University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Sciences

CS 150 — J. Wawrzynek and N. Weaver
Fall 1999 — modified by R. Fearing and M. Beneš and P. Kim

Lab 5
Shift Registers and Counters

1 Objective

Counters and shift registers are commonly-used finite state machines. In this lab, you will build some and use the Xilinx board and see how they can be used for error correction.

2 Prelab

Read through this, enter the circuit described in Section 5, and answer the checkoff sheet questions.

3 Linear Feedback Shift Registers

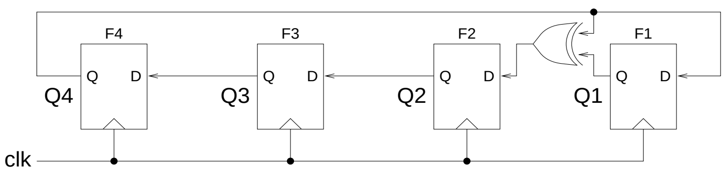

Figure 1: A four-bit linear feedback shift register

Linear feedback shift registers (later called LFSR), such as the one in Figure 1, are n-bit counters exhibiting pseudorandom behavior using little circuitry. They have many applications, including computer graphics and pseudorandom number generation. In this lab, you will use an LFSR to perform error checking - application used widely in digital communications.

Consider the operation of the LFSR shown in Figure 1. When the FFs are all zero, it does nothing. Every Q output is zero, the feedback path is zero, and the output of the XOR is zero, so the LFSR stays in this state.

When a 1 is introduced, the circuit counts through $2^4 - 1 = 15$ different non-zero bit patterns. The sequence is shown on right. Each box corresponds to a particular state of the shift register. The leftmost (top) bit decides whether the "10011" XOR pattern is used to compute the next value of the shift register, or if the register just shifts left.

This circuit uses 4 FFs, but you can build a similar circuit with any number of FFs. For a particular n, you may need more XOR gates on different positions. The table of primitive polynomials on Page 6 can be used to design these circuits. Section 7 discusses *why* these circuits do what they do.

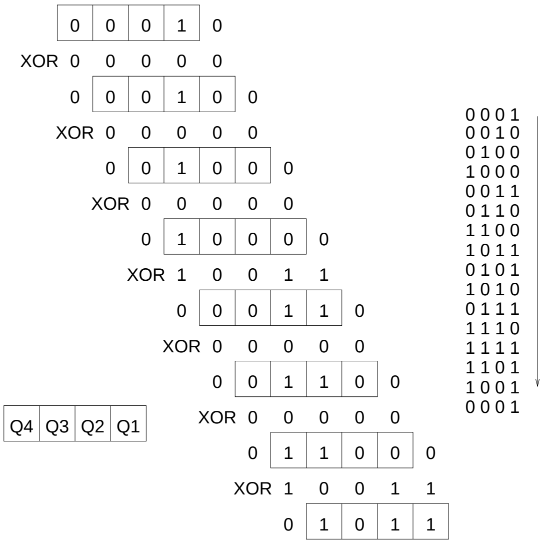

Figure 2: 4-bit LFSR sequence

3.1 Using the Table to Build an LFSR

To build a k-bit LFSR, number the flip-flops starting with 1 on the right. The feedback path comes from the Q output of the leftmost FF k.

Find the primitive polynomial of the form $x^k + \cdots + 1$ in the table on Page 6. Its $x^0 = 1$ term corresponds to connecting the feedback directly to the D input of flip-flop 1. Each term of the form x^n corresponds to connecting an XOR between flip-flop n and $n+1$.

To build the LFSR of Figure 1, I used the primitive polynomial $x^4 + x + 1$:

$$\underbrace{x^4}_{\text{F4's Q Output}} + \underbrace{x}_{\text{XOR between F1 and F2}} + \underbrace{1}_{\text{F1's D input}}$$

To build an eight-bit LFSR, use the primitive polynomial $x^8 + x^4 + x^3 + x^2 + 1$ and connect XORs between F2 and F3, F3 and F4, and F4 and F5.

4 Messages, Parity Bits, and Error Correction

Now, suppose we will make the following modification to our shift register. Instead of simply feeding Q_4 back into the first FF, we XOR it with an incoming bit sequence. The new circuit is shown on Figure 3.

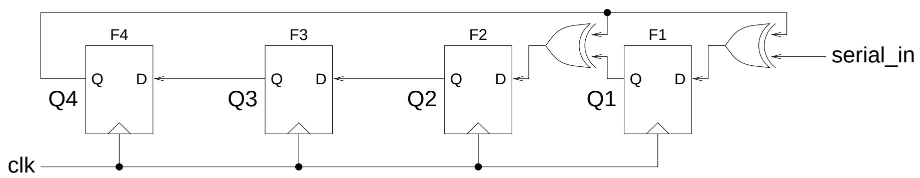

Figure 3: A four-bit linear feedback shift register

The operation is similar to the simple LFSR shown on Figure 1, but in this case the values of the shift register don't follow a fixed pattern — because they depend on the serial input sequence. What we are are interested in is the final value of the shift register after all serial input bits are shifted in. In this case it is the value of the register after 15 cycles : "1010".

Note that the length of the input sequence is $2^4 - 1 = 15$, the same as the number of different nonzero patterns for the original LFSR. Also note that our binary message really occupies only 11 bits, the remaining 4 bits are "0000". Those are reserved for "check bits" (or parity bits). They would be replaced by the final result of our LFSR — "1010".

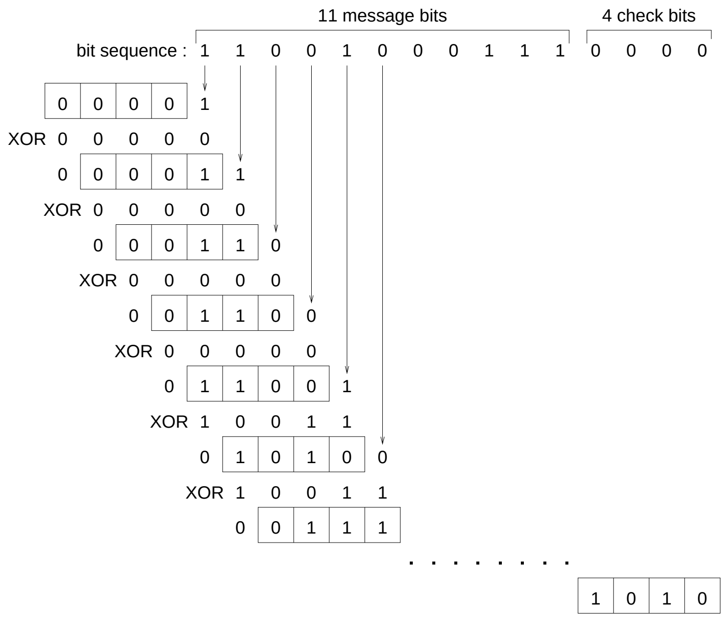

Figure 4: LFSR processing an input sequence

Thanks to some "magic", when we replace the last 4 bits of the sequence with our computed parity bits, and then run it through our modified LFSR, we get a "0000" final result. So what the 4 parity bits do to the message sequence, they "neutralize" it with respect to our LFSR engine. Here is the whole process is shown once again :

$$11001000111 \quad 0000 \longrightarrow 1010$$

$$11001000111 \quad 1010 \longrightarrow 0000$$

The new sequence, "110010001111010" is thus our complete "message". Now, why is this all useful? If we enhance our message with the 4 parity bits, we make our message less prone to errors. When the receiver receives the whole message, it will run it through our LFSR engine. When the message is received correctly (and, assuming there is no more than 1 error in it), the final value of the LFSR should be 0. When it is not zero, an error occurred in one of the 15 positions.

It turns out that, based on the principle of superposition, an all-zero message with "1" in the same position as on which the error occurred will produce the same result as the message containing the error. This could be summarized as follows :

$$11000000111 \quad 1010 \longrightarrow 0111$$

$$00001000000 \quad 0000 \longrightarrow 0111$$

Our message was received with an error in the 5th position and so the LFSR engine produced an "0111" output (usually called a *symptom*). The bottom sequence contains "1" in the fifth position, and will produce the same symptom.

There are 15 different symptoms — one for each bit position. Therefore once we know the symptom for our received message, if it is nonzero, we can determine the bit position on which the error occurred. So this method not only lets us detect whether there was an error or not, it also lets us correct that error, on the assumption that there was only one error.

5 What to Build

Your task is to Build a LFSR on the Xilinx Design Demonstration Board and an FSM that will perform the parity computation on a 255-bit message sequence. Your LFSR will have 8 FFs and will produce 8 parity bits. The previous section describes how to build an 8-bit LFSR.

Figure 5 shows the block diagram of the whole circuit. You only need to build the parts shaded in gray. Moreover, you can use standard XILINX parts for the 8-bit counter (`CC8CE`) and the comparator (look that one up in the lab reader). You need to design the Control FSM and the LFSR.

The ROM will be provided to you – it outputs a single bit based on the address provided by the 8-bit counter. This will be your serial bit sequence. The value on the DIP switches directly replaces the last 8 bits in the 256-bit sequence.

The circuit would have two modes, controlled by the SPARE button. In the first mode (MODE=1), your LFSR would shift in 256 bits from the ROM, then stop, and display the result on the 8 LEDs.

In the second mode (MODE=0), your LFSR would process only the "10000000....." sequence (the ROM already does that for you). The Control FSM would then stop as soon as the pattern computed by the LFSR matches the pattern on the DIP switches. The final number stored in the 8-bit counter (COUNT[7:0]) will be displayed on the 2 digits on your XILINX board. The number is directly related to the position of the error in the bit sequence. You should be able to understand why this is so, and be able to explain it to your TA (Hint: Think about the LFSR when all bits and the input are zero).

The *Control FSM* is quite simple. It has 3 inputs: the TC output of the counter, the result of the comparator, and the MODE button. It has 2 states: upon RESET, it enters an ACTIVE state — it enables the counter and the LFSR. Then it waits until either TC or the PAR=LED signals go high (depending on MODE), and then enters a DONE state. The only way to leave the DONE state is another RESET. Thus we only need 1 FF for the state variable. Use the standard procedure to design this FSM: start with a state diagram, then go through the truth table, all the way down to gates.

To summarize the things you need to build :

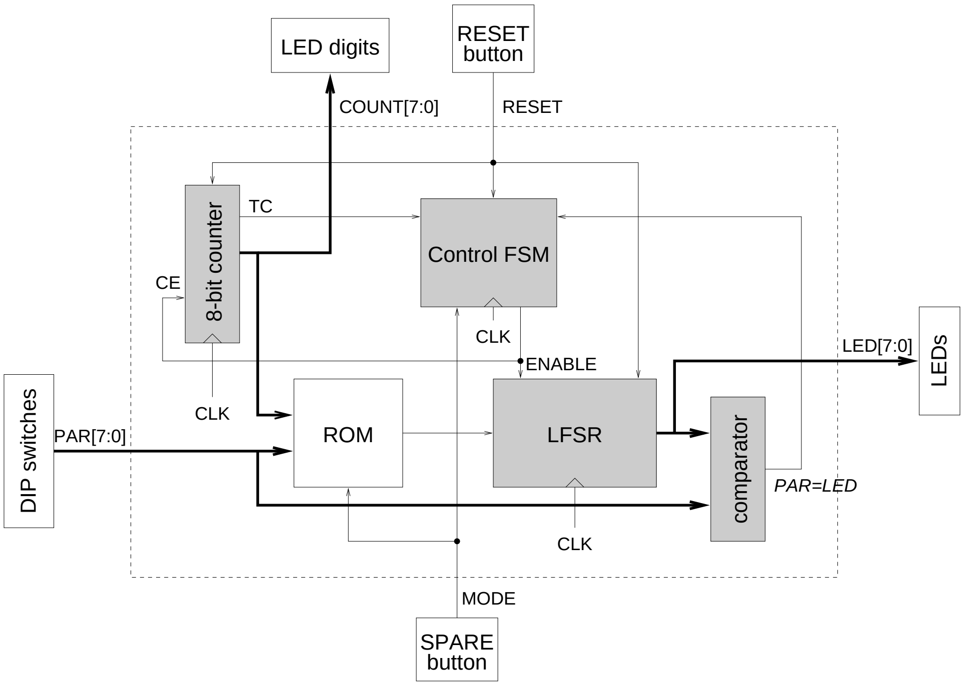

Figure 5: Lab5 block diagram

1. Build your 8-bit LFSR as shown in Section 3.1. Keep in mind that you need to provide an enable signal for the FFs. If the "enable" is off, the LFSR should keep its value.
2. Add the 8-bit counter.
3. Add the ROM block (go to the `wvlib/CS150` directory and use `ROM.SCH`), then connect it to the counter and your LFSR. You should copy the ROM schematics and it's symbol into your directory, since you will be editing them later.
4. Add the comparator and design the Control FSM.
5. Load the `LAB5.SCH` file from the `wvlib/CS150` directory and append it as a second page of your schematic just like in Lab 3. This is your front panel. Examine it and name signals in your schematic so that you connect to the switches and LEDs. The following is a list of LEDs, DIP switches and pin number. This is already done for you though.

	left							right
switch	19	20	23	24	25	26	27	28
LED	61	62	65	66	57	58	59	60

Table 1: DIP switch and LED pins

6 What to Do

You will use this circuit to compute the 8 parity bits for a 255-bit long message stored in a ROM. The ROM stores a random bit sequence that we have chosen — ours really contains 256 bits, the top bit must be "0" though.

Your FSM will have to cycle 256 times and the LFSR will process all 256 bits stored in the ROM and then stop — displaying the result value of the 8 LEDs. If your DIP switches are set to "00000000" initially, the pattern on your LEDs will be your parity bits.

The next step you enter the same pattern on the DIP switches, rerun the LFSR and you should get a "00000000" pattern out — just like a correct sequence should do.

Finally you will experiment with error correction. Leave your parity bits on your DIP switches after your previous task, and go back to your ROM schematic and change one bit in the bit sequence. You click on one of the ROM blocks, select the "INIT" attribute, and change one of the bits. Then you recompile the whole circuit and run the new sequence through your LFSR with your old parity bits. Since there is an error, you will get a nonzero value on the LEDs.

Next, you enter this error value on the DIP switches and run the second mode. The number on the 2 digits indicates the position of the error. Go back to your ROM schematic and check that indeed that is the same position you have introduced the error.

Again, to summarize :

1. Run MODE=1 with the DIP switches all zero. Write down the resulting parity bits.
2. Enter the parity bits on the switches, run MODE=1 and observe the LEDs. They should all be zero.
3. Introduce an error in your bit sequence — modify the ROM schematic. Recompile the design.
4. Use MODE=1 operation to detect the error and use MODE=0 to find it's position.

Since the XChecker cable clock runs at about 1MHz, it would be very difficult to see what your circuit is doing. Therefore we have used a 16-bit counter to divide the clock by 2^{16}, which slows the frequency to about 15Hz. At that speed you will be able to see the operation of the LFSR. If you want to observe your circuit cycle-by-cycle, you can use the "Apply" button on the Hardware Debugger control panel. However, first you will need to bypass the 16-bit counter in the schematics. Ask the TA if you don't know how.

7 Galois Fields

This circuit behaves as it does because it is performing multiplication on a *field*, a set with two operations defined on it: "addition" and "multiplication." The set is closed under these operations, the associative and distributive laws hold, there is both an additive and multiplicative identity element, there is an additive inverse for every element, and a multiplicative inverse for every non-zero element.

Familiar infinite fields include the set of rational numbers, the set of real numbers, and the set of complex numbers. The set of integers is *not* a field, since most integers do not have a integer multiplicative inverse.

Finite fields are called *Galois* (gal-WAH) fields. Binary numbers form a simple Galois field with two elements called $GF(2)$. The XOR function is "addition," and the AND function is "multiplication."

Consider polynomials whose coefficients come from $GF(2)$, that is, each term of the form x^n is either present or absent. For example, 0, 1, x, x^2, and $x^7 + x^6 + 1$ are all $GF(2)$-coefficient polynomials.

Adding these polynomials is easy. "Add" (XOR) each element individually—there is no carry:

$$\begin{array}{rccccc} & x^4 & +x^3 & & +x & +1 \\ + & x^4 & & +x^2 & +x & \\ \hline & & x^3 & +x^2 & & +1 \end{array}$$

Multiplying these polynomials is also easy. Multiplying by a monomial of the form x^n is like shifting to the left:

$$\begin{array}{rcccc} & & x^2 & +x & +1 \\ \times & & & x & +1 \\ \hline & & x^2 & +x & +1 \\ & x^3 & +x^2 & +x & \\ \hline & x^3 & & & +1 \end{array}$$

If we use this addition and take the results of this multiplication modulo[1] a prime polynomial[2] $p(x)$, these polynomials form a Galois field. For any degree, there is at least one prime polynomial. By using it, we can form $GF(2^n)$, the Galois field with 2^n elements, for any positive integer n.

Every Galois field has a primitive element: an element α such that every non-zero element can be expressed as a power of α. So if we know a primitive field element and can raise it to any power, we can obtain all non-zero elements of a field.

Certain choices of $p(x)$, the prime generating polynomial for a field (i.e., multiplication is taken modulo this polynomial), make the simple polynomial x a primitive element. It turns out there is such a $p(x)$, called a primitive polynomial, of every degree.

For example, the polynomial $x^4 + x + 1$ is primitive. So $\alpha = x$ is a primitive element, and successive powers of α will generate all non-zero elements of $GF(16)$:

$$\begin{array}{ll}
\alpha^0 & = 1 \\
\alpha^1 & = x \\
\alpha^2 & = x^2 \\
\alpha^3 & = x^3 \\
\alpha^4 & = x + 1 \\
\alpha^5 & = x^2 + x \\
\alpha^6 & = x^3 + x^2 \\
\alpha^7 & = x^3 + x + 1 \\
\alpha^8 & = x^2 + 1 \\
\alpha^9 & = x^3 + x \\
\alpha^{10} & = x^2 + x + 1 \\
\alpha^{11} & = x^3 + x^2 + x \\
\alpha^{12} & = x^3 + x^2 + x + 1 \\
\alpha^{13} & = x^3 + x^2 + 1 \\
\alpha^{14} & = x^3 + 1 \\
\alpha^{15} & = 1
\end{array}$$

This pattern of coefficients matches the bits in the figure on Page 1.

Primitive Polynomials
$x^2 + x + 1$
$x^3 + x + 1$
$x^4 + x + 1$
$x^5 + x^2 + 1$
$x^6 + x + 1$
$x^7 + x^3 + 1$
$x^8 + x^4 + x^3 + x^2 + 1$
$x^9 + x^4 + 1$
$x^{10} + x^3 + 1$
$x^{11} + x^2 + 1$
$x^{12} + x^6 + x^4 + x + 1$
$x^{13} + x^4 + x^3 + x + 1$
$x^{14} + x^{10} + x^6 + x + 1$
$x^{15} + x + 1$
$x^{16} + x^{12} + x^3 + x + 1$
$x^{17} + x^3 + 1$
$x^{18} + x^7 + 1$
$x^{19} + x^5 + x^2 + x + 1$
$x^{20} + x^3 + 1$
$x^{21} + x^2 + 1$
$x^{22} + x + 1$
$x^{23} + x^5 + 1$
$x^{24} + x^7 + x^2 + x + 1$
$x^{25} + x^3 + 1$
$x^{26} + x^6 + x^2 + x + 1$
$x^{27} + x^5 + x^2 + x + 1$
$x^{28} + x^3 + 1$
$x^{29} + x + 1$
$x^{30} + x^6 + x^4 + x + 1$
$x^{31} + x^3 + 1$
$x^{32} + x^7 + x^6 + x^2 + 1$

In general, finding these primitive polynomials is difficult. Looking them up in a table, such as the one on the right, is easiest.

The equivalences between Galois fields and hardware are:

Galois Field		Hardware
Multiplication by x	$\Longleftrightarrow$	shift right
Taking the result mod $p(x)$	$\Longleftrightarrow$	XOR-ing with the coefficients of $p(x)$ when the most significant coefficient is 1.
Obtaining all $2^n - 1$ non-zero elements by evaluating x^k for $k = 1, \ldots, 2^n - 1$	$\Longleftrightarrow$	Shifting and XOR-ing $2^n - 1$ times

This is a small fraction of the power of Galois fields, which are used throughout the large, rich field of error control (error-correcting) codes.

[1] This ensures the leading coefficient of the result is smaller than the leading coefficient of $p(x)$. Do this by subtracting a (polynomial) multiple of $p(x)$ from the result. Often this multiple is 1, corresponding to XOR-ing the result with $p(x)$.

[2] That is, a polynomial that cannot be written as the product of two non-trivial polynomials $q(x)r(x)$.

Name: ______________________ Name: ______________________
Lab Section (Check one)
M: ☐AM ☐PM T: ☐AM ☐PM W: ☐AM ☐PM Th: ☐PM

8 Checkoffs

1. Schematics complete before lab. TA: ____________(10%)

2. Use the table of primitive polynomials on Page 6 to draw a schematic for an LFSR with $2^9 - 1 = 511$ unique states. How many flip-flops do you need? How many XORs?

   TA: ____________(10%)

3. Build your LFSR shift register. TA: ____________(15%)

4. Compute the parity bits for the provided ROM sequence TA: ____________(20%)

5. Show that when entering the parity on the DIP switches, the LEDs will all go to "0"
   TA: ____________(20%)

6. Introduce an error into the ROM sequence and use the LFSR to find out where it was.
   TA: ____________(25%)

7. **Extra Credit (Only if completed during your Lab):**
   Make the SPARE button *toggle* between the two display modes
   OR show how you can single-step your circuit using the Apply button
   TA: ____________(5%)

8. Turned in on time TA: ____________(full credit: 100%)

9. Turned in one week late TA: ____________(1/2 credit: 50%)