MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.111 Introductory Digital Systems Laboratory
Fall 2019

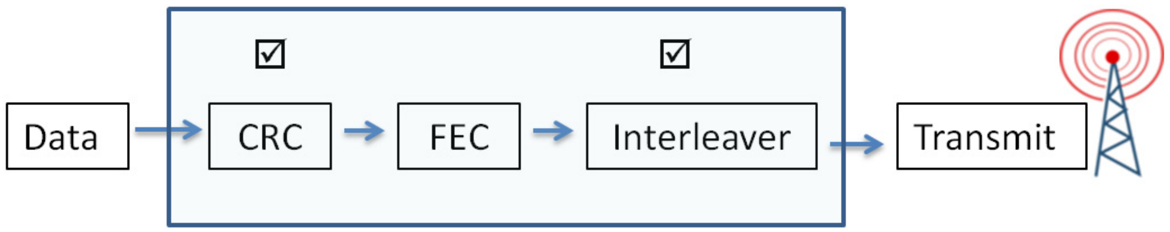
Lecture PSet #7 of 8
Convolution Codes

*Due: **Tue** 10/08/2019 Upload by 2:30p*

Problem 1
This is the third of four parts in implementing a communications system. In the last LPset, CRC was calculated. This LPset implements forward error correction (FEC). FEC can be used when it is known beforehand that a received signal will be very weak and likely to contain errors resulting in a high bit error rate (BER). This could be when the signal is transmitted over long distance such as the Mars Rover, New Horizon or in the case of medical electronics, when the transmitted power is very low by design, for example a tiny ultra-low power wireless transmitter attached to a patient.

One example of FEC is convolution codes[1]. Convolution codes involve calculating parity bits and then sending only the parity bits. At the receiver, the message can be recovered despite a high BER. One elegant and efficient method for recovering the data is the Viterbi algorithm[2] invented by Andrew Viterbi '57.

A typical data transmitting system with FEC would append a CRC to the data stream, apply a convolution encoder, interleave the data and transmit. (As with previous LPsets, this was actually implemented with a FPGA as part of a research project.) In previous lecture LPsets you have designed the CRC generator and the interleaver.

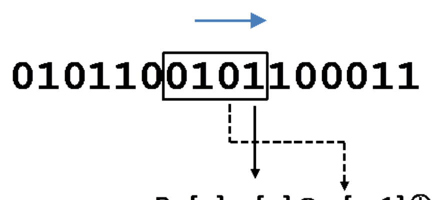


Your task will be to implement in Verilog a convolution encoder. A convolution encoder uses a sliding window to calculate the parity bits. The size of the window is the constraint length (k). The rate is the number of parity bits (r) expressed as 1/r , i.e. an encoder with two parity bits is a rate ½ encoder. Increasing the number of parity bits and the constraint length increases the resiliency to errors but at the cost of increased transmission time. In this problem we will use rate ½ constraint length 4 code (k=4).

[1] Fall 2011 6.02 Lecture Error Correction: Convolution Coding http://web.mit.edu/6.02/www/f2011/handouts/7.pdf
[2] Fall 2011 6.02 Lecture Viterbi decoding http://web.mit.edu/6.02/www/f2011/handouts/8.pdf

A data stream is shown below with convolution code (k=4) using these generators $g_0 = 1,1,1,1$ and $g_1 = 1,1,0,1$. The parity bits are then

$$p_i[n] = \left(\sum_{j=0}^{k-1} g_i[j]x[n-j]\right) \bmod 2 .$$



$P_0[n]=x[n] \oplus x[n-1] \oplus x[n-2] \oplus x[n-3]$
$P_1[n]=x[n] \oplus x[n-2] \oplus x[n-3]$

The parity bits are then sent as a single data stream:

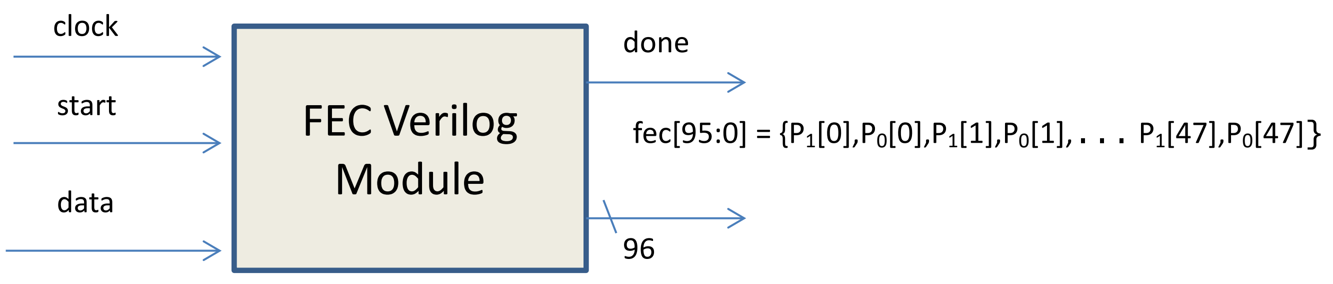
```
     P₁[0],P₀[0],P₁[1],P₀[1],P₁[2],P₀[2], ...
```

(In practice extra bits (trellis termination bits) are appended to the data before FEC to bring the convolution encoder to a known state. This helps in the decoding at the receiving end.)

Using the data with CRC appended from a previous LPset, implement a digital system with Verilog that takes the data, generates the parity bits and stores the output in **fec[95:0].** Notice that a rate ½ encoder doubles the number bits transmitted. The input **start** pulses high for one clock cycle when data is available. See the test bench for the exacting timing of data. When all the parity bits are generated, **done** is asserted with **fec[95:0]** containing encoded data stream. (In the actual implementation, data is streamed in a byte at a time thus requiring bit manipulation between bytes in generating the parity bits – a more complex design!)

As in the CRC generator, the bit stream sent to the FEC encoder is MS bit first. For the input data shown, the first eight bits sent to the convolution encoder input x[0], x[1], x[2] ... are six zero followed by two ones corresponding to hex **[03]**. Since the first data bit is x[0], with a constraint length (sliding window width) of four, set x[-1]= x[-2] = x[-3] = 0 to generate $P_1[0],P_0[0]$. When **start** is asserted, input will be available on data.

input data: 48'h03_01_02_03_30_3A



fec[95:0] = $\{P_1[0],P_0[0],P_1[1],P_0[1], . . . P_1[47],P_0[47]\}$

Since high throughput is required, **done** must be asserted as soon as the encoding is completed. “**done**” can be the output of combinatorial logic since it is sample later on in the system.

Here is how to get 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, reset your FSM; reset counters and other registers; and load any initial values required. When **start** is DE asserted, with each clock pulse, shift in one bit of data and calculate $P_1[n]$,$P_0[n]$ $0 \leq n \leq 47$ beginning with $P_1[0]$,$P_0[0]$ and shift into **fec[95:0]**. Note the ordering of the bits in **fec[95:0]**. After all parity bits have been calculated, assert **done**.
Step 3: Create a behavior test bench and verify your design with a simulation using the process outlined in LPset #6. You can use a 5ns clock in your test bench. The input data should be 48'h03_01_02_03_30_3A
Step 4: When **done** is asserted your encoding (using hex radix) should be

fec[95:0] = 96'h000E_8C03_7C0D_F00E_828C_0E5E

Step 5: Take a screen shot showing **fec[95:0]** when **done** is asserted. Use hex radix for **fec[95:0].** Include the Verilog (Verilog module and test bench) and screen shot in one pdf file and upload to the course website.

Lpset grading rubric

Grading	
Comments in Verilog when needed	
2	Verilog with comments meeting all the specs
1	Functional test bench
1	Screenshot showing fec[95:0] when done is asserted
5	**Total Grade**

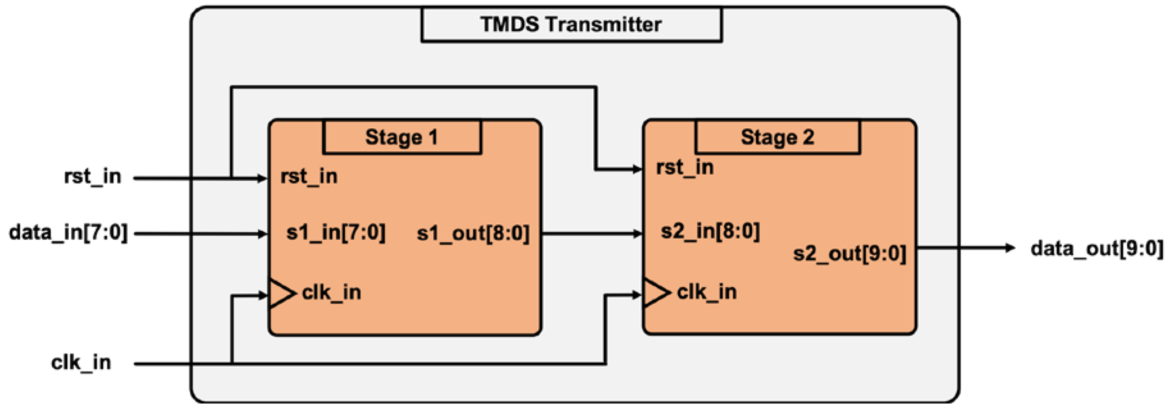
In simulation, state values are unknown unless explicitly set. (Unknown values are shown in red during simulation. Outputs not defined are shown in blue.) For a simulation to run correctly, state variables must be initialized or explicitly set to known value at some point in the simulation. This can be accomplished by using a reset or some other signal in your Verilog.

[Don't panic! Though this problem is three pages long, the Verilog design should be a dozen lines or so. It took me way longer to write this LPset and much longer for the actual design!]

<u>Problem 2</u> [5 points]
In LPset #3, you got a little practice with the math behind *a portion of* Transmission-Minimized Differential Signaling. In LPset #6 you wrote the first part of a TMDS pipeline. In this LPset you'll implement the rest of the TMDS encoder,

The full system you have been building is represented by the block diagram below. It takes in an 8-bit data block, and produces a full 10-bit TMDS signal. This 10 bit signal would then be transmitted one-bit at a time down a line. The TMDS transmitter is based on two stages:

- **<u>Stage 1:</u>** 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.
- **<u>Stage 2:</u>** 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. True TMDS uses a somewhat complicated ruleset, but for this problem we'll use an abbreviated, not entirely correct, one described below.

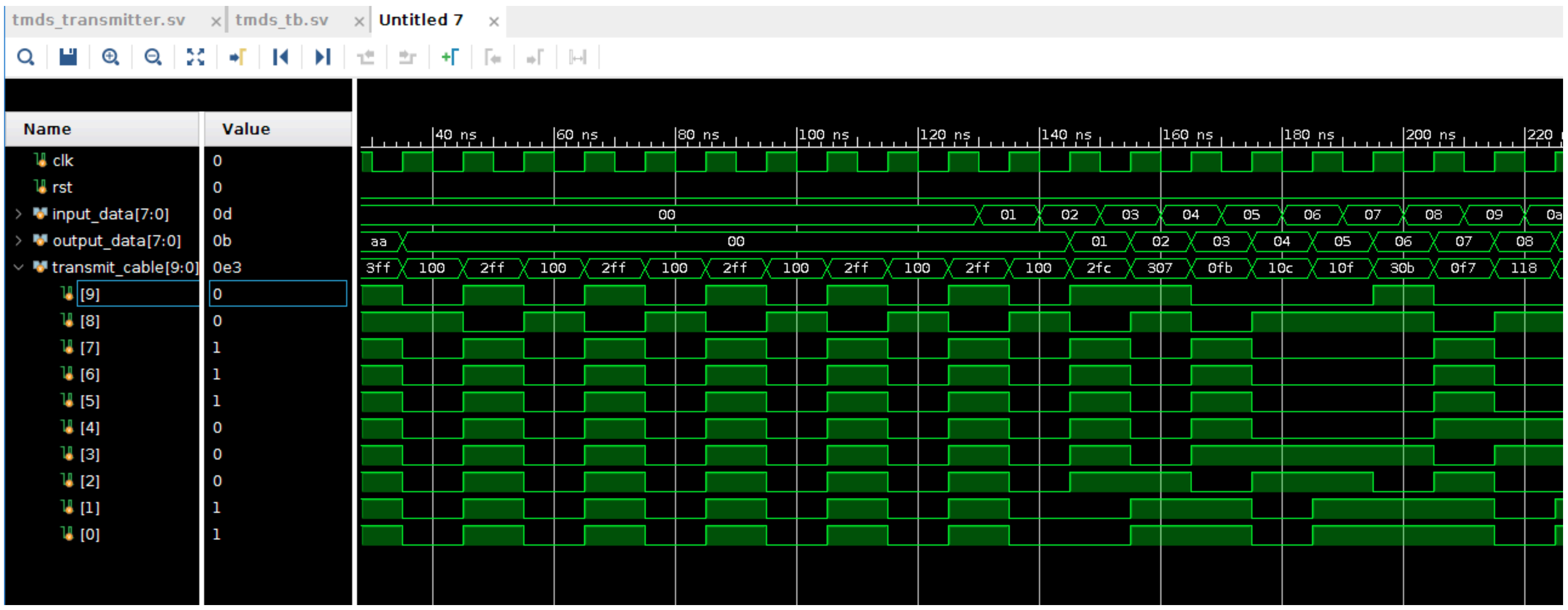


Write, test, and verify Stage 2. The system should have a one-cycle latency (one stage of D flip-flops encountered going from s2_in to s2_out). The system should take in the 9 bit output from stage 1 and do the following:

- Determine if there are more 1's or 0's in the 9 bit message.
- If there is a majority of 1's:
    - If the previous output had a majority of 1's, bitwise invert the current data, append a 1 as the 10th bit, and send out
    - If the previous output had a majority of 0's, append a 0 as the 10th bit and send out
- Otherwise (implied majority of 0's):
    - If the previous output had a majority of 1's, append a 0 as the 10th bit and send out
    - If the previous output had a majority of 0's, bitwise invert the current data, append a 1 as the 10th bit, and send out

The net result should be a system that transmits a signal that minimizes bit transitions within its 10 bit word, while also keeping the long term ratio of 1's to 0's on the line should stay roughly close to 1:1 regardless of the data being transmitted (this latter operation minimizes the buildup of a long-term DC voltage on the line which could mess with sensing circuitry on the receive side).

We’ve included starter code and a testbench for development. The starter code includes a fully-functioning TMDS receiver/decoder which is used in the testbench to analyze the output of your TMDS transmitter/encoder. When running the testbench, you should expect a correctly encoded signal to be decoded by the TMDS receiver/decoder two clock cycles later like shown below:



Submit your Verilog and a screenshot of your test bench results.