Counters

• Special sequential circuits (FSMs) that repeatedly sequence through a set of outputs.
• Examples:
  - binary counter: 000, 001, 010, 011, 100, 101, 110, 111, 000,
  - gray code counter:
    000, 010, 110, 100, 101, 111, 011, 001, 000, 010, 110, ...
  - one-hot counter: 0001, 0010, 0100, 1000, 0001, 0010, ...
  - BCD counter: 0000, 0001, 0010, ..., 1001, 0000, 0001
  - pseudo-random sequence generators: 10, 01, 00, 11, 10, 01, 00, ...
• Moore machines with “ring” structure in State Transition Diagram:
What are they used?

- Counters are commonly used in hardware designs because most (if not all) computations that we put into hardware include iteration (looping). Examples:
  - Shift-and-add multiplication scheme.
  - Bit serial communication circuits (must count one “words worth” of serial bits).
- Other uses for counter:
  - Clock divider circuits
  - Systematic inspection of data-structures
    - Example: Network packet parser/filter control.
- Counters simplify “controller” design by:
  - providing a specific number of cycles of action,
  - sometimes used with a decoder to generate a sequence of timed control signals.
  - Consider using a counter when many FSM states with few branches.

Controller using Counters

Example, Bit-serial multiplier ($n^2$ cycles, one bit of result per $n$ cycles):

```
repeat n cycles {  // outer (i) loop
  repeat n cycles {  // inner (j) loop
    shiftA, selectSum, shiftHI
  }
  shiftB, shiftHI, shiftLOW, reset
}
```

Note: The occurrence of a control signal $x$ means $x=1$. The absence of $x$ means $x=0$. 

![Diagram of controller using counters](image-url)
Controller using Counters

- **State Transition Diagram:**
  - Assume presence of two binary counters. An "i" counter for the outer loop and "j" counter for inner loop.

  ![](image)

  TC is asserted when the counter reaches its maximum count value.
  CE is "count enable". The counter increments its value on the rising edge of the clock if CE is asserted.

- **Controller circuit implementation:**

  ![Controller Circuit Diagram]

  - Outputs:
    - \( CE_i = q_2 \)
    - \( CE_j = q_1 \)
    - \( RST_i = q_0 \)
    - \( RST_j = q_2 \)
    - \( shiftA = q_1 \)
    - \( shiftB = q_2 \)
    - \( shiftLOW = q_2 \)
    - \( shiftHI = q_1 + q_2 \)
    - \( reset = q_2 \)
    - \( selectSUM = q_1 \)
How do we design counters?

- For binary counters (most common case) incrementer circuit would work:

![Incrementer Circuit Diagram](image)

- In Verilog, a counter is specified as: \( x = x+1; \)
  - This does not imply an adder
  - An incrementer is simpler than an adder
  - And a counter might be simpler yet.

- In general, the best way to understand counter design is to think of them as FSMs, and follow general procedure, however some special cases can be optimized.

Synchronous Counters

*All outputs change with clock edge.*

- Binary Counter Design:
  Start with 3-bit version and generalize:

\[
\begin{array}{ccc|ccc|c}
  a & b & c & c'b' & a' & c' = ab' + a'b' + abc \\
  0 & 0 & 0 & 1 & 0 & 1 \\
  0 & 0 & 1 & 0 & 1 & 0 \\
  0 & 1 & 1 & 1 & 0 & 0 \\
  1 & 0 & 0 & 1 & 0 & 1 \\
  1 & 0 & 1 & 1 & 1 & 0 \\
  1 & 1 & 0 & 1 & 1 & 1 \\
  1 & 1 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{align*}
a^+ &= a' \\
b^+ &= a \oplus b \\
c^+ &= abc' + a'b'c + ab'c + a'bc \\
&= a'c + abc' + b'c \\
&= c(a' + b') + c'(ab) \\
&= c(ab) + c'(ab) \\
&= c \oplus ab
\end{align*}
\]
Synchronous Counters

- How do we extend to n-bits?
- Extrapolate \( c^+ : d^+ = d \oplus abc, \ e^+ = e \oplus abcd \)

- Has difficulty scaling (AND gate inputs grow with \( n \))

- CE is "count enable", allows external control of counting,
- TC is "terminal count", is asserted on highest value, allows cascading, external sensing of occurrence of max value.

Synchronous Counters

- How does this one scale?
  - Delay grows \( \alpha n \)
- Generation of TC signals very similar to generation of carry signals in adder.
  - “Parallel Prefix” circuit reduces delay:
Odd Counts

- Extra combinational logic can be added to terminate count before max value is reached:
- Example: count to 12

Alternative:

reset

4-bit binary counter

= 11 ?

4

load

4-bit binary counter

TC
Ring Counters

- "one-hot" counters
  0001, 0010, 0100, 1000, 0001, ...

  "Self-starting" version:

  What are these good for?

Johnson Counter

(a) Four-stage switch-tail ring counter

(b) Count sequence and required decoding
Asynchronous “Ripple” counters

Each stage is $\times 2$ of previous.

Look at output waveforms:

Often called “asynchronous” counters.

A “T” flip-flop is a “toggle” flip-flop. Flips it state on cycles when $T=1$.

Linear Feedback Shift Registers (LFSRs)

These are n-bit counters exhibiting *pseudo-random* behavior.

Built from simple shift-registers with a small number of xor gates.

Used for:
- random number generation
- counters
- error checking and correction

Advantages:
- very little hardware
- high speed operation

Example 4-bit LFSR:
4-bit LFSR

- Circuit counts through $2^4 - 1$ different non-zero bit patterns.
- Leftmost bit decides whether the “10011” xor pattern is used to compute the next value or if the register just shifts left.
- Can build a similar circuit with any number of FFs, may need more xor gates.
- In general, with n flip-flops, $2^n - 1$ different non-zero bit patterns.
- (Intuitively, this is a counter that wraps around many times and in a strange way.)

Applications of LFSRs

- Performance:
  - In general, xors are only ever 2-input and never connect in series.
  - Therefore the minimum clock period for these circuits is:
    \[ T > T_{2\text{-input-xor}} + \text{clock overhead} \]
  - Very little latency, and independent of n!
- This can be used as a fast counter, if the particular sequence of count values is not important.
  - Example: micro-code micro-pc
- Can be used as a random number generator.
  - Sequence is a pseudo-random sequence:
    - numbers appear in a random sequence
    - repeats every $2^n - 1$ patterns
  - Random numbers useful in:
    - computer graphics
    - cryptography
    - automatic testing
- Used for error detection and correction
  - CRC (cyclic redundancy codes)
  - ethernet uses them
Galois Fields - the theory behind LFSRs

- LFSR circuits performs multiplication on a field.
- A field is defined as a set with the following:
  - two operations defined on it:
    - “addition” and “multiplication”
  - closed under these operations
  - associative and distributive laws hold
  - additive and multiplicative identity elements
  - additive inverse for every element
  - multiplicative inverse for every non-zero element

Example fields:
- set of rational numbers
- set of real numbers
- set of integers is *not* a field (why?)

Finite fields are called Galois fields.

Example:
- Binary numbers 0,1 with XOR as “addition” and AND as “multiplication”.
- Called GF(2).

Galois Fields - The theory behind LFSRs

- Consider polynomials whose coefficients come from GF(2).
- Each term of the form $x^n$ is either present or absent.
- Examples: 0, 1, x, $x^2$, and $x^7 + x^6 + 1$

\[
= 1 \cdot x^7 + 1 \cdot x^6 + 0 \cdot x^5 + 0 \cdot x^4 + 0 \cdot x^3 + 0 \cdot x^2 + 0 \cdot x + 1 \cdot x^0
\]

- With addition and multiplication these form a field:
- “Add”: XOR each element individually with no carry:

\[
\begin{array}{c}
\frac{x^4 + x^3 + x + 1}{x^3 + x^2 + x + 1}
\end{array}
\]

- “Multiply”: multiplying by $x^n$ is like shifting to the left.

\[
\begin{array}{c}
\frac{x^2 + x + 1}{x^3 + x^2 + x + 1}
\end{array}
\]

\[
\begin{array}{c}
\frac{x + 1}{x^2 + x + 1}
\end{array}
\]
Galois Fields - The theory behind LFSRs

- These polynomials form a Galois (finite) field if we take the results of this multiplication modulo a prime polynomial \( p(x) \).
  - A prime polynomial is one that cannot be written as the product of two non-trivial polynomials \( q(x)r(x) \).
  - Perform modulo operation by subtracting a (polynomial) multiple of \( p(x) \) from the result. If the multiple is 1, this corresponds to XOR-ing the result with \( p(x) \).
- For any degree, there exists at least one prime polynomial.
- With it we can form \( GF(2^n) \).

Additionally, …
- Every Galois field has a primitive element, \( \alpha \), such that all non-zero elements of the field can be expressed as a power of \( \alpha \). By raising \( \alpha \) to powers (modulo \( p(x) \)), all non-zero field elements can be formed.
- Certain choices of \( p(x) \) make the simple polynomial \( x \) the primitive element. These polynomials are called primitive, and one exists for every degree.
- For example, \( x^4 + x + 1 \) is primitive. So \( \alpha = x \) is a primitive element and successive powers of \( \alpha \) will generate all non-zero elements of \( GF(16) \). Example on next slide.

\[
\begin{align*}
\alpha^0 &= 1 \\
\alpha^1 &= x \\
\alpha^2 &= x^2 \\
\alpha^3 &= x^3 \\
\alpha^4 &= x + 1 \\
\alpha^5 &= x^2 + x \\
\alpha^6 &= x^3 + x^2 \\
\alpha^7 &= x^3 + x + 1 \\
\alpha^8 &= x^2 + 1 \\
\alpha^9 &= x^3 + x \\
\alpha^{10} &= x^2 + x + 1 \\
\alpha^{11} &= x^3 + x^2 + x \\
\alpha^{12} &= x^3 + x^2 + x + 1 \\
\alpha^{13} &= x^3 + x^2 + 1 \\
\alpha^{14} &= x^3 + 1 \\
\alpha^{15} &= 1
\end{align*}
\]

- Note this pattern of coefficients matches the bits from our 4-bit LFSR example.

\[
\alpha^4 = x^4 \mod x^4 + x + 1 \\
= x^4 \text{xor } x^4 + x + 1 \\
= x + 1
\]

- In general finding primitive polynomials is difficult. Most people just look them up in a table, such as:

Primitive Polynomials

\[
\begin{align*}
x^2 + x + 1 & \quad x^{12} + x^6 + x^4 + x + 1 & \quad x^{22} + x + 1 \\
x^3 + x + 1 & \quad x^{13} + x^4 + x^3 + x + 1 & \quad x^{23} + x^5 + 1 \\
x^4 + x + 1 & \quad x^{14} + x^{10} + x^6 + x + 1 & \quad x^{24} + x^7 + x^2 + x + 1 \\
x^5 + x^2 + 1 & \quad x^{15} + x + 1 & \quad x^{25} + x^3 + 1 \\
x^6 + x + 1 & \quad x^{16} + x^{12} + x^3 + x + 1 & \quad x^{26} + x^6 + x^2 + x + 1 \\
x^7 + x^3 + 1 & \quad x^{17} + x^3 + 1 & \quad x^{27} + x^5 + x^2 + x + 1 \\
x^8 + x^4 + x^3 + x^2 + 1 & \quad x^{18} + x^7 + 1 & \quad x^{28} + x^3 + 1 \\
x^9 + x^4 + 1 & \quad x^{19} + x^5 + x^2 + x + 1 & \quad x^{29} + x + 1 \\
x^{10} + x^3 + 1 & \quad x^{20} + x^3 + 1 & \quad x^{30} + x^6 + x^4 + x + 1 \\
x^{11} + x^2 + 1 & \quad x^{21} + x^2 + 1 & \quad x^{31} + x^3 + 1 \\
\end{align*}
\]

Galois Field

<table>
<thead>
<tr>
<th>Multiplication by $x$</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Leftrightarrow$ shift left</td>
<td>$\Leftrightarrow$ XOR-ing with the coefficients of $p(x)$ when the most significant coefficient is 1.</td>
</tr>
</tbody>
</table>

Taking the result mod $p(x)$ $\Leftrightarrow$ Shifting and XOR-ing $2^{n-1}$ times.

Obtaining all $2^n - 1$ non-zero elements by evaluating $x^k$ for $k = 1, \ldots, 2^n - 1$

Building an LFSR from a Primitive Polynomial

- For $k$-bit LFSR number the flip-flops with FF1 on the right.
- The feedback path comes from the Q output of the leftmost FF.
- Find the primitive polynomial of the form $x^k + \ldots + 1$.
- The $x^i = 1$ term corresponds to connecting the feedback directly to the D input of FF 1.
- Each term of the form $x^n$ corresponds to connecting an xor between FF $n$ and $n + 1$.
- 4-bit example, uses $x^4 + x + 1$
  - $x^4 \Leftrightarrow$ FF4’s Q output
  - $x \Leftrightarrow$ xor between FF1 and FF2
  - $1 \Leftrightarrow$ FF1’s D input
- To build an 8-bit LFSR, use the primitive polynomial $x^8 + x^4 + x^3 + x^2 + 1$ and connect xors between FF2 and FF3, FF3 and FF4, and FF4 and FF5.
Error Correction with LFSRs

- XOR Q4 with incoming bit sequence. Now values of shift-register don’t follow a fixed pattern. Dependent on input sequence.
- Look at the value of the register after 15 cycles: “1010”
- Note the length of the input sequence is $2^4 - 1 = 15$ (same as the number of different nonzero patterns for the original LFSR)
- Binary message occupies only 11 bits, the remaining 4 bits are “0000”.
  - They would be replaced by the final result of our LFSR: “1010”
  - If we run the sequence back through the LFSR with the replaced bits, we would get “0000” for the final result.
  - 4 parity bits “neutralize” the sequence with respect to the LFSR.
- If parity bits not all zero, an error occurred in transmission.
- If number of parity bits = log total number of bits, then single bit errors can be corrected.
- Using more parity bits allows more errors to be detected.
- Ethernet uses 32 parity bits per frame (packet) with 16-bit LFSR.