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 2017

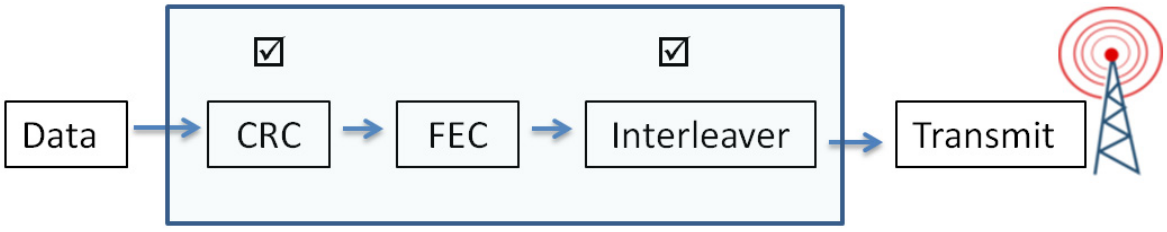
Lecture PSet #7 of 8
Convolution Codes (back to 6.02 – but without python!)

*Due: **Friday,** 10/6/2017 Upload by 23:59*

Forward error correction (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.

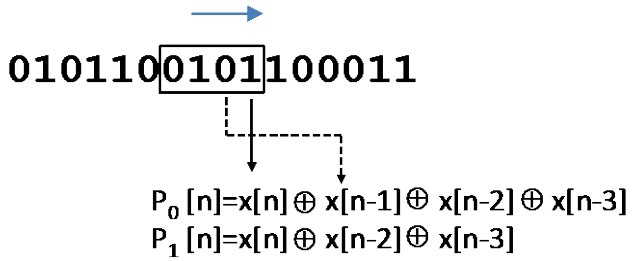



Your task will be to implement in Verilog a convolution encoder. A convolution encoder uses a sliding window to calculate the parity bits. The size of the window is the constraint length (k). The rate is the number of parity bits (r) expressed as 1/r , i.e. an encoder with two parity bits is a rate ½ 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

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



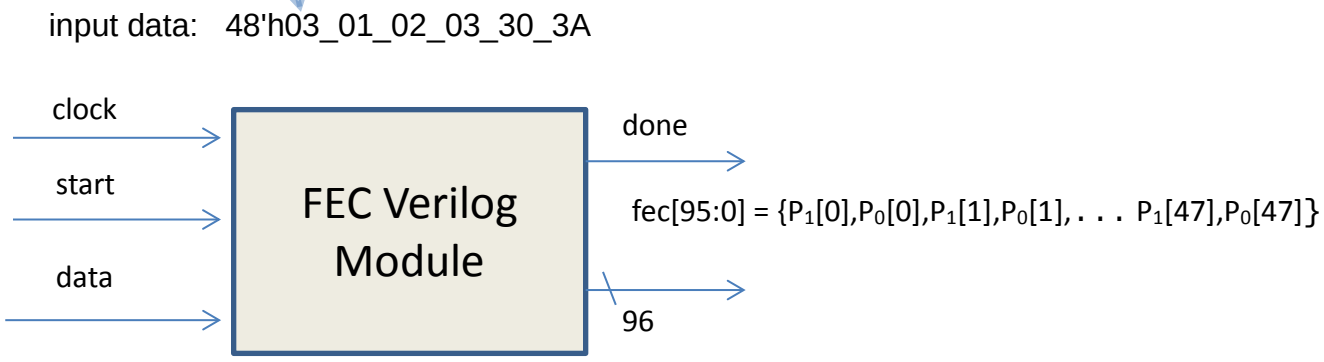

The parity bits are then sent as a single data stream:

$P_1[0],P_0[0],P_1[1],P_0[1],P_1[2],P_0[2],\ ...$

(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. 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.




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 ISE, 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 $P_1[n], P_0[n]$ $0 \le n \le 47$ 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:** Using ISE, create a behavior test bench (Verilog Test Fixture) and verify your design with a simulation using the process outlined in Lab 2 exercise 1(b) and 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 | |
|---|---|
| 1 | Easy to read & formatted Verilog (See "Verilog Editors" tab for help.) |
| 1 | Correct use of blocking/non-blocking assignments |
| 1 | Comments in Verilog when needed |
| 3 | Verilog meeting all the specs |
| 2 | Functional test bench |
| 2 | Screenshot showing fec[95:0] when done is asserted |
| **10** | **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!]