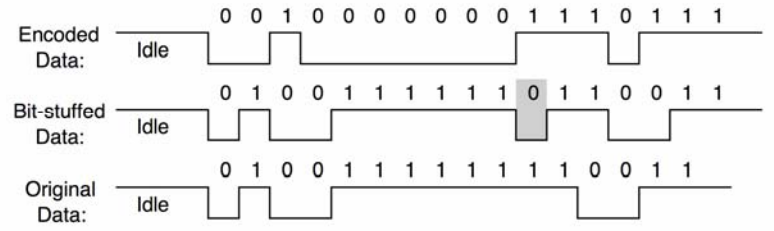

Figure 2.2. Inverted NRZI encoded and bit-stuffed data.

The primary reason for using differential signals is to provide noise cancellation within the USB cables.

Data Encoding

Data is sent over USB with the least-significant bit of a byte coming first, and the most-significant bit coming last. Most signals on USB utilize a type of encoding known as NRZI (Non Return to Zero, Inverted) encoding. However, for our USB design we are going to use the "Inverted NRZI" encoding. In Inverted NRZI encoding (on the transmitter), a "0" is represented as a change in the output level (from a 0 to a 1 or from a 1 to a 0), while a "1" is represented with no change in state.

When receiving data, as in this project, the Inverted NRZI-encoded data must then be decoded. When decoding data, a change in state on the input is translated to a "0" in the original data, while remaining in the same state will be seen as a "1" in the original data. Figure 2.1 shows data before and after the Inverted NRZI decoding.

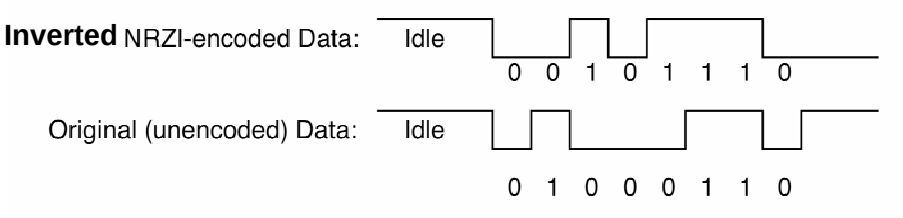

Figure 2.1. NRZI decoded data.

Packets

USB specifies several types of packets that can be transmitted and received; some of these packets have a fixed length while others are variable. In this project, the body of the packet will be handled by the software controlling the USB peripheral, our design need only find the beginning and end of each packet on the bus.

When in the idle state, a USB bus will have a logic high on the D+ line and a logic low on the D- line. Each packet then starts with a SYNC field, defined as a "10000000" in binary before Inverted NRZI encoding. After encoding, and remembering that data is sent with the least-significant bit first, the SYNC field will be received (on the D+ line) as shown in Figure 2.3.

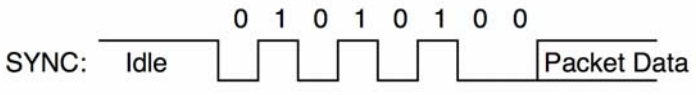

Figure 2.3. USB SYNC byte.

The receiver will begin capturing and storing data beginning with the first byte after a SYNC byte. The reading will continue until an EOP (End Of Packet) is received. The EOP is defined as both the D+ and the D- lines going to a logic low state (they are no longer differential) and lasts for at least 1 bit length. Figure 2.4 shows the EOP signal, where D+ is shown as a black line and D- is shown as a dashed, blue line.

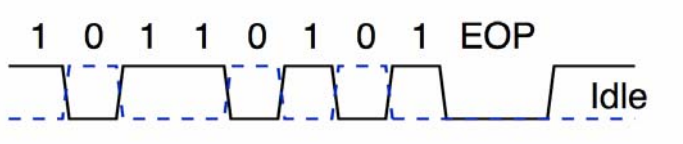

Figure 2.4. USB EOP signal.

5. Hardware Design

An overall block diagram for the design is shown in figure 3.1. Note that the connections are not shown for the CLK and RST lines. In addition, synchronizers should be inserted at the D_PLUS and D_MINUS inputs. The top-level block for this project must be named USB_RCVR.

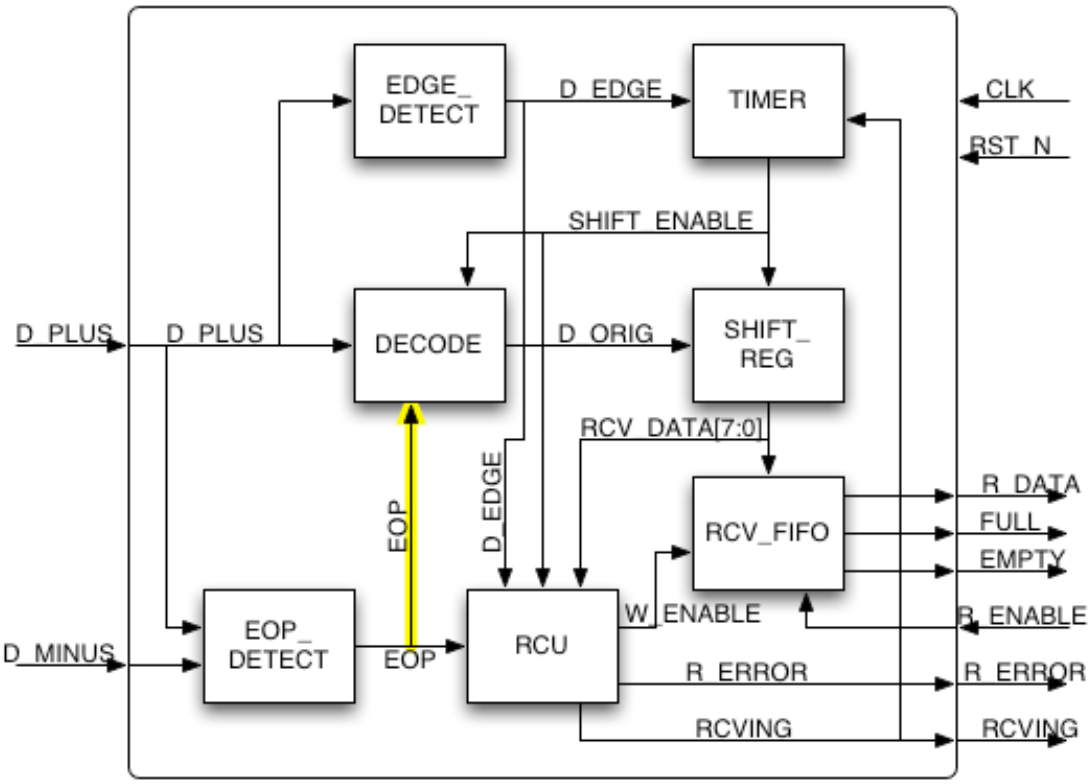

Figure 3.1. Top-level block for USB_RCVR design.

You will need to design and test each of the blocks above, as well as the top-level block (with the exception of the RCV_FIFO, which will be provided). Each block is described in the following section, including the block name, basic functionality, and the input & output signals of the block.

DECODE

The DECODE block is responsible for removing the Inverted NRZI encoding described in Section 2 [of lab 6] from the incoming data. The block will store the incoming bit present on the rising edge of the CLK line when SHIFT_ENABLE is high, then output a 1 as long as the current input and the stored value match, or a 0 if the current input and the stored input differ. The block should synchronously reset to an idle-line state when an End-Of-Packet is detected. This block is much simpler than it sounds.

The code for this block must contain one process for the state register, dataflow code to generate the next state, and dataflow code to generate the output. Also, be careful not to combine the CLK'event and the SHIFT_ENABLE signal into an if-statement, as this would result in a gated clock. The SHIFT_ENABLE signal should only be used in the dataflow line that generates the next state.

Signal	**Direction**	**Description**
CLK	Input	The system clock (12 MHz), will be used to latch the data into the stored bit when SHIFT_ENABLE is high.
RST_N	Input	This is an *asynchronous*, active-low system reset. When this line is low, the D_ORIG output will immediately go to the idle-line state.
D_PLUS	Input	The incoming Inverted NRZI-encoded data.
SHIFT_ENABLE	Input	This signal enables the bit on D_PLUS to replace the stored bit on the rising edge of CLK.
EOP	Input	This is a *synchronous* reset signal that causes the stored bit to return to the idle-line state.
D_ORIG	Output	The decoded data.

EDGE_DETECT

The EDGE_DETECT block is very similar in operation to the DECODE block, except for the lack of the SHIFT_ENABLE. The EDGE_DETECT block will save the bit on the D_PLUS line at the rising edge of each CLK cycle, then output a 1 when the bit on the D_PLUS line does not match the stored bit, and a 0 when the value on the line matches the stored bit. This edge is used to synchronize the TIMER block.

The code for this block must contain one process for the state register, and dataflow code to generate the next state (if necessary) and the output of the block.

Signal	**Direction**	**Description**
CLK	Input	The system clock (12 MHz), the D_PLUS line is stored on each rising edge.
RST_N	Input	This is an *asynchronous*, active-low system reset. When this line is low, the D_EDGE output will

		immediately go to a 0.
D_PLUS	Input	The incoming Inverted NRZI-encoded data.
D_EDGE	Output	The D_EDGE line will be asserted for 1 cycle of the system clock whenever the value of the D_PLUS line changes state.

TIMER

The TIMER block is used to generate the SHIFT_ENABLE signal that allows the shift registers to shift in the next bit of data. Because the system clock (12 MHz) runs at 8 times the speed of the USB input (1.5 MHz), the data must only be shifted once for every 8 system clock cycles. It is important to capture the data as near as possible to the middle of the incoming bit. Also, since the clock may drift slightly during a long packet, the clock must be constantly resynchronized with the incoming data. This is done by resetting the timer to a known state when the D_EDGE line is high. This is one of the more difficult block to design.

Signal	**Direction**	**Description**
CLK	Input	The system clock (12 MHz), the SHIFT_ENABLE output will only change on the rising edge of the system clock.
RST_N	Input	This is an *asynchronous*, active-low system reset. When this line is low, the counter will immediately be reset and SHIFT_ENABLE will go to a 0.
D_EDGE	Input	The D_EDGE line is asserted for 1 clock cycle when the D_PLUS input changes state. This signal will be used to reset the counter to a known state. This is a *synchronous* reset, and is not necessarily the same reset state as the RST signal.
RCVING	Input	The RCVING line will be asserted by the RCU when the TIMER is to begin counting. When this line is low, the timer should reset to the initial state and the SHIFT_ENABLE should be a 0.
SHIFT_ENABLE	Output	The SHIFT_ENABLE signal will be high for 1 clock cycle, after which the shift registers will shift the next bit. After a reset or when TIMER_ENABLE is low, this line will remain a 0.

SHIFT_REG

The shift register will store 1 byte of incoming values and hold those values while they are read into the FIFO buffer. The USB data is sent least-significant bit first, so the shift register should shift so that the data ultimately has the least-significant (first) bit on the right and the most-significant (last) bit on the left.

Signal	Direction	Description
CLK	Input	The system clock (12 MHz), all shifting will occur on the rising edge of CLK.
RST_N	Input	This is an *asynchronous*, active-low system reset, which causes the shift register to be reset to all 0s.
SHIFT_ENABLE	Input	The shift register will only perform a shift on the rising edge of CLK and when the SHIFT_ENABLE line is asserted.
D_ORIG	Input	This is the decoded input data to be shifted in. This signal is connected to the most-significant bit of the shift register to allow the data to be shifted least-significant bit first.
RCV_DATA[7:0]	Output	The RCV_DATA [7:0] bus contains the value stored in each of the shift registers. This value will be read after each byte is received and stored into the FIFO buffer.

RCV_FIFO

The FIFO (First-In, First-Out) buffer allows the receiver to store data for a period until the microcontroller has a chance to read the data. Note that **you do not need to design the FIFO, a synthesized version has been provided.** You will need to create a RCV_FIFO.vhd file, but your file will only be a "wrapper" for the provided FIFO, mapping the provided FIFO's ports to the port names used in your design. The provided FIFO is in the gold_lib library and has the following entity declaration:

```
component goldFifo
    port (
      RCLK, WCLK, RST_N : in  std_logic;
      RENABLE, WENABLE  : in  std_logic;
      WDATA             : in  std_logic_vector(7 downto 0);
      RDATA             : out std_Logic_vector(7 downto 0);
      EMPTY, FULL       : out std_logic);
  end component;
```

You will likely get error messages regarding the FIFO when running scriptgen, but your top-level design should still be able to run.

The table below specifies the port names to be used in the RCV_FIFO entity as well as some specifics on the operation of the FIFO.

Signal	Direction	Description

CLK	Input	This is the READ clock signal. This clock signal is used by the FIFO to keep track of the current read address. This is also the WRITE clock signal. This clock signal is used by the FIFO to (1) latch data into the FIFO's RAM at its rising edge and when WENABLE is asserted (Logic '1') and to (2) keep track of the current write address.
RST_N	Input	This is the RESET signal. When this signal is asserted low (logic '0'), all registered element's outputs are set to their reset logic value (the entire FIFO contents are cleared).
R_ENABLE	Input	This is the READ enable signal. When this signal is asserted (logic '1'), the READ pointer is incremented at the rising edge of the READ clock. The purpose of this signal is to let the FIFO know that the current data being pointed to by the READ pointer has been read.
W_ENABLE	Input	This is the WRITE enable signal. When this signal is asserted (logic '1'), (1) the data on the WDATA bus is latched into the FIFO's RAM at the rising edge of the WRITE clock and (2) WRITE pointer is incremented at the rising edge of the WRITE clock.
W_DATA[7:0]	Input	This is the WRITE data bus. This bus holds the data to be written to the FIFO's RAM at the rising edge of the WRITE clock and while the WRITE enable signal is held high (logic '1').
R_DATA[7:0]	Output	This is the READ data bus. This bus holds the data to which the READ pointer is currently positioned (the next piece of data to be read from the FIFO's RAM).
EMPTY	Output	This is the EMPTY flag. When this signal is asserted (logic '1'), the FIFO is empty.
FULL	Output	This is the FULL flag. When this signal is asserted (logic '1'), the FIFO is full.

EOP_DETECT

The EOP_DETECT block detects the USB End-Of-Packet signal. As specified in Section 2, this occurs when both the D_PLUS and D_MINUS lines are set to a logic low. This is an extremely simple block.

Signal	Direction	Description
D_PLUS	Input	The positive USB input line.
D_MINUS	Input	The negative USB input line.
EOP	Output	EOP will be asserted *asynchronously* when both D_PLUS and D_MINUS are low.

RCU

The RCU (Receiver Control Unit) is in charge of the operation of the entire receiver. The RCU will do the following:

- On reset, the RCU goes to the idle (waiting) state.
- When an edge is first detected, begin receiving data through the DECODE block to the SHIFT_REG. Set the RCVING line high.
- After the first byte is shifted in, check to see that the byte matches the USB SYNC byte.
    - If the byte does match the SYNC byte, do not store the byte into the FIFO, but begin recieving and storing data from the next byte.
    - If the byte does *not* match the SYNC byte, set the R_ERROR flag to a 1 and disregard any input until the next EOP is reached. The RCVING line should remain high until the EOP is reached. Keep the R_ERROR flag high until the next packet begins.
- Continue recieving until the EOP is detected, then set the RCVING line low.
    - If an EOP is reached with an incomplete byte in the shift register (i.e. the EOP should occur just after a byte is shifed into the FIFO), set the ERROR flag high and do not store the last byte. Leave the R_ERROR flag high until the next packet begins.

Signal	Direction	Description
CLK	Input	This is the system clock (12 MHz), all state transitions will occur on the rising edge of the CLK.
RST_N	Input	This is an *asynchronous*, active-low system reset, which causes the RCU to go to the idle state.
D_EDGE	Input	D_EDGE is asserted by the EDGE_DETECT block whenever the D_PLUS line changes state. This will be used to detect the beginning of a packet from the idle state.
EOP	Input	EOP will be asserted when the USB EOP signal is detected on the input lines. This flag indicates the end of a packet.

SHIFT_ENABLE	Input	This line will be set high for 1 clock cycle when the shifte register is to shift in the next bit from the inout line. This signal will be used to count the number of bits received in order to store the data after an appropriate number of bits.
RCV_DATA[7:0]	Input	This line contains the data in the shift register, and is used to compare the received value to ensure that it matches the SYNC byte.
RCVING	Output	This line will be set high when the circuit is recieving a packet.
W_ENABLE	Output	This line enables the FIFO to read the byte currently in the shift register. It will be set high for 1 clock cycle after each complete byte has been received.
R_ERROR	Output	This flag indicates an error occured while receiving the packet. It will be asserted until the start of the next packet.