Exercise 7: UART and AXI

Prof. Dr. Sven Simon

1 UART

UART stands for Universal Asynchronous Receiver/Transmitter. UART takes a byte of data and transmits the individual bits in a sequential manner. At the destination, a second UART re-assembles the bits into complete bytes. The baud rate specifies how fast data is sent in bps. Common baud rates are (1200, 2400, 4800, 19200, 38400, 57600, and 115200).
The transaction starts when the transmitter sends a start bit to the receiver ('0'). Then comes the (8) data bits one after another, that are followed with an even-parity bit ('1' when the number of ones in the data bits is even). Finally, two stop bits ("11") are sent to indicate the end of the transaction. The entity declaration is given below (also in `uart.vhd`):

```
entity uart is

  port (
    clk       : in  std_logic;            -- 115.2MHz
    data_bits : in  std_logic_vector(7 downto 0);
    data_val  : in  std_logic;            -- AXI valid
    ready     : out std_logic;            -- AXI ready
    TX        : out std_logic);

end entity uart;
```

The input clock is 115.2MHz. The module should work as an AXI slave where ready is asserted to indicate that the UART module is ready to receive new data bits.

2 Fibonacci LFSR

Fibonacci linear-feedback shift register (LFSR) can be used to generate random numbers. It has a simple architecture that merely consists of a shift register initialized with a seed and a few xor gates. Implement the Fibonacci LSFR shown in Figure 1. The entity declaration is given below (also in lfsr.vhd)

```
entity lfsr is

  generic (
    SEED : std_logic_vector(1 to 16) := X"DEAD");
  port (
    clk      : in  std_logic;
    rand_bit : out std_logic);          --rightmost bit
end entity lfsr;
```

3 Postman

In order to test your implementation in Task 1, implement an AXI master that has a rom within initialized with the following values:

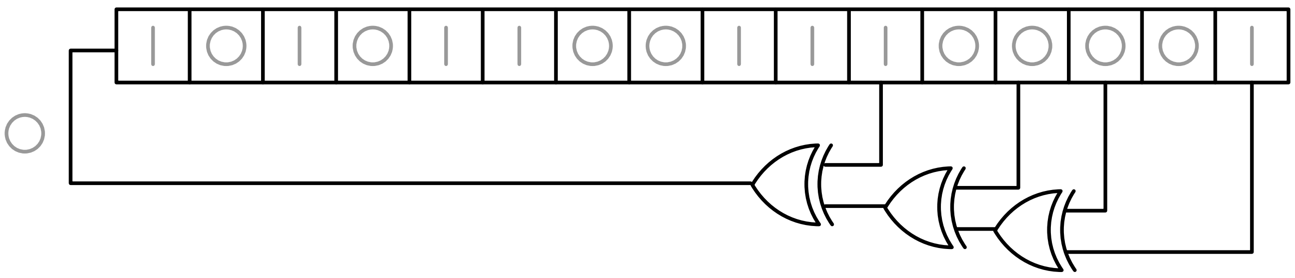

Figure 1: LFSR [By KCAuXy4p - Own work, CC0, `https://commons.wikimedia.org/w/index.php?curid=60225898`]

```
  type rom_t is array(0 to 14) of std_logic_vector(data_bits'range);
  signal rom : rom_t := ("01010110",     -- V
                         "01100001",     -- a
                         "01101100",     -- l
                         "01100001",     -- a
                         "01110010",     -- r
                         "00100000",     --
                         "01101101",     -- m
                         "01101111",     -- o
                         "01110010",     -- r
                         "01100111",     -- g
                         "01101000",     -- h
                         "01110101",     -- u
                         "01101100",     -- l
                         "01101001",     -- i
                         "01110011");    -- s
```

This module should work as an AXI master that sends the contents of the rom one at a time to the UART. The module should use the LFSR from Task 2 such that only when `rand_bit` is high, the transaction is initiated. The entity is given below (also given in postman.vhd):

```
entity postman is

  generic (
    SEED : std_logic_vector(1 to 16) := X"BEEF");  -- for lfsr

  port (
    clk       : in  std_logic;
    data_bits : out std_logic_vector(7 downto 0);
    data_val  : out std_logic;            -- AXI valid
    ready     : in  std_logic);           -- AXI ready
end entity postman;
```

AXI transaction

An AXI master initiates the transaction by asserting `data_val`. Once it is asserted, `data_val` as well as `data_bits` is kept unchanged. The transaction is complete when both *valid* and *ready* are sampled high at the clock edge. Important: **There must be no cominational path between input and output signals on both master and slave interfaces.**. That is, one cannot generate a ready *combinationally* from a valid or the other way around. More details about the AXI handshake can be found in `http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf`.

4 Verification

Write a minimal testbench that instantiates and interconnets `postman.vhd` and `uart.vhd`. In the testbench generate a 115.2MHz clock to drive the two module.