M A S S A C H U S E T T S   I N S T I T U T E   O F   T E C H N O L O G Y
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.111 Introductory Digital Systems Laboratory

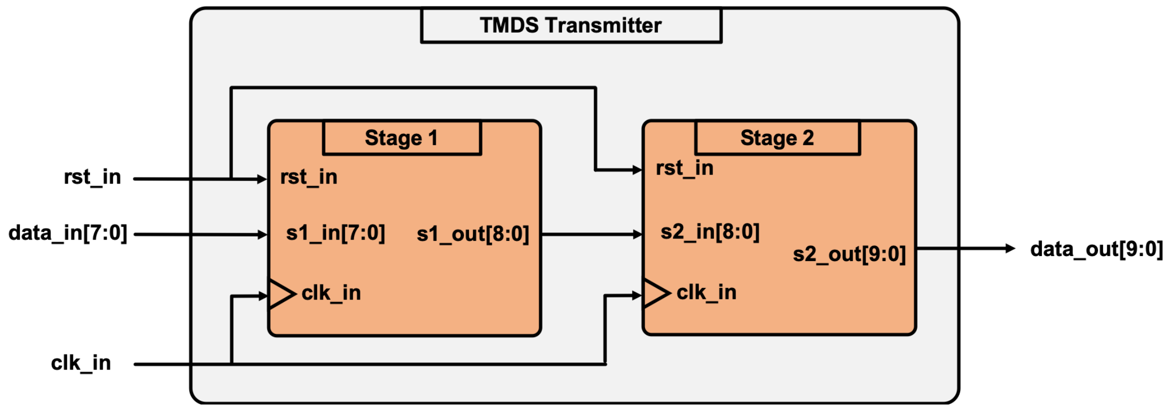
Fall 2019

Lecture PSet #6 (of 8)
Upload as PDF by 14:30 Thu, 10/03/2019

Problem 1 [5 points]
In LPset #3, you got a little practice with the math behind *a portion* of Transmission-Minimized Differential Signaling. In this LPset you'll implement *that portion* of a fully-functional TMDS encoder, and in the follow-up LPset you'll finish the complete design.

The full system you'll be building over the next two LPsets is depicted in the block diagram below. It takes in an 8-bit data value, and produces a 10-bit TMDS signal. This 10 bit signal would ultimately be transmitted one-bit at a time down a line using some differential signaling (but that's beyond the scope of this course). The TMDS transmitter is based on two stages:

- **Stage 1 (which you'll build here):** Converts the original 8 bit data to 9 bits using the encoding scheme discussed in LPset #3. The goal of this stage is to minimize bit transitions. This is a *stateless* operation where the output for a given input is based solely on the input but with a one cycle latency.
- **Stage 2:** Converts the 9 bit output from stage 1 into a 10 bit output with the goal of keeping the long-term running ratio of 1's to 0's to be as close as possible to 50/50 in order to minimize any building up a DC voltage offset on the transmission line. This module is *stateful* such that its output is based on both its current input and the previous output. We'll worry more about Stage 2 in LPset #7.

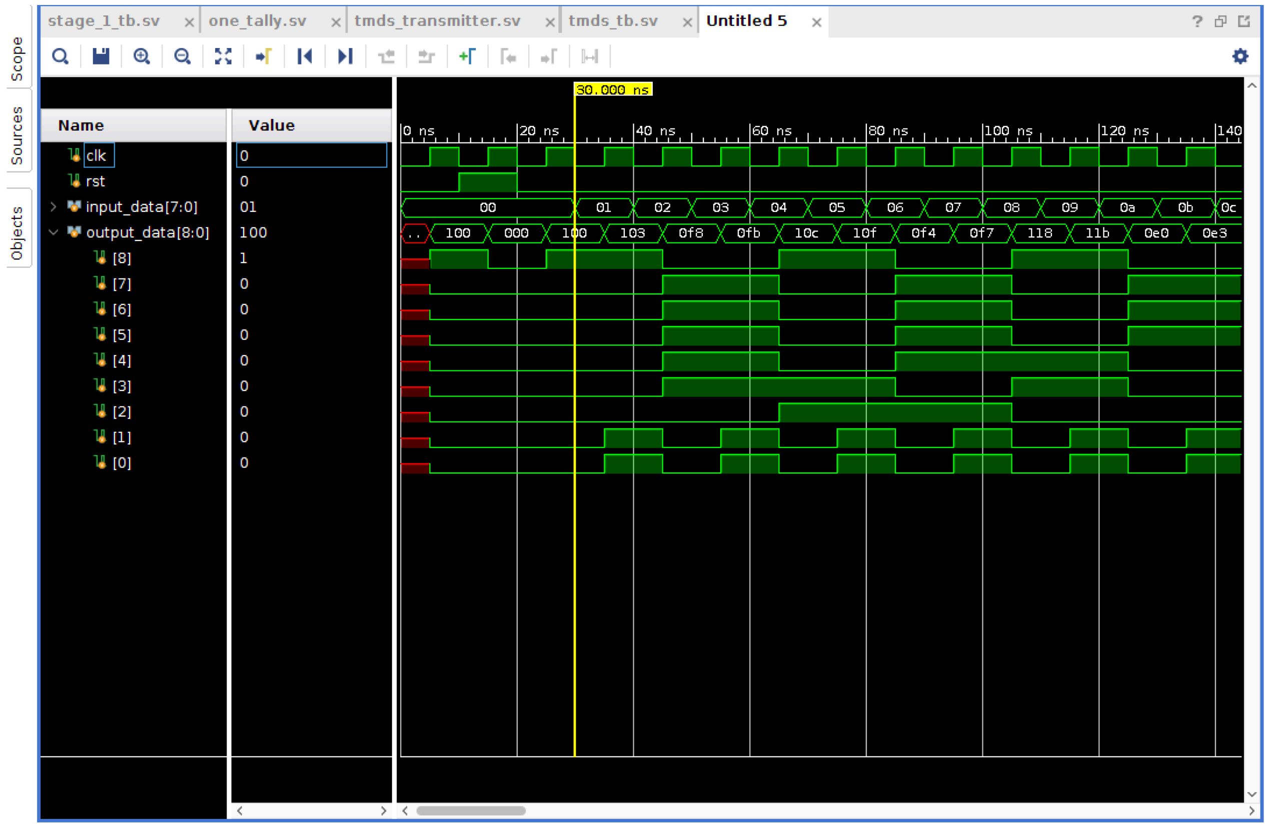


Write, test, and verify Stage 1. The system should have a one-cycle latency (one stage of D flip-flops encountered going from s1_in to s1_out)

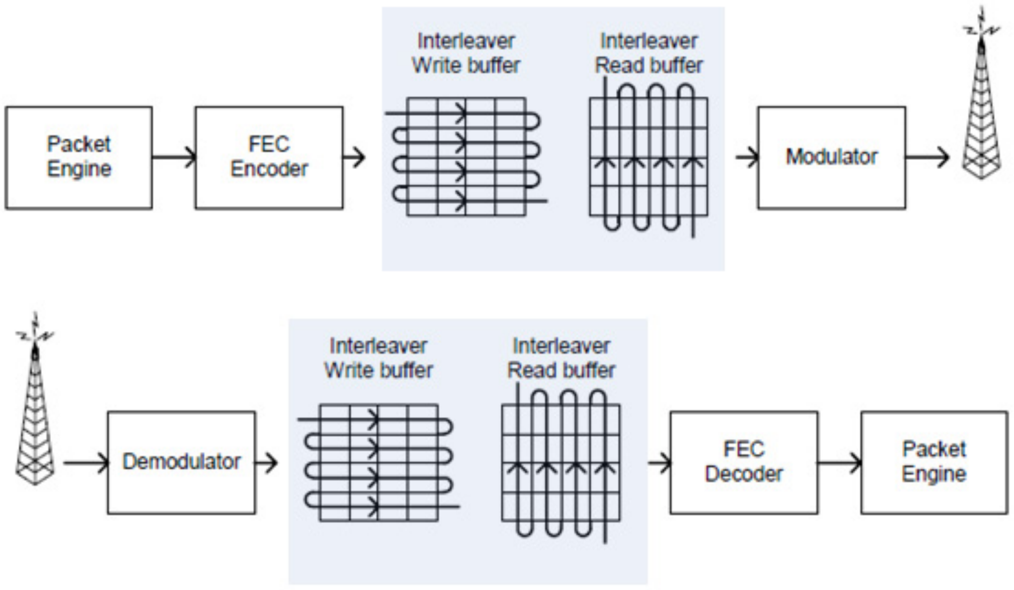
We've included starter code for development which is annotated with some comments. While we won't check and/or grade testbenches, it would be wise to use testbenches to verify that your system is operating correctly! You figured out how to manually do the math in LPset#3 so you should be able to compare your inputs and outputs here and write your own test cases.

```
        module stage_1 (input clk_in,
                        input rst_in,
                        input[7:0] s1_in,
                        output logic [8:0] s1_out
                         );

            //Generate two options: (See LPset 3)

            //Identify transitions in each:

            //Tally the transitions in each situation:

            //Based on tallies, choose one with fewer (or equal) and
            //produce correct output

        endmodule
```

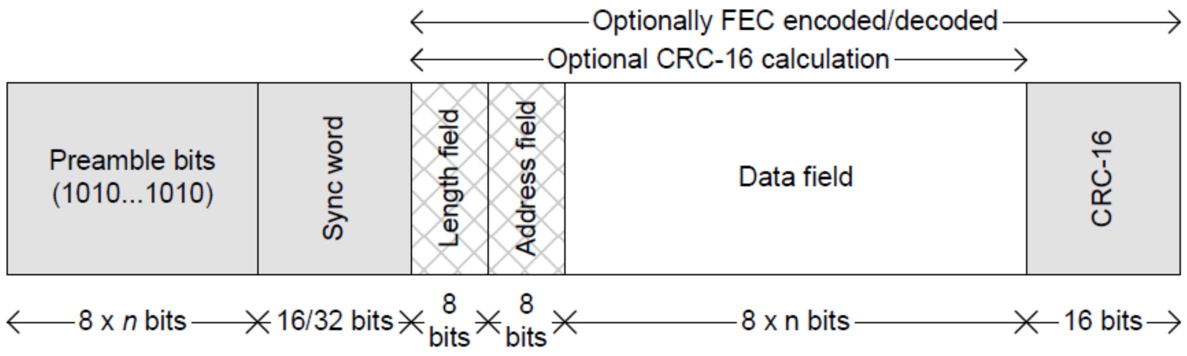
We've included a testbench for testing this module, called stage_1_tb.sv and posted on the course website. An example of the output is shown below. Make sure to test your modules operation by "decoding" some of the operations manually and/or comparing them with asserts in your testbench. **Submit your Verilog and a screenshot of the waveform with your test cases as a pdf.** Everyone should get a perfect grade.



Problem 2 [5 points] This problem is the second part of a 4 part LPset designing and implementing a digital communications system. It is based on an actual FPGA implementation for a low power wireless ECG monitor attached to a patient and transmitting to a receiver at a nurse base station.   The Interleaver module was the first part. This problem is focused on the Cyclic Redundancy Check (CRC) algorithm that is the front end of the FEC (Forward Error Correction) encoder.



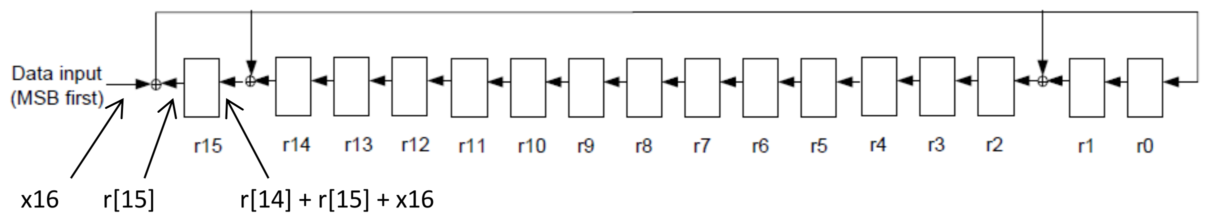
A typical transmitted data packet is show below.



The preamble bits and sync word are used by the receiver for synchronization. For this problem, you will only be concerned with the three blocks of data consisting of the Length Field, Address Field and Data Field. FEC, forward error correction, is a technique that allows a receiver to correct errors in a received packet by convolving the data and sending the parity bits. More on FEC in Lpset #7.

Cyclic Redundancy Check (CRC), often appended at the end of a data packet, is used to detect errors in data transmission and is capable of detecting all single and double errors and many multiple errors with a small number of bits. CRC is generated by a modulo 2 division with a generator polynomial. The remainder is the CRC. For our application, the generator polynomial is CRC16 (x16 + x15 + x2 + 1) with the CRC register initialized to all 1's prior to calculating the CRC.

[CRC16 is the generator polynomial for data packets in the USB:
https://www.usb.org/sites/default/files/crcdes.pdf
Initializing to all 1's ensures that leading 0s in front of a packet are protected by the CRC. The figure below shows the shift register implementation for the CRC.



In the shift register implementation, each "r" is a register, all clocked with a common clock. The common clock is NOT shown. The small round circles with the plus sign in the diagram are adders implemented with XOR gates. As shown, for register r15, the input is the sum of r[14], r[15] and data input x16; and the output is r[15] .

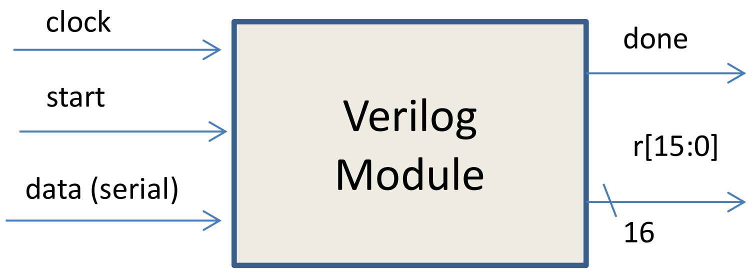
You can see from the location of the XOR gates (input to r15, r2 and r0, equivalent to a generator polynomial 0x8005) in the shift register configuration how CRC16 (x16 + x15 + x2 + 1) is implemented. The data input is x16 with the most significant ***bit*** (MSB) shifted in first. With each clock pulse, the next data bit is provided to x16. Using this hardware will give the following result:

Input: [4 bytes] **03 01 02 03**
Appended with CRC: [6 bytes] **03 01 02 03 30 3A**

In this example the first six data bits sent to x16 are six zero followed by two ones [**03].** After 32 bits are shifted in, the value in r[15:0] is the CRC [30 3A]. The data along with the CRC is then transmitted to the receiver wirelessly or other means. *At the receiver, putting the data and appended CRC (6 bytes total) through the same hardware used to generate the CRC will give a CRC of [00].* The zero CRC verifies that there are no errors in the received packets.

Problem: Write a Verilog module that takes the data, run it through the CRC generator and calculates the CRC. The input **start** pulses high for one clock cycle when data is available. When CRC calculation is completed, done is asserted with r[15:0] containing the CRC for the incoming data. Since the data input has the CRC appended, the resultant CRC is [00]. Be sure to initialize r[15:0] to 16'hFFFF at the start. For performance, done must be assert when all 48 bits are processed.

data input 48'h03_01_02_03_30_3A



Getting started:

Step 1: Using Vivado, create a new Verilog module with inputs and outputs as shown above.

Step 2: The Verilog module: when **start** is asserted (one clock pulse wide), reset your FSM; reset counters and other registers; and load any initial values. With each following clock pulse, begin the CRC calculation. Assert **done** when 48 bits are processed. The module should use only one clock domain always_ff @(posedge clock).

Step 3: Run a simulation using the test bench here and posted on the course website. Verify your design. The test bench includes a 10ns clock. Note the syntax [@posedge] for a test bench is slightly different than a Verilog module. The input data is 48'h03_01_02_03_30_3A and sent one bit at time. The first eight bits sent to the Verilog module are six zero followed by two ones corresponding to hex [03]. You may modify the test bench if needed for your implementation (generally not the case). In the actual FPGA ECG implementation, the data length is variable and processed one byte at a time.

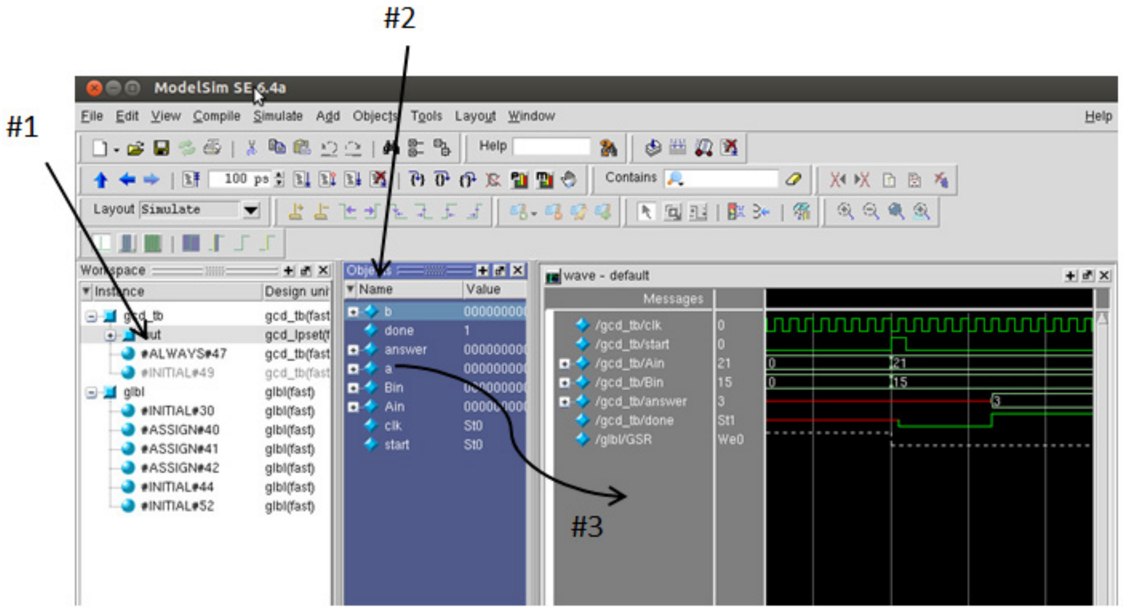
Step 4: Take a screen shot showing r[15:0] after 32 bits are shifted in and a screen shot showing r[15:0] when **done** is asserted. **Use hex radix for r[15:0]. Include the Verilog and screen shots in one pdf file. Upload to the course website.**

Lpset grading rubric

Grading	
1	Verilog with comments as needed
2	Functional Verilog & test bench
1	Screen Shot 1 r[15:0] after 32 bits are shifted in
1	Screen Shot 2 r[15:0] when **done** is asserted
5	**Total Grade**

In simulation, state values are unknown unless explicitly set. (Unknown values are shown in red during simulation. Outputs not defined are shown in blue.) For a simulation to run correctly, state variables must be initialized or set to some value at some point in the simulation. This can be accomplished by using a reset (recommended) or other input. For the CRC you can use the **start** pulse to initialize the CRC.

In simulation, by default, only inputs and outputs from the unit under test are displayed in the Wave window. It may be useful to display internal wires in your module that are not inputs nor outputs, for example, a bit counter. After running the initial simulation, to display the internal wires, click “uut” (unit under test) in the Workspace window (#1). This will display the internal signals in the Object window (#2). Drag the desired signals to the Wave window (#3).



To display the additional signals, rerun the simulation. In the Transcript window, type

```
restart         // force a restart
run 2000ns      // run simulation for 2000ns (longer if needed)
```

You can also select restart from the Run drop down menu.

You can verify your design with different inputs by comparing your results with a CRC calculator website http://www.sunshine2k.de/coding/javascript/crc/crc_js.html.

Select CRC-16, use custom CRC parameterization:

Uncheck “Input reflected”, uncheck “Result reflected”
Polynomial 0x8005
Initial value 0xFFFF
Final XOR value 0x0000
CRC input data (bytes) 0x03 0x01 0x02 0x03 0x30 0x3a