CS150 - EE141/241A
Fall 2014
Digital Design and Integrated Circuits

Instructors:
John Wawrzynek and Vladimir Stojanovic

Lecture 21

Outline

- *Constant Multiplication*
- *Shifters*
- *LFSRs*

Constant Multiplication

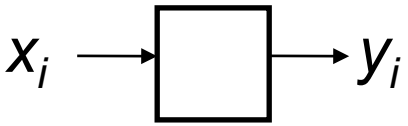
Constant Multiplication

- Our multiplier circuits so far has assumed both the multiplicand (A) and the multiplier (B) can vary at runtime.
- What if one of the two is a constant?

$$Y = C * X$$

- "Constant Coefficient" multiplication comes up often in signal processing and other hardware. Ex:

$$y_i = \alpha y_{i-1} + x_i$$

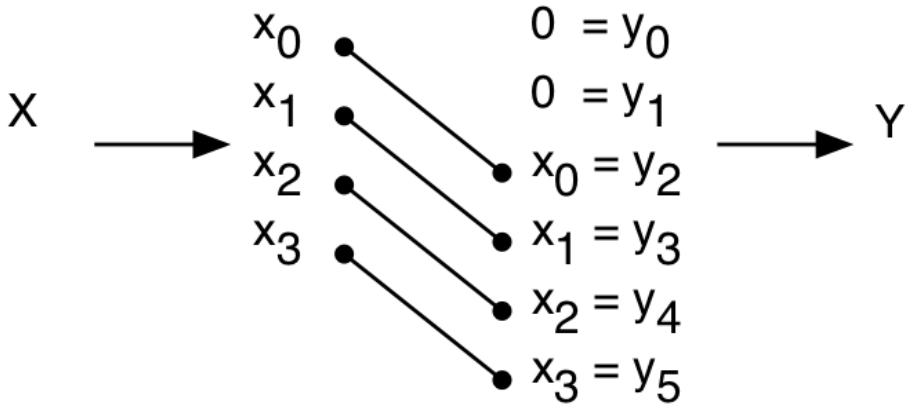


where α is an application dependent constant that is hard-wired into the circuit.

- How do we build and array style (combinational) multiplier that takes advantage of the constancy of one of the operands?

Multiplication by a Constant

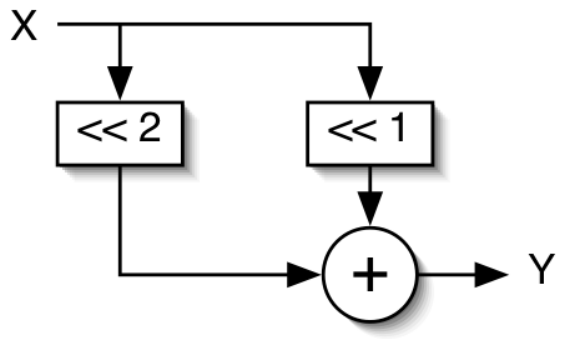
- If the constant C in C*X is a power of 2, then the multiplication is simply a shift of X.
- Ex: 4*X



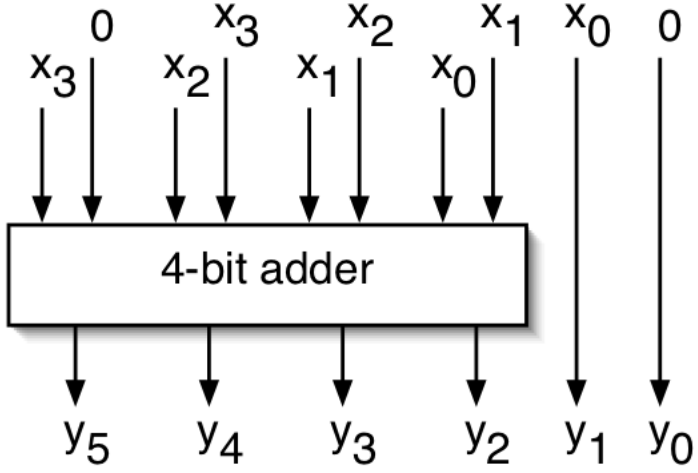
- What about division?
- What about multiplication by non- powers of 2?

Multiplication by a Constant

- In general, a combination of fixed shifts and addition:
  - Ex: $6*X = 0110 * X = (2^2 + 2^1)*X = 2^2 X + 2^1 X$

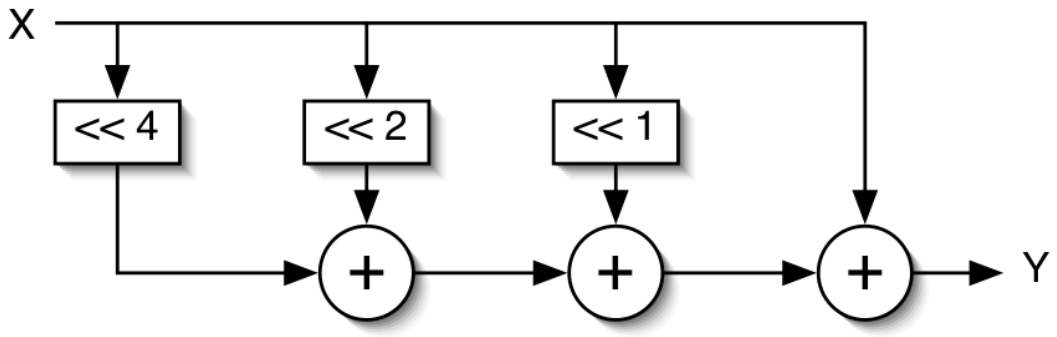


  - Details:



Multiplication by a Constant

- Another example: C = 23_{10} = 010111



- In general, the number of additions equals one less than the number of 1's in the constant.
- Using carry-save adders (for all but one of these) helps reduce the delay and cost, and using balanced trees helps with delay, but the number of adders is still the number of 1's in C minus 2.

- Is there a way to further reduce the number of adders (and thus the cost and delay)?

Multiplication using Subtraction

- Subtraction is ~ the same cost and delay as addition.
- Consider C*X where C is the constant value 15_{10} = 01111.

  C*X requires 3 additions.

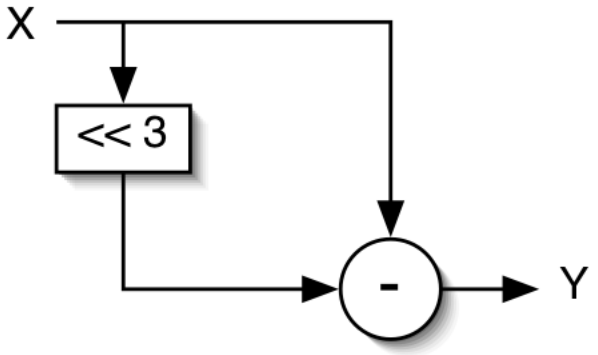
- We can "recode" 15

  from 01111 = $(2^3 + 2^2 + 2^1 + 2^0)$

  to 1000$\overline{1}$ = $(2^4 - 2^0)$

  where $\overline{1}$ means negative weight.

- Therefore, 15*X can be implemented with only one subtractor.



Canonic Signed Digit Representation

- CSD represents numbers using 1, $\bar{1}$, & 0 with the least possible number of non-zero digits.
  - Strings of 2 or more non-zero digits are replaced.
  - Leads to a unique representation.
- To form CSD representation might take 2 passes:
  - First pass: replace all occurrences of 2 or more 1's:

    01..10 by 10..$\bar{1}$0
  - Second pass: same as a above, plus replace 01$\bar{1}$0 by 0010 and 0$\bar{1}$10 by 00$\bar{1}$0
- Examples:

011101 = 29
100$\bar{1}$01 = 32 - 4 + 1

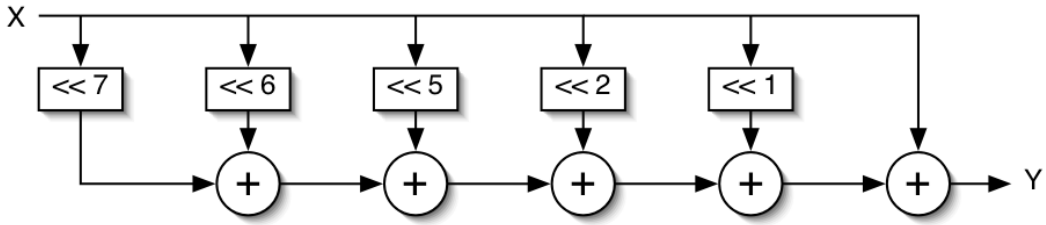
0010111 = 23
0011001$\bar{1}$
0101001$\bar{1}$ = 32 - 8 - 1

0110110 = 54
10$\bar{1}$10$\bar{1}$0
100$\bar{1}$0$\bar{1}$0 = 64 - 8 - 2

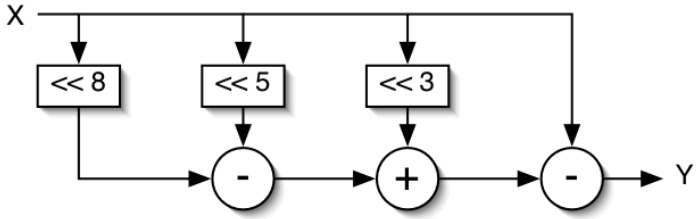
- Can we further simplify the multiplier circuits?

"Constant Coefficient Multiplication" (KCM)

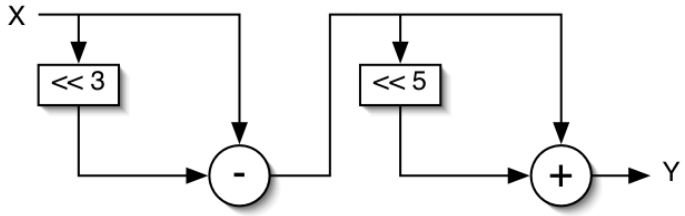
Binary multiplier: $Y = 231*X = (2^7 + 2^6 + 2^5 + 2^2 + 2^1+2^0)*X$



- CSD helps, but the multipliers are limited to shifts followed by adds.
  - CSD multiplier: $Y = 231*X = (2^8 - 2^5 + 2^3 - 2^0)*X$



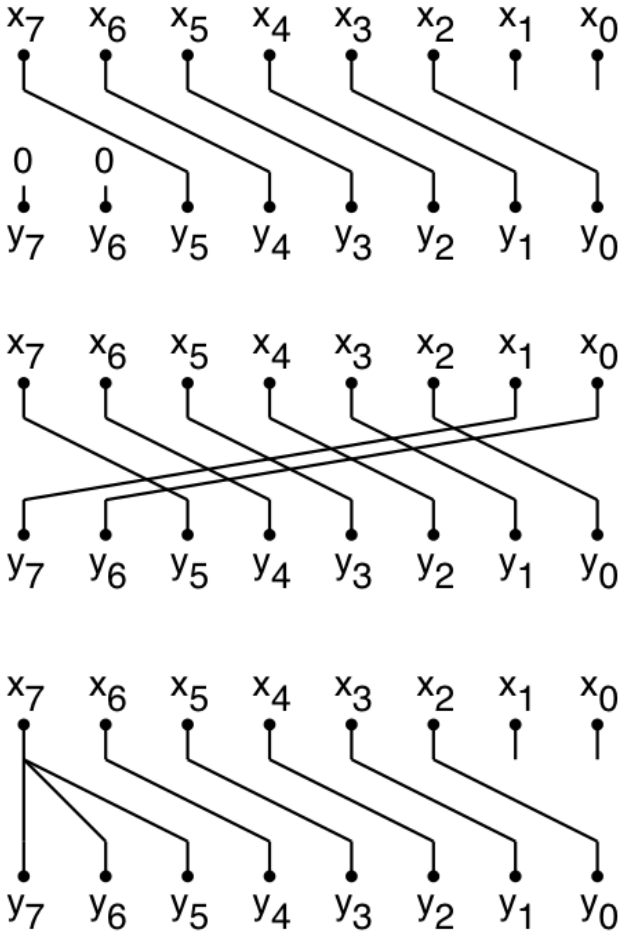
- How about shift/add/shift/add …?
  - KCM multiplier: $Y = 231*X = 7*33*X = (2^3 - 2^0)*(2^5 + 2^0)*X$



- No simple algorithm exists to determine the optimal KCM representation.
- Most use exhaustive search method.

Shifters

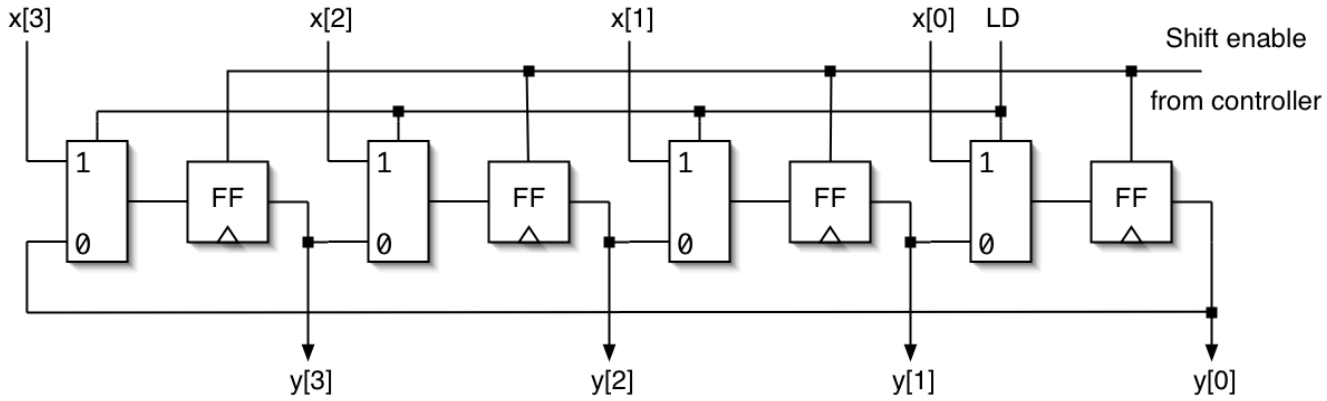
Fixed Shifters / Rotators Defined



Variable Shifters / Rotators

- Example: X >> S, where S is unknown when we synthesize the circuit.
- Uses: shift instruction in processors (ARM includes a shift on every instruction), floating-point arithmetic, division/multiplication by powers of 2, etc.
- One way to build this is a simple shift-register:

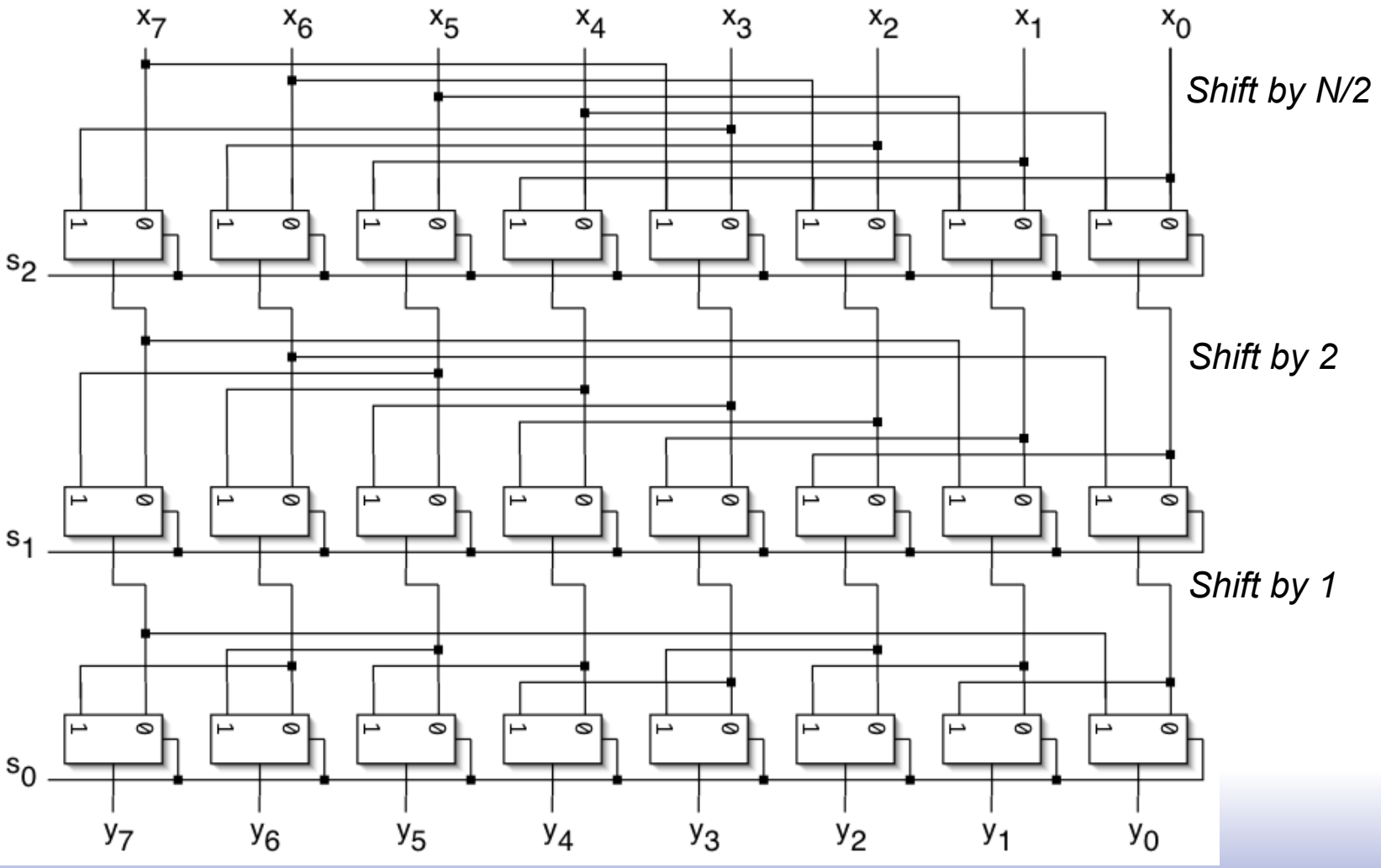
  a) Load word, b) shift enable for S cycles, c) read word.



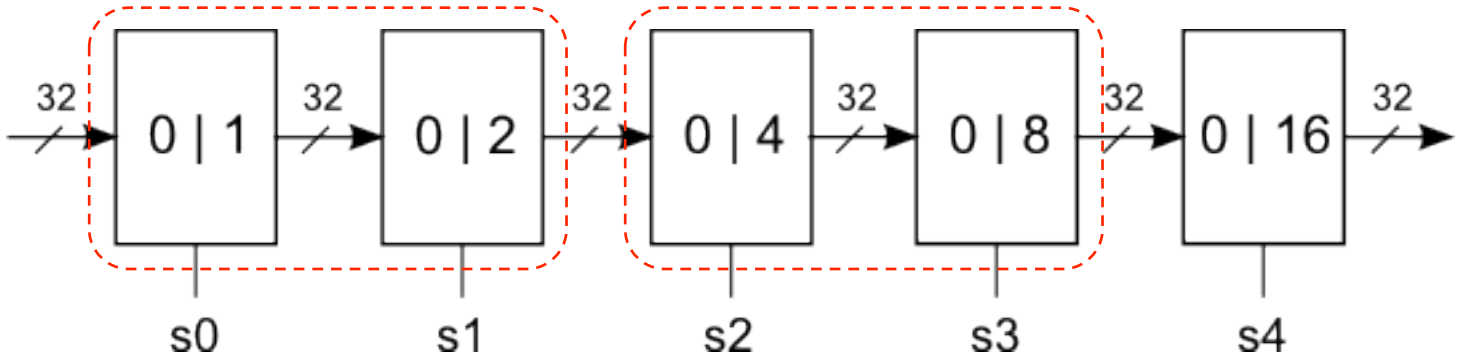
- Worst case delay O(N) , not good for processor design.
- Can we do it in O(logN) time and fit it in one cycle?

Log Shifter / Rotator

- Log(N) stages, each shifts (or not) by a power of 2 places, S=[s_2;s_1;s_0]:

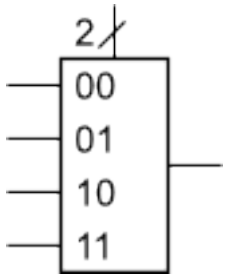


LUT Mapping of Log shifter

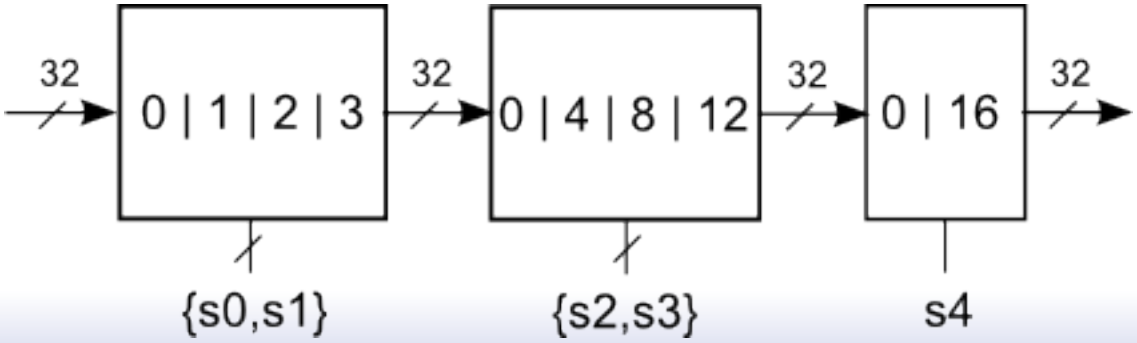


Efficient with 2to1 multiplexors, for instance, 3LUTs.

Virtex6 has 6LUTs. Naturally makes 4to1 muxes:



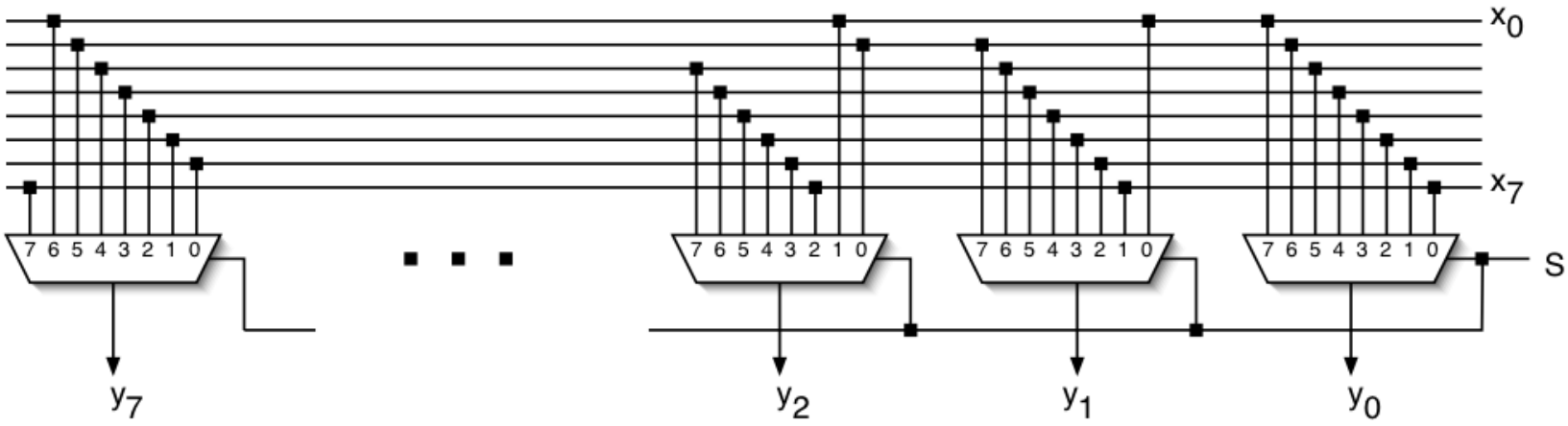
Reorganize shifter to use 4to1 muxes.



Final stage uses F7 mux

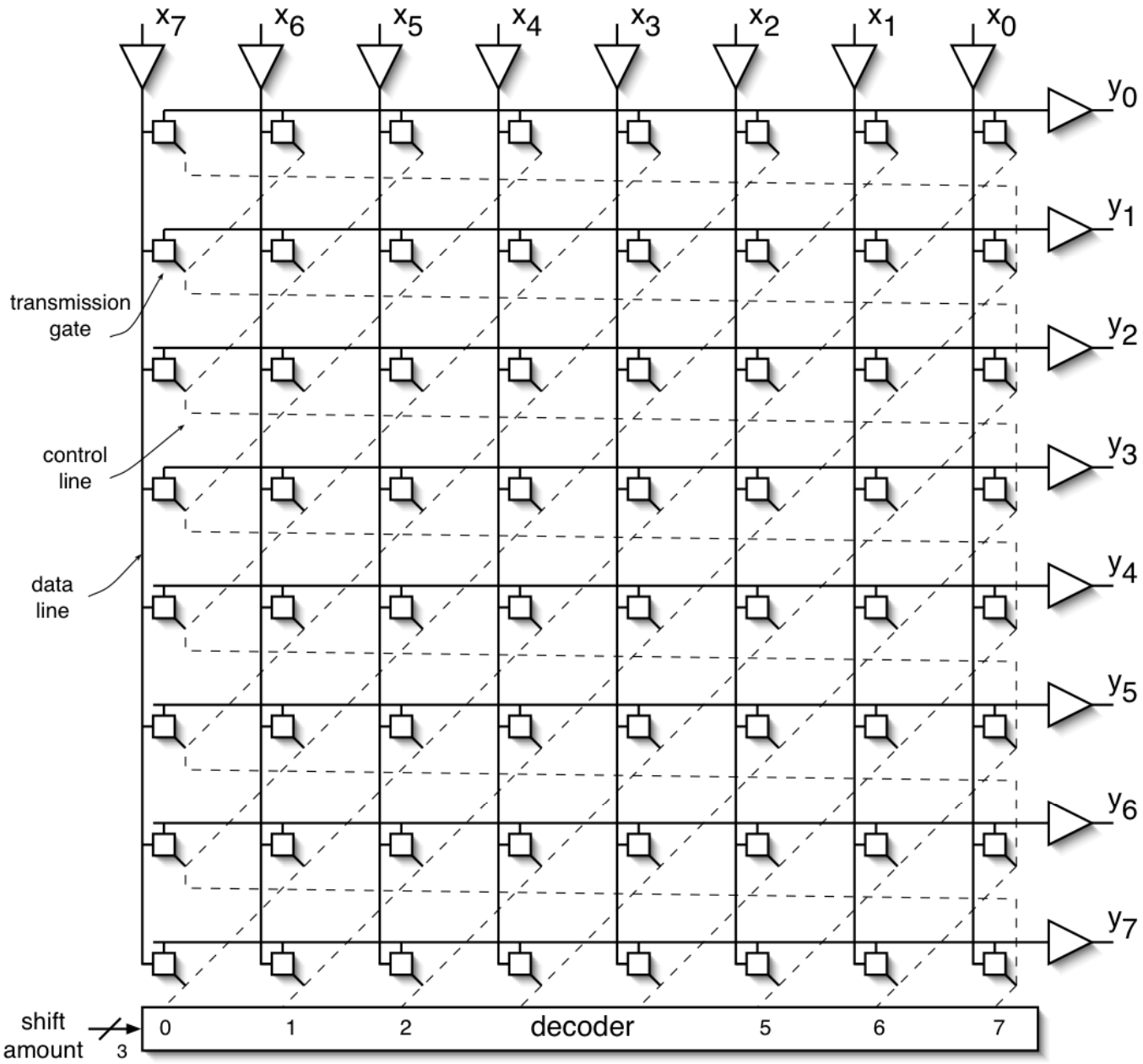
"Improved" Shifter / Rotator

- How about this approach? Could it lead to even less delay?



- What is the delay of these big muxes?
- Look a transistor-level implementation?

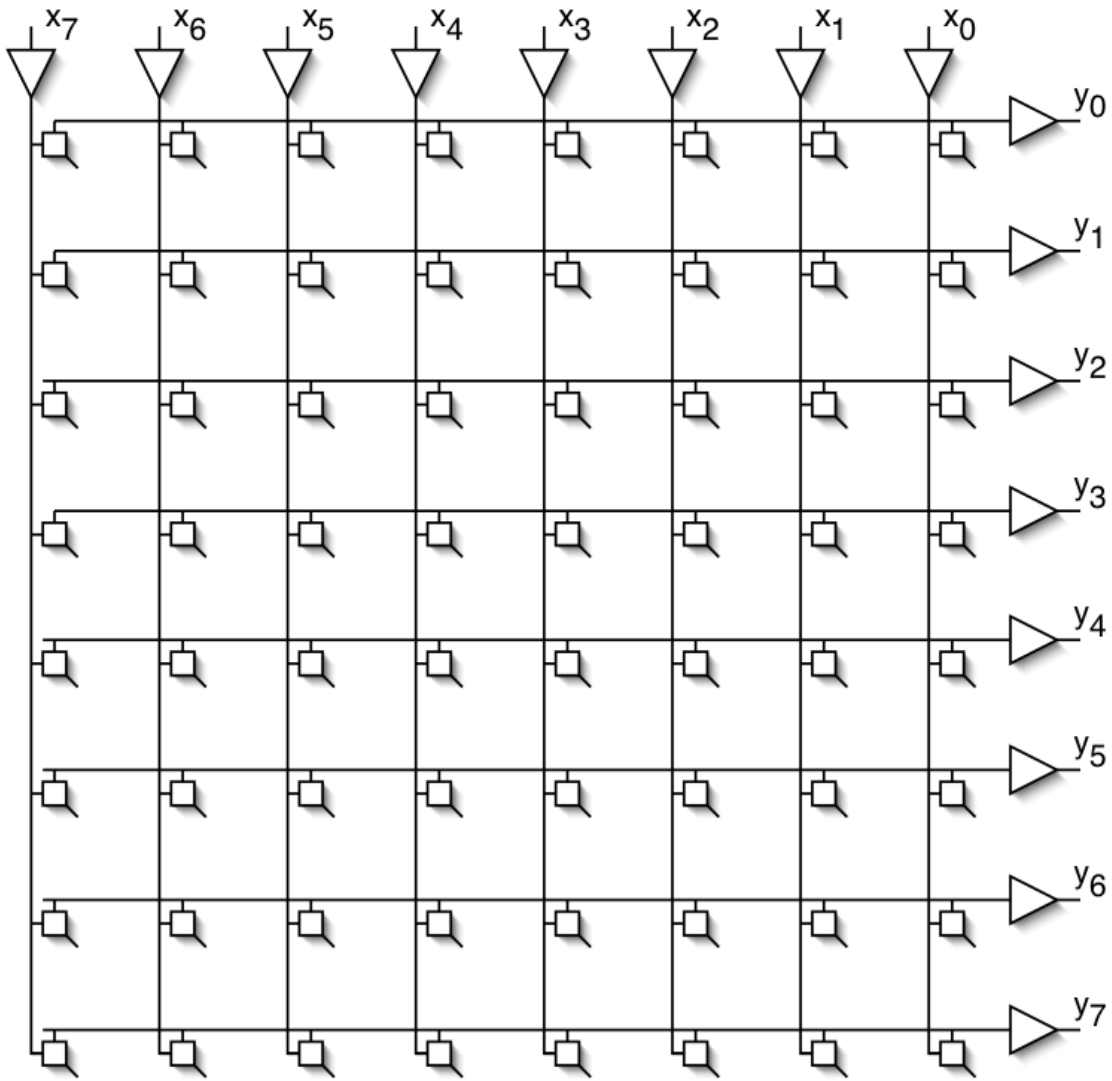
Barrel Shifter



Cost/delay?

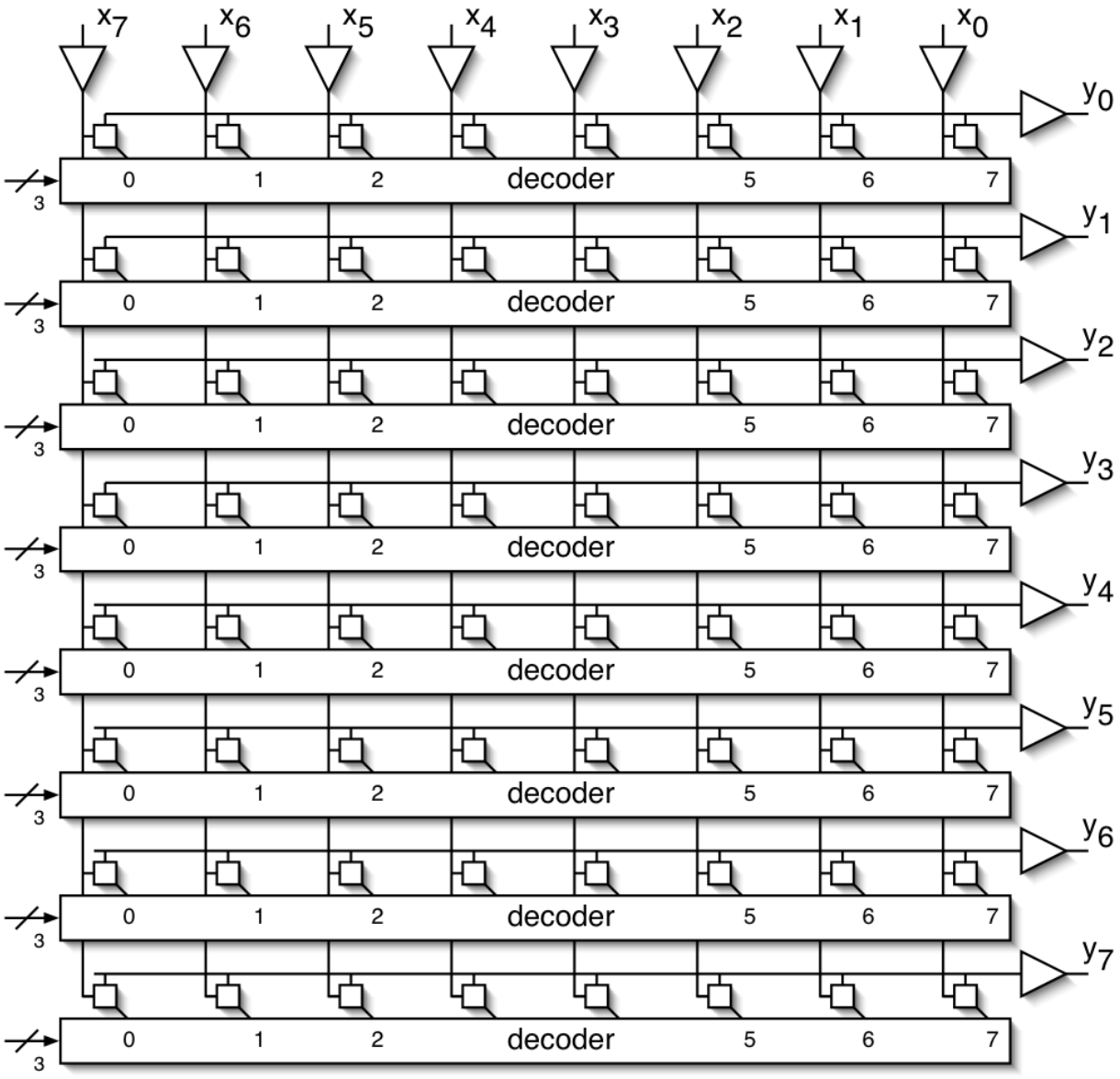
- (don't forget the decoder)

Connection Matrix



- Generally useful structure:
  - N^2 control points.
  - What other interesting functions can it do?

Cross-bar Switch

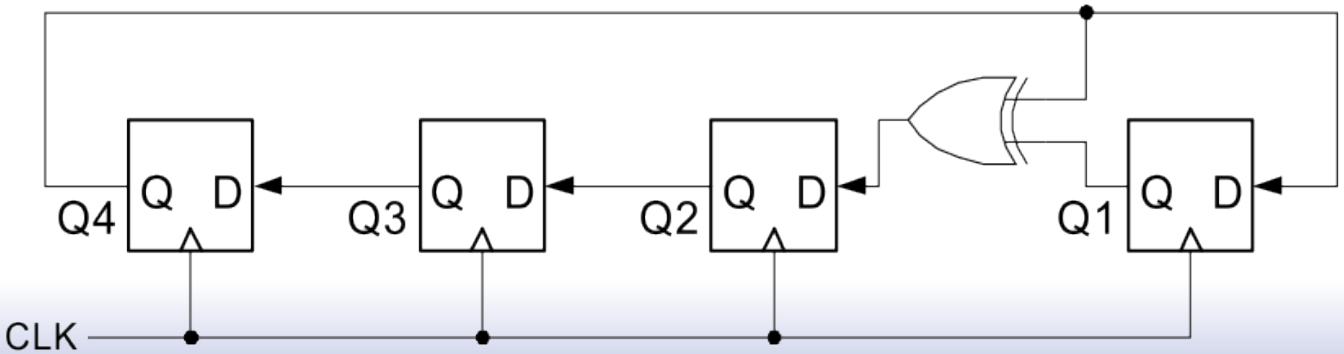


- Nlog(N) control signals.
- Supports all interesting permutations
  - All one-to-one and one-to-many connections.
- Commonly used in communication hardware (switches, routers).

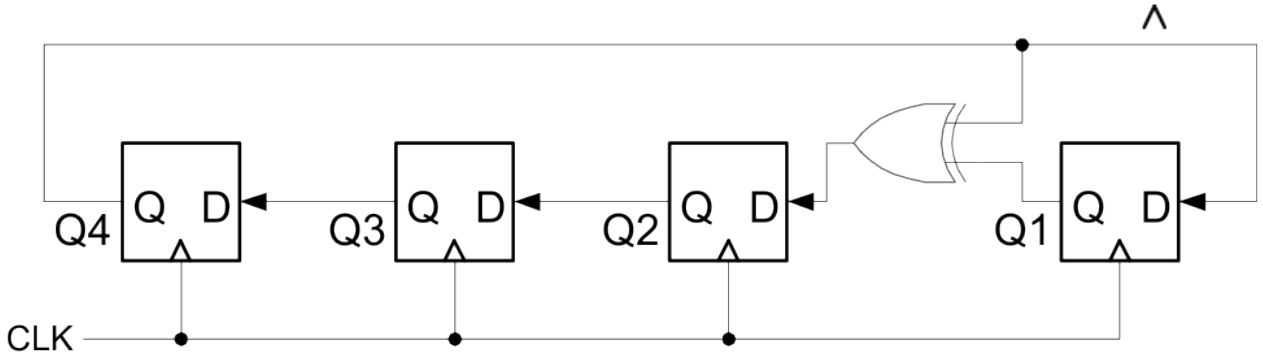
LFSRs

Linear Feedback Shift Registers (LFSRs)

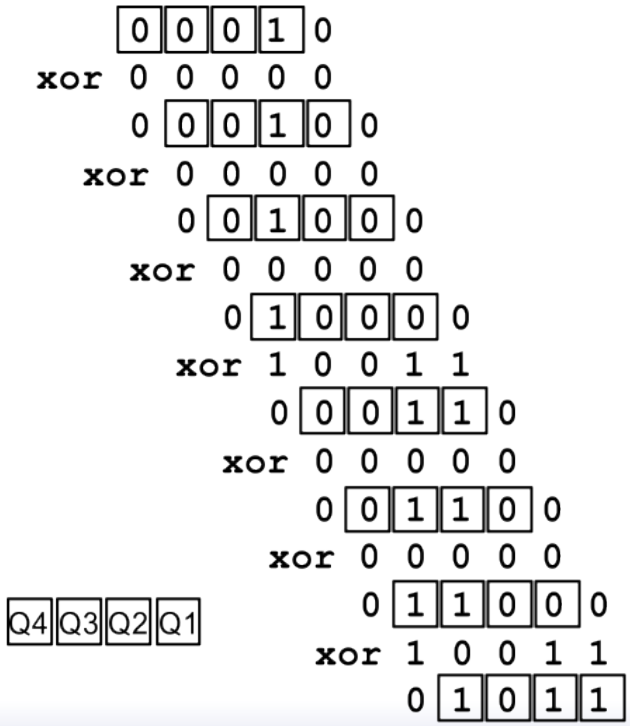
- These are n-bit counters exhibiting pseudo-random behavior.
- Built from simple shift-registers with a small number of xor gates.
- Used for:
  - random number generation
  - counters
  - error checking and correction
- Advantages:
  - very little hardware
  - high speed operation
- Example 4-bit LFSR:



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.)



0001
0010
0100
1000
0011
0110
1100
1011
0101
1010
0111
1110
1111
1101
1001
0001

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:

    $T > T_{2\text{-input-xor}}$ + 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 $x^7 + x^6 + 1$

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

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

$$\begin{array}{r} x^4 + x^3 + \quad + x + 1 \\ + \quad x^4 + \quad + x^2 + x \\ \hline x^3 + x^2 \quad + 1 \end{array}$$

- "Multiply": multiplying by x^n is like shifting to the left.

$$\begin{array}{r} x^2 + x + 1 \\ \cdot \quad x + 1 \\ \hline x^2 + x + 1 \\ x^3 + x^2 + x \\ \hline x^3 \quad + 1 \end{array}$$

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 $GF(2^n)$, n is the degree of the prime polynomial (the modulus)

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

Galois Fields - The theory behind LFSRs

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

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

$$
\begin{aligned}
\alpha^4 &= x^4 \bmod x^4 + x + 1 \\
&= x^4 \text{ xor } x^4 + x + 1 \\
&= x + 1
\end{aligned}
$$

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

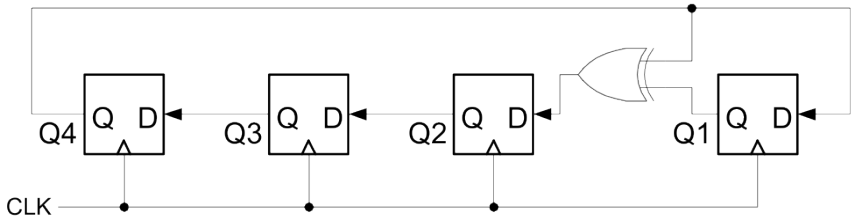
Primitive Polynomials

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

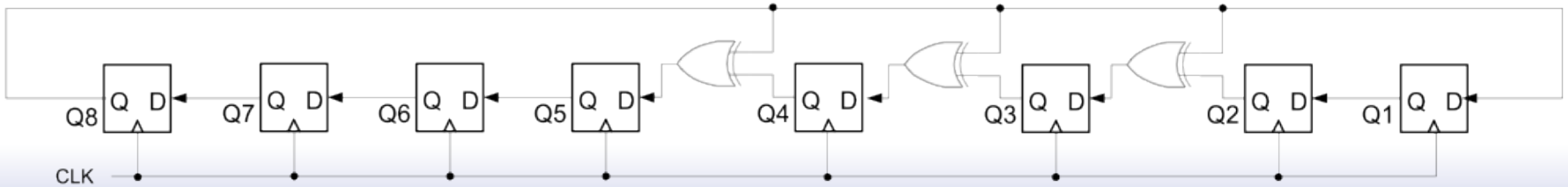
Galois Field		*Hardware*
Multiplication by x	$\Leftrightarrow$	*shift left*
Taking the result mod $p(x)$	$\Leftrightarrow$	*XOR-ing with the coefficients of* $p(x)$ *when the most significant coefficient is 1.*
Obtaining all 2^n-1 *non-zero elements by evaluating* x^k *for* $k = 1, \ldots, 2^n-1$	$\Leftrightarrow$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

Building an LFSR from a Primitive Polynomial

- Find the primitive polynomial of the form $x^k + \ldots + 1$.
- 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.
- The $x^0 = 1$ 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



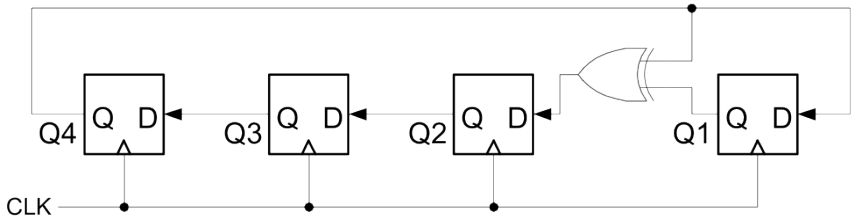
- To build an 8-bit LFSR, use the primitive polynomial $x^8 + x^4 + x^3 + x^2 + 1$ and connect xors between FF2 and FF3, FF3 and FF4, and FF4 and FF5.



Building an LFSR from a Primitive Polynomial (without understanding Galois Fields)

- For a k-bit LFSR lookup the primitive polynomial of the form $x^k + \ldots + 1$.
- For k-bit LFSR number the flip-flops with FF1 on the right.
- Connect a feedback path from the Q output of the leftmost FF directly to the D input of FF 1.
- For each term of the form x^n, except for x^k connect an xor between FF n and n+1.

- 4-bit example, uses $x^4 + x + 1$
  - FF4's Q output feeds back to FF1
  - x ⇔ xor between FF1 and FF2



- 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.