```
/**
---  This file is an example piece of system verilog that implements a
real linear feedback shift register but with
---  comments that help you understand what sections should exist and
what should be in them.
---  While there is alot of information on the internet for how to use
system verilog, there were so few examples
---  of how to code something real in system verilog so I thought I'd
write one that is not covered under some NDA.
---
---  Like any other coding effort, you should start off with a block
comment/header describing the module
---  The following conforms with a "doxygen" compatible format so you can
extract documentation.
---   https://www.doxygen.nl
**/
/**
 * @file    verilog_example_lfsr.sv
 * @brief   This module describes a finite state machine (FSM) for CDMA
communications.
 * @details It implements a "linear feedback Shift Register"
 * (LFSR) method of generating codes that, even if sent simultaneously by
two different transmitters, can be distinguished.
 *
 * This FSM uses a size n=13 LFSR to yield a 8191-bit "gold" code for
tranmission.  This is implemented as two registers
 * whose feedback is represented in the following way:
 *
 * shift_reg.lfsr1 = x^13+ x^12 + x^10+ x^9+ x^7+ x^6+ x^5 + x^1+x^0,
and the initial condition is an 11 bit seed value.
 * LFSR2 = x^13 + x^4 + x^3 + x^1 + x^0, and the initial condition is  0
0000 0000 0001
 *
 * This is code for a demonstration of a "real" network so the sending of
a data frame is somewhat random.
 * The first frame is sent after an amount of time determined by the
device's unique ID.
 * All other transmissions are at a fixed interval.
 *
 * The format of each frame is a header followed by the code followed by
the unique ID.
 *
 * inputs are the selection of the period of transmission and the unique
ID.
 * Outputs is the serial data stream representing the frame and a "data
valid" indication.
 *
 * @author  David Durfee  ddurfee@baycomp.com
 *
 * @date    Created 04/24/2022
```

```
 *
 * @copyright Copyright; David Durfee 2022-2023 ddurfee@baycomp.com \n
**/

/*
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

*/


/** this states that time is in 100nS increments with 1nS resolution so
4.1232 would be 4123nS **/
`timescale 100ns/1ns

/*
---There should be a global macros.sv file for macro defined constants
---   that are providing constants and types system-wide.
---We'll put them in this file for now since this file is a standalone
example and not part of a project yet.
---Normally you would use the directive `include "include_file.sv".
---Macros are upper case to differentiate them from variables.
*/

`define VERSION 1.1
`define UNIQUE_ID_W 64         // system has a register that contains a
unique ID.
`define ALMOST_NO_DELAY 'h2    // leave a couple of clocks between
transmissions but not mS
`define FIVE_MS_COUNT 'hC350   // number of clk ticks in 5mS -- 50,000
decimal.
`define TRUE  'b1
`define FALSE 'b0

/*
---There should be a global packages.sv file, mostly for typedefs that
are system wide.
---We'll put them here for now since this file is a standalone example
and not part of a project yet.
---Typedefs are lower case and are appended with "_t"
--- The "package" provides a namespace to allow for sharing.  It is
"imported" later in the file.
*/
package project_types;
```

```
// We define an enum for the period of transmission that is defined on
external pins on the chip.
typedef enum logic [2:0] {CONTINUOUS = 3'b000, FIVE_MS = 3'b001, TEN_MS =
3'b010, TWENTY_MS = 3'b100} period_t;

endpackage



/**
 * @brief   module cdma_stream.
 * @details Since this file contains only one module, the description for
the file and the module are the same
 *
 * @param [parameter] HEADER_VALUE:   the value can change easity but has
a default
 * @param [parameter] HEADER_W:   the header part of the transmission is
parameterized.
 * @param [in] clk:
 * @param [in] rst_n:
 * @param [in] xmt_period: an enum representing the xmt period.
 * @param [in] unique_id:
 * @param [out] data_out: serial bit stream of data
 * @param [out] data_valid: this is TRUE when xmting, FALSE when idle.
 */

/*
---Use "parameters" to allow for this module to be easily re-used in the
future.
---There were alot of issues with paramter usage in verilog so Verilog
2001 fixed that with the
---    ability to instantiate modules with named parameters.  (previously
they were done by order of assignment).
---(Do not use the "defparams" statement to change parameter values -- it
can create alot of problems!)
--- Note that the "import" command is used to include the definitions
found within the package we defined above.
*/
module cdma_stream
import project_types::*;
 #(parameter HEADER_VALUE = 8'hB4 , HEADER_W = 8)
(
 output logic data_out,           // serialized output
 output logic data_valid,
 input clk,
 input rst_n,                      // active low reset -- generally use
"_n" in the name to denote.
 input period_t xmt_period,      // sets the time between transmissions.
 input [(`UNIQUE_ID_W-1):0] unique_id      // value of unique identifier
register
  );

/*
```

```
---Like parameters, a local parameter is a constant that is local to a
module but it cannot be optionally be redefined.
---They were created to avoid inadvertant redefinition.
---More specifically, a local parameter can change value only if it
references a regular parameter that changes.
-- All Parameters are upper case.
*/

// local constants.

/*
---You should have nmemonic names for all of the constants you use in
your code (just like good "C" programming)
---NO MAGID NUMBERS!
---Widths of registers are a common parameter.  Append with _W in that
case.
*/

localparam DELAY_W = 18;
localparam INDEX_W = 14;
localparam COUNTER_W = DELAY_W;

localparam LFSR_W = 14; // Note, if you change this you likely want to
change the feedback.
localparam SEED_W = 11;
localparam SHIFT_REG_W = `UNIQUE_ID_W;

localparam CODE_COUNT = 'h2ECF8;

// states for each portion of the frame being transmitted and idle.
enum logic [2:0] {IDLE_ST = 0, XMT_HDR_ST = 4, XMT_CODE_ST = 5,
XMT_UID_ST = 6} xmt_state;  // state machine variable.


// Declarations
/*
---In general, the newer versions of verilog allow you to ignore what
data object to chose (wire,reg) in your design.
---Just use "logic".
---Logic can have one of 4 states 0, 1, X (undefined), Z (high
impedance).
---There are types that are only 2 states ( 0 and 1) for use in your test
bench that will simulate faster.
---Unlike constants that are capitalized, variables are snake case (lower
case with underscores)
---
---Just a comment on state machine programming (or any programming for
that matter), If you are creating logic
---   that tests a "flag" and you have a state machine as well, it is
likely you should add states to
---   a state machine and get rid of the flag to avoid confusion.
*/
```

```
//this is the main FSM counter which counts the repeat rate (delay) of
the frame transmission
logic [COUNTER_W-1:0] xmt_period_count;


/*
---Structs and Unions are synthesizeable and valuable so we should use
them.
---This design was originally for a custom ASIC that has limited area
---Reusing the registers for mutually exclusive functions saves some
registers.
---Using a union reuses the same registers but allows you to have
nmemonic names.
*/

union {  // make a union so the usage is nmenoic but only one register is
instantiated
  logic [HEADER_W-1:0] header;  // shifts out the header value.
  logic [LFSR_W-1:0] lfsr1;     // one of two feedback registers used for
the code.
  logic [`UNIQUE_ID_W-1:0] uid; // shifts out the unique ID.
} shift_reg;

logic [LFSR_W-1:0] lfsr2;     // second of the two feedback registers
that are combined.

/*
---"structs" are very helpful in grouping vairables into an data object
that can be passed as ports.
---There wasn't a compelling reason to use one in this application but
we've inserted one as an example.
*/

struct packed {    // the two variables that are combined to make the
data output stream.
  logic lfsr1_nxt;
  logic lfsr2_nxt;
  } next;

/*
---While they are not used in this manner in this file, unions are also
very helpful in
---   allowing you to access parts of a "word" and do it in a way that
allows for appropriate mnemonic names.
*/

union {  // make a union so the usage is nmenoic but only one register is
instantiated
logic [DELAY_W-1:0] delay;
logic [INDEX_W-1:0] index;
} counter;
```

```
// Assigns
/*
--- Intermediate variables are important for debugging.
--- Assignments allow you to view the combinatorial condition that are
separate from the registered version.
--- Also, it is likely you will want to see some of these closer to the
production version in an internal
---  logic analyzer and inserting them later would change the design.
--- They allow you to isolate a smaller portion of a large assignment.
--- If these assigns get complicated you should think about using and
"always_comb" block.
--- We have made use of one below.
--- Older verilog just had an "always" block and more specific versions
of "always_" are added
---  to prevent unintended consequences.
*/

assign next.lfsr1_nxt =
shift_reg.lfsr1[12]^shift_reg.lfsr1[10]^shift_reg.lfsr1[9]^shift_reg.lfsr
1[7]^shift_reg.lfsr1[6]^shift_reg.lfsr1[5]^shift_reg.lfsr1[1]^shift_reg.l
fsr1[0];
assign next.lfsr2_nxt = lfsr2[4]^ lfsr2[3] ^ lfsr2[1] ^ lfsr2[0];
//polynomial x^13 + x^4 + x^3 + x^1 + x^0

// Sequential logic
/*
---  Nost of your always blocks are going to be "always_ff" blocks.  This
was added to verilog becuase there
---  was a real possiblity of creating unintended latches.  (If you want
a latch use "always_latch")
---  Every register should have "clear" condition (usually done with a
rst_n signal)
---  Need to think about what the enable condition is for the registers
being defined in the always_ff block.
---  The two bullets above are satisfied with the if/else/if conditional
in the always_ff below.
---  Just as in "C" programming, it is not good practice to have too many
actions being taken in one block.
---  Break the tasks up into several blocks.  Often output registers are
separated into their own block.
*/


// state machine transitions.

// Start off with a unique delay and then transmit frames with a
consistent periodicity
// When each phase of transmission completes, this state machine sets up
the next state.

// The overall system is two statemachines.  One state machine choses
what "mode" you are currently in.
// The other state machine sends out the appropriate data for that
"mode".
```

```
/*
---You should initialize your variables on reset so there is no surpise
undefined initial
---   conditions in your design.
---Using "always_ff" rather than just "always" helps the compiler give
you appropriate warnings.
*/

always_ff @(posedge clk, negedge rst_n) begin
  if (~rst_n) begin
    // wake up out of reset and delay a unique amount of time for the
first transmission.
    counter.delay <= unique_id[(SEED_W-1):0];
    shift_reg.uid <= 'h0; // clear all of the shift registers, uid is the
largest
    lfsr2 <= 'b1;
    data_valid <= `FALSE;
    xmt_state <= IDLE_ST;

  end else begin

    case (xmt_state)
      IDLE_ST:  begin
        if (counter.delay == 'b0) begin // done - now waiting between
transmits
          // Do nothing in IDLE -- thats the idea -- until the counter
says to transmit
          // Then setup the transmit of the first part of the frame.
          counter.index <= HEADER_W-1;
          shift_reg.header <= HEADER_VALUE;
          data_valid <= `TRUE;
          xmt_state <= XMT_HDR_ST;
        end // if
      end

      XMT_HDR_ST: begin
        if (counter.index == 'b0) begin
           // transmission of the header part of the packet is over when
index == 0
         // setup the transmit of the code
          counter.index <= CODE_COUNT;                  // setup how many
bits to xmt
          counter.delay <= unique_id[(SEED_W-1):0];
          lfsr2 <= 'b1;
          data_valid <= `TRUE;
          xmt_state <= XMT_CODE_ST;
        end // if
      end

      XMT_CODE_ST: begin
        if (counter.index == 'b0) begin
          // transmission of the code part of the packet is over when
index == 0
```

```
          // setup the transmission of the UID.
          counter.index <= `UNIQUE_ID_W;
          shift_reg.uid <= unique_id[(`UNIQUE_ID_W-1):0];
          data_valid <= `TRUE;
          xmt_state <= XMT_UID_ST;
        end // if
      end

      XMT_UID_ST:  begin
        if (counter.index == 'b0) begin
          // transmission of the whole packet is over when index == 0
          // setup the next wait/idle period
          counter.delay <= xmt_period_count;
          data_valid <= `FALSE;
          xmt_state <= IDLE_ST;
        end // if
      end

      default : begin

      end

    endcase
  end // else
end  // always_ff




// State machine actions
// Send out data based on your state
/*
---Case statements should have a defalut to avoid a warning about an
unintended
---   latch being created.
---Also, since there are a limited number of actions, the output is
included in this
---   always statement.  Sometimes a separate always statement is helpful
to make
---   the outputs of the module more clear.
*/

always_ff @(posedge clk, negedge rst_n) begin
  if (~rst_n) begin
     data_out <= 1'b0;
  end else begin

    case (xmt_state)

      IDLE_ST : begin  // just mark time till its time to transmit.
        data_out <= 1'b0;
        if (counter.delay != 0) begin
          counter.delay <= counter.delay - 'b1;
        end   ;
```

```
      end

      XMT_HDR_ST : begin
        data_out <= shift_reg.header[HEADER_W-1] ;
        if (counter.index != 0) begin
          shift_reg.header <= shift_reg.header << 1; //shift to get the
next bit next time.
          counter.index <= counter.index - 'b1;
        end
      end

      XMT_CODE_ST : begin
        data_out <= shift_reg.lfsr1[0] ^ lfsr2[0];

        if (counter.index != 0) begin
          shift_reg.lfsr1 <= {next.lfsr1_nxt,shift_reg.lfsr1[LFSR_W-
1:1]}; //shift to get the next bit in shift reg.
          lfsr2 <= {next.lfsr2_nxt,lfsr2[LFSR_W-1:1]}; //shift to get the
next bit in shift reg.
          counter.index <= counter.index - 'b1;
        end
      end

      XMT_UID_ST : begin
        data_out <= shift_reg.uid[0];
        if (counter.index != 0) begin
          shift_reg.uid <= shift_reg.uid >> 1;
          counter.index <= counter.index - 'b1;
        end
      end

      default : begin
        data_out <= 1'b0;
      end

    endcase
  end // else
end // always_ff


// Combinatorial logic

/*
Need to set the count to provide for the right period of transmission.
(With the exception of the first time which is set by the unique
identifier)
*/
/*
---You should recognize that the block below is just a multiplexor of one
of four constants to one.
---Use the always_comb when you want combinatorial logic.  The compiler
doesn't enforce
---   it but you will get a warning if it didn't result in combinatorial
logic.
```

```
---So, it pays to get rid of warnings....
---Make sure you have a default in your case statement so you don't have
an unintended latch.
*/
  always_comb begin

    case (xmt_period)

      CONTINUOUS : begin  // pretty much start tranmitting right away.
        xmt_period_count = `ALMOST_NO_DELAY;
      end
      FIVE_MS : begin
        xmt_period_count = `FIVE_MS_COUNT;
      end

      TEN_MS : begin
        xmt_period_count = `FIVE_MS_COUNT*2;
      end

      TWENTY_MS : begin
        xmt_period_count = `FIVE_MS_COUNT*4;
      end

      default : begin
        xmt_period_count = 'h2;
      end


    endcase
  end // always_comb
endmodule
```