ECE 337 Lab 6:
USB Peripheral Receiver Mini Design Project

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 ece337/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 4. In addition, up to this point, you have been introduced to all the Verilog syntax and constructs that are needed to implement the USB Peripheral Receiver. That is to say that with the knowledge of Verilog 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.

3. Grading Policy, Deadlines, and Submit Commands

All modules designed as part of this lab must be exclusively either a purely combinational block or a Moore style sequential design.

3.1. Design Preparation

- State-Transition Diagram is required for the RCU
- Timing/Waveform Diagram is required for the Timer block
- Functional block diagram based on your flex_counter module is required for the Timer block
- RTL Diagrams are required for:
    - Decode block
    - RCU block
- Schematic Diagrams are required for:
    - EOP Detect block
    - Edge Detect block

You must have these diagrams signed off or submitted via the "submit Lab6prep" command no later than 'late night' 2 days prior to your respective phase 1 deadline

- **Tuesday Labs:** 2am 10/5
- **Wednesday Lab:** 2am 10/6
- **Thursday Lab:** 2am 10/7

3.2. Phase 1

Code and test benches are due at the beginning of lab during the second week of Lab 6.

For Phase I, you will need complete code for the Decode, Edge Detection, and EOP Detection. You will also need to create test benches for the Decode, Edge Detection, EOP Detection, and FIFO blocks, which you will electronically submit using "**submit Lab6phase1**".

Your credit on this section is entirely dependent on the completeness of the required blocks and their test benches that you submitted with the "**submit Lab6phase1**" command. **Test benches will be graded based on complete coverage testing; 1/0 toggle, finite-state-machine (FSM) where applicable, statement, branch, and expression coverage will be graded for each test bench.**

3.3. Pre-Phase 2 Test Runs

To encourage early testing of your design, you will have three chances to test your source code against the grading script without penalty, via the "**submit Lab6prephase2**" command, **up to 48 hours before you lab's Phase 2 deadline given below**.

3.4. Phase 2

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 start of lab during the second week of Lab 6. The command for Phase 2 is '**submit Lab6phase2**'.

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 'submit LabX -c', where LabX corresponds to a submission phase/lab.

4. 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, top level test bench included.**
- 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 the optional complete source grading, with "submit Lab6prephase2", and 3 for the complete mapped grading in Phase 2, with "submit Lab6phase2".
- **All module names and the input/output port names must exactly match specifications (including being all lower case).** Failure to do this will cause you to fail the grading script and it will be counted as one attempt.

5. 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.

[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

6. Protocol Description

6.1. 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 full-speed USB 1.1, which is 12 Mbits/second, ±3.5% (USB 2.0 allows data rates of 480Mb/s, but these will not be covered in this project). Note that since USB would normally have bit stuffing after 6 consecutive ones in the input data, your design must be able to handle 7 consecutive non-transitions in the encoded INRZI datastream without any errors. It does not need to be able to handle more than that since that would normally be prevented by bit-stuffing.

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

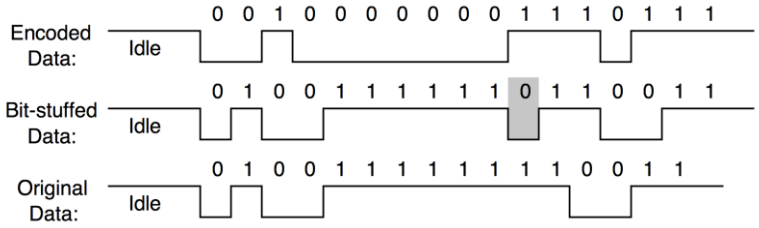
6.2. Data Encoding

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. ***Note: Inverted NRZI encoding is opposite of the USB standard, which just uses NRZI.***

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

---

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. The figure below 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. Our grading testbench, and your testbench, must never encode more than six ones in a row. The below "bit-stuffed data" and "encoded data" are valid testbench bitstreams, but the below "original data" must not be used.**



Inverted NRZI encoded and bit-stuffed data.

original data, while remaining in the same state will be seen as a "1" in the original data. Figure 1 shows data before and after the Inverted NRZI decoding.

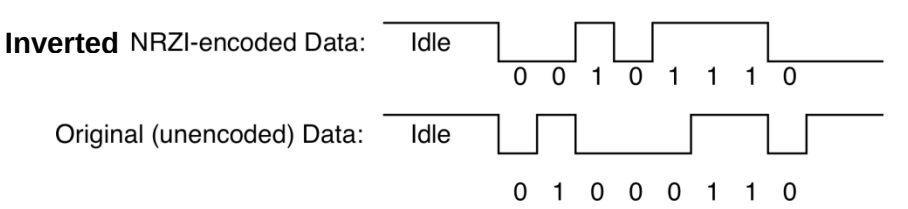


Figure 1: NRZI decoded data

6.3. 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.

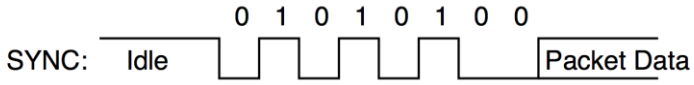


Figure 2: USB SYNC Byte

The receiver will begin capturing and storing data beginning with the first byte after a SYNC byte. Sending partial bytes is not supported by the standard, and will cause an error condition. The reading will continue until an EOP (End Of Packet) is received. The EOP is three data bit durations long, and is defined as both the D+ and the D- lines going to a logic low state (they are no longer differential) for 2 bit durations, followed by D+ high and D- low for one bit duration. After this, the bus may remain idle by leaving D+ high and D- low. Figure 3 shows the EOP signal followed by an idle period, in which D+ is shown as a black line and D- is shown as a dashed, blue line. Note that the testbench must not provide a datastream with D+ low for more than 7 bit durations in a row, so this imposes further restrictions on the last few bits of a packet.

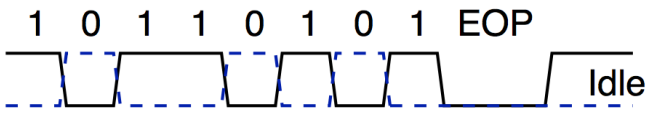


Figure 3: USB EOP signal

7. 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 'n_rst' lines. You will need to design and write test benches for each of the blocks above, as well as the top-level block (with the exception of the rcv_fifo, which will be provided). A test bench for rcv_fifo will be required for '**submit Lab6phase1**'. Each block is described in the following section, including the block name, basic functionality, and the input & output signals of the block.

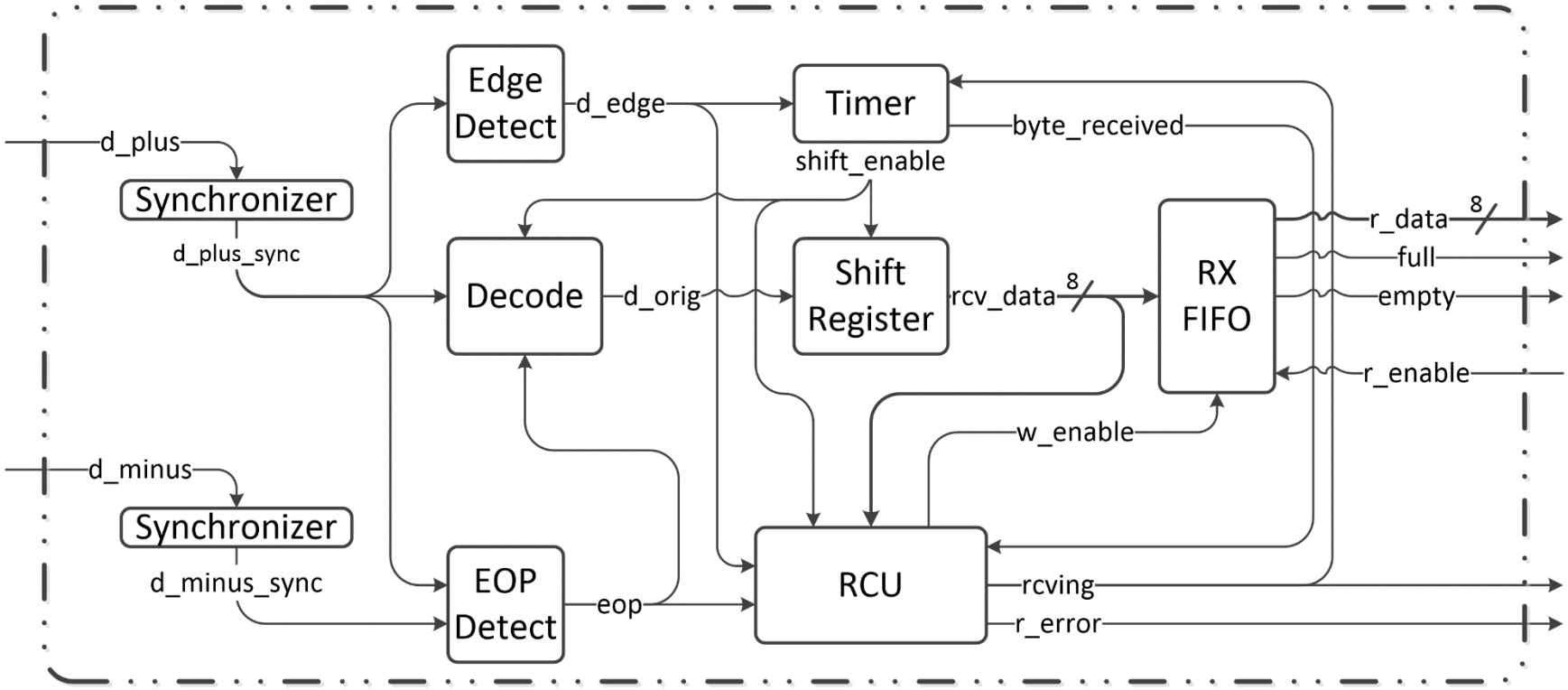


Figure 4: Top-level Architecture Block Diagram

7.1. Top-Level File Description and Requirements

The top-level design file is where you must connect up all of the other functional unit design files. As a reminder only the blocks specified by the COMPONENT_FILES makefile variable will be compiled during the source simulation of your design. Also, the entity (including port names) for the top-level design block must exactly match what is presented below, or your design will fail the grading tests.

7.1.1. Requirements

The required entity name is: usb_receiver

The required filename is: usb_receiver.sv

7.1.2. Port Descriptions

Signal	Direction	Description
clk	Input	This is the system clock (96 MHz), all state transitions will occur on the rising edge of the clk.
n_rst	Input	This is an asynchronous, active-low system reset, which causes the rcu to go to the idle state.
d_plus	Input	The positive USB input line.
d_minus	Input	The negative USB input line.
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 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.
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.
rcving	Output	This line will be set high when the circuit is receiving a packet.
r_error	Output	This flag indicates an error occurred while receiving the packet. It will be asserted until the start of the next packet.

7.2. Decode Block

The decode block is responsible for removing the Inverted NRZI encoding described in Section 2 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 at the same time as a shift enable. This block is much simpler than it sounds.

7.2.1. Requirements

The required entity name is: decode

The required filename is: decode.sv

The decode block must match the indicated signal timing:

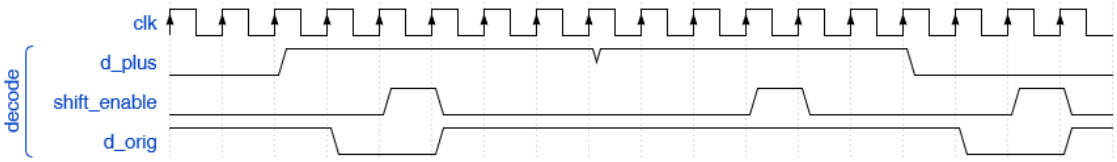


Figure 5: Decode Block Timing Specifications Diagram

The design must be implement a Moore-type FSM with an always_ff procedural block for all registers and separate always_comb procedural blocks or dataflow code to generate the next state and output. Also, be careful not to combine the clk and the shift_enable signal into an if-statement, as this would result in a gated clock.

7.2.2. Port Descriptions

Signal	Direction	Description
clk	Input	The system clock (96 MHz), will be used to latch the data into the stored bit when shift_enable is high.
n_rst	Input	This is an asynchronous, active-low system reset. When this line is low, the stored bit 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. It must only have an effect while shift_enable is high.
d_orig	Output	The decoded data.

7.3. Edge Detector Block

The edge detector block is very similar in operation to the decode block, except for the lack of the shift_enable. At each positive clock edge, the edge_detect block will compare the d_plus with the value of d_plus during the previous clock cycle, and 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 signal is used to synchronize the timer block.

7.3.1. Requirements

The required entity name is: edge_detect

The required filename is: edge_detect.sv

The decode block must match the indicated signal timing:

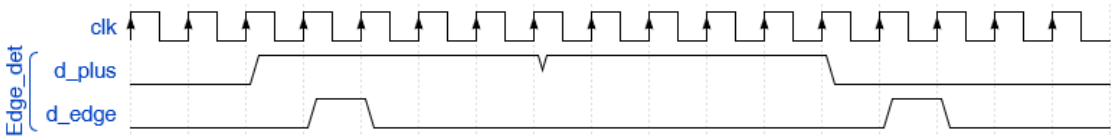


Figure 6: Edge Detect Block Timing Specifications Diagram

The design must be implement a Moore-type FSM with an always_ff procedural block for the all registers and separate always_comb procedural blocks or dataflow code to generate the next state and output.

7.3.2. Port Descriptions

Signal	Direction	Description
clk	Input	The system clock (96 MHz), the d_plus line is stored on each rising edge.
n_rst	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.

7.4. Timer Block

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 (96 MHz) runs at 8 times the speed of the USB input (12 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 because of errors in the data clock rate, the timer 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 blocks to design.

7.4.1. Requirements

The required entity name is: timer

The required filename is: timer.sv

You are required to create a timing diagram to help with your understanding of this block. You should use a starter wave (json) file provided in 'Lab6/docs' folder and fill in its details of the required signals. Use the wavedrom.com/editor.html tool to create this waveform. It will provide two example of input signals, one with the minimum data rate and one with the maximum data rate. Based on this timing diagram, you must choose where to place the shift_enable pulse.

This block can be built almost entirely using your already working flex_counter module. ***However, if you decided to build it as a customized sequential design, it must be implement a Moore-type FSM with an always_ff procedural block for the all registers and separate always_comb procedural blocks or dataflow code to generate the next state and output.***

7.4.2. Port Descriptions

Signal	Direction	Description
clk	Input	The system clock (96 MHz), the shift_enable output will only change on the rising edge of the system clock.
n_rst	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 n_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 rcving is low, this line will remain a 0.
byte_received	Output	The byte_received line will be high for 1 clock cycle each time a new byte has been shifted into the shift register, during the clock cycle after the 8^{th} shift enable.

7.5. Shift Register Block

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.

7.5.1. Requirements

The required entity name is: shift_register

The required filename is: shift_register.sv

This block should be a wrapper around your already working flex_stp_sr module.

7.5.2. Port Descriptions

Signal	Direction	Description
clk	Input	The system clock (96 MHz), all shifting will occur on the rising edge of clk.
n_rst	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.

7.6. Receive FIFO Block

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 rx_fifo.sv 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 FIFO is provided in the library ECE337_IP.

You may notice that the RCU block does not pay attention to the “full” flag of the FIFO. You may assume that the device reading out of the FIFO (the testbench) is reading fast enough to ensure the fifo never overflows. Thus, you do not need to check its full flag. Also please be aware that the full and empty signals are delayed by a few cycles due to internal synchronizers, so don’t expect them to be accurate immediately after a read or write.

The component declaration for the FIFO you can use is as follows:

```
module fifo
(
    input wire r_clk,
    input wire w_clk,
    input wire n_rst,
    input wire r_enable,
    input wire w_enable,
    input wire [7:0] w_data,
    output wire [7:0] r_data,
    output wire empty,
    output wire full
);
```

7.6.1. Requirements

The required entity name is: rx_fifo

The required filename is: rx_fifo.sv

7.6.2. Port Descriptions

Signal	Direction	Description
clk	Input	This is the READ clock signal, rclk. This clock signal is used by the FIFO to keep track of the current read address. This is also the WRITE clock signal, wclk. This clock signal is used by the FIFO to (1) latch data into the FIFO's RAM at its rising edge and when w_enable is asserted (Logic '1') and to (2) keep track of the current write address.
n_rst	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 w_data 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.

7.7. EOP Detector Block

This block detects the USB End-Of-Packet signal. As specified in Section 6.3, this occurs when both the d_plus and d_minus lines are set to a logic low. This is an extremely simple **combinational** logic block. Your testbench should exhaustively test this module.

7.7.1. Requirements

The required entity name is: eop_detect

The required filename is: eop_detect.sv

7.7.2. Port Descriptions

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.

7.8. RCU Block

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 receiving 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 receiving until the eop input signals that both d_plus and d_minus are low during a shift_enable strobe. Note that you may need to add a delay to the eop signal in order for it timing to be consistent with that of the decode output and timer. Once the

three-bit-long EOP is received, set the rcving line low. Be sure that your design can handle receiving a following packet starting immediately after the three-bit EOP.

If an EOP is reached with an incomplete byte in the shift register (i.e. the EOP should occur just after a byte is shifted into the FIFO), set the r_error flag high and do not store the last byte. Leave the r_error flag high until the next packet begins.

7.8.1. Requirements

The required entity name is: rcu

The required filename is: rcu.sv

7.8.2. Port Descriptions

Signal	Direction	Description
clk	Input	This is the system clock (96 MHz), all state transitions will occur on the rising edge of the clk.
n_rst	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 shift register is to shift in the next bit from the d_orig 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.
byte_received	Input	The byte_received line will be asserted for 1 clock cycle each time a new byte has been shifted into the shift register.
rcving	Output	This line will be set high when the circuit is receiving 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 occurred while receiving the packet. It will be asserted until the start of the next packet.