# EECS150 - Digital Design

## Lecture 21 - Design Blocks

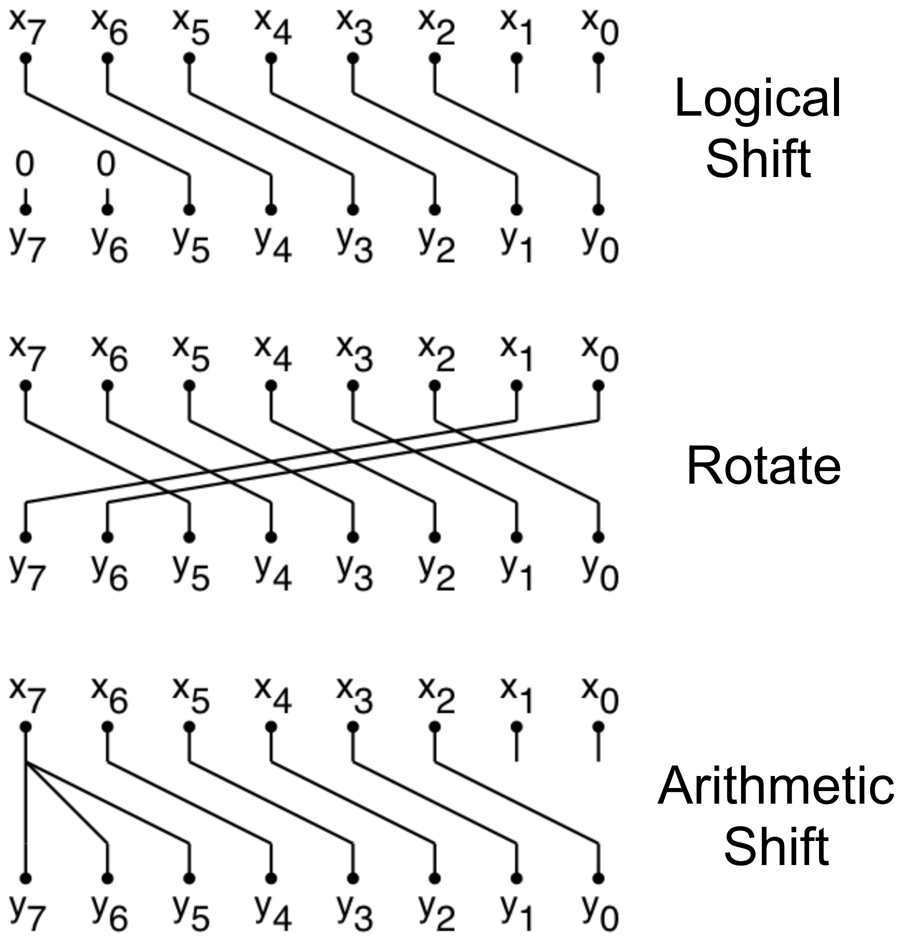
April 3, 2012

John Wawrzynek

# Fixed Shifters / Rotators

- “fixed” shifters “hardwire” the shift amount into the circuit.
- Ex: verilog: X >> 2
  - (right shift X by 2 places)
- Fixed shift/rotator is nothing but wires!

  So what?


$x_7$ $x_6$ $x_5$ $x_4$ $x_3$ $x_2$ $x_1$ $x_0$
0 0
$y_7$ $y_6$ $y_5$ $y_4$ $y_3$ $y_2$ $y_1$ $y_0$
$x_7$ $x_6$ $x_5$ $x_4$ $x_3$ $x_2$ $x_1$ $x_0$
$y_7$ $y_6$ $y_5$ $y_4$ $y_3$ $y_2$ $y_1$ $y_0$
$x_7$ $x_6$ $x_5$ $x_4$ $x_3$ $x_2$ $x_1$ $x_0$
$y_7$ $y_6$ $y_5$ $y_4$ $y_3$ $y_2$ $y_1$ $y_0$

Logical Shift

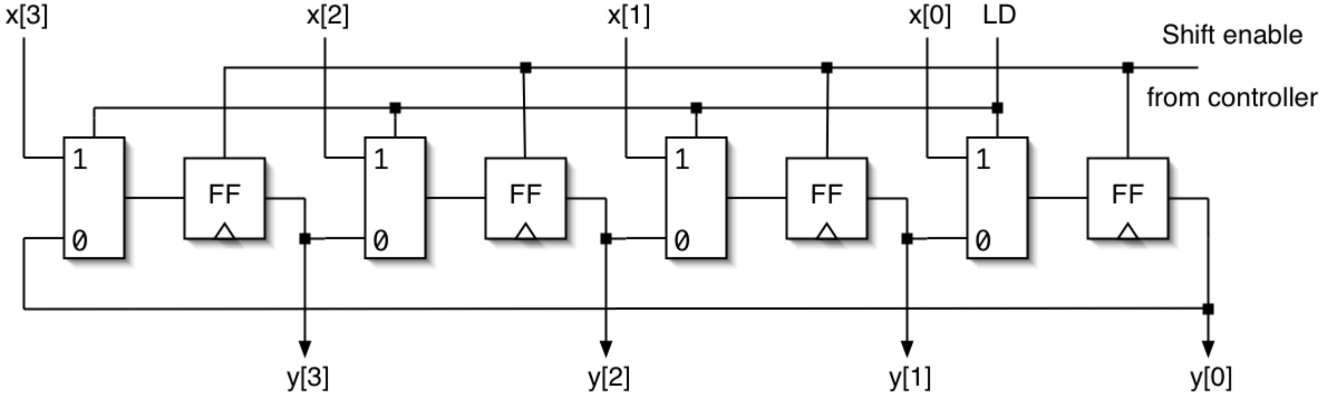
Rotate

Arithmetic Shift

# 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.

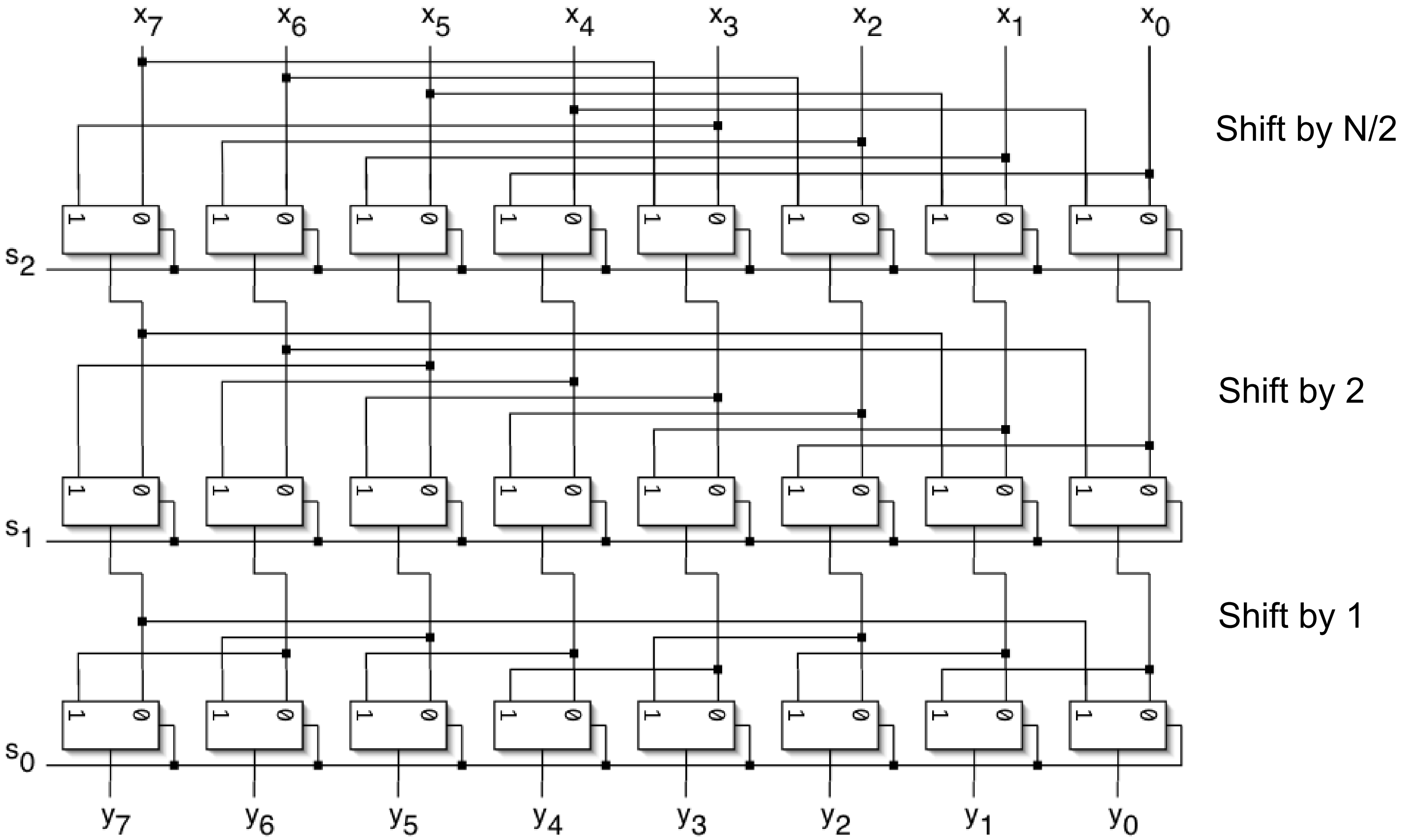


– Worst case delay O(N) , not good for processor design.

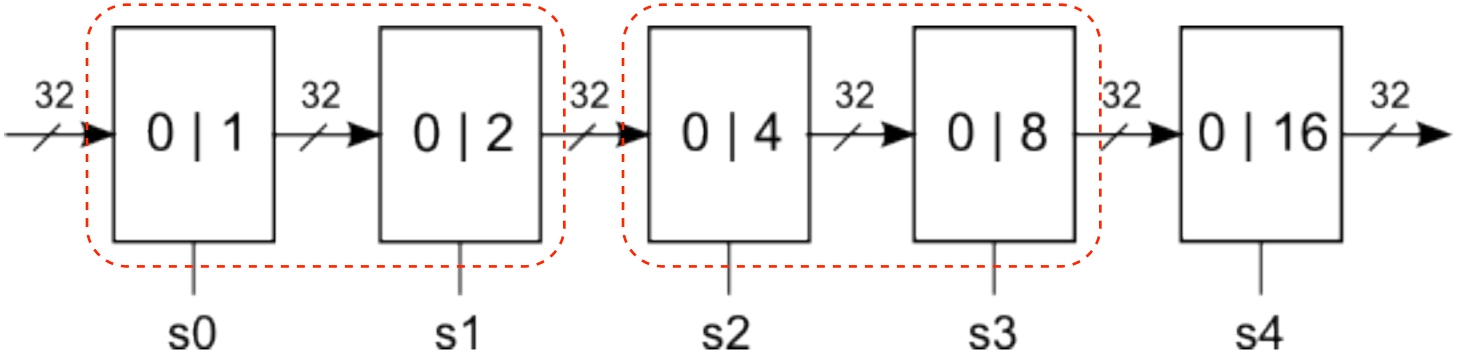
– Can we do it in O(logN) time and fit it in one cycle?

# Log Shifter / Rotator

- Log(N) stages, each shifts (or not) by a power of 2 places,

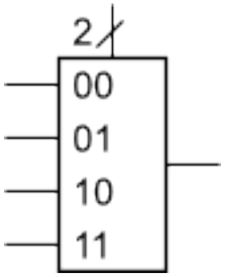


# LUT Mapping of Log shifter

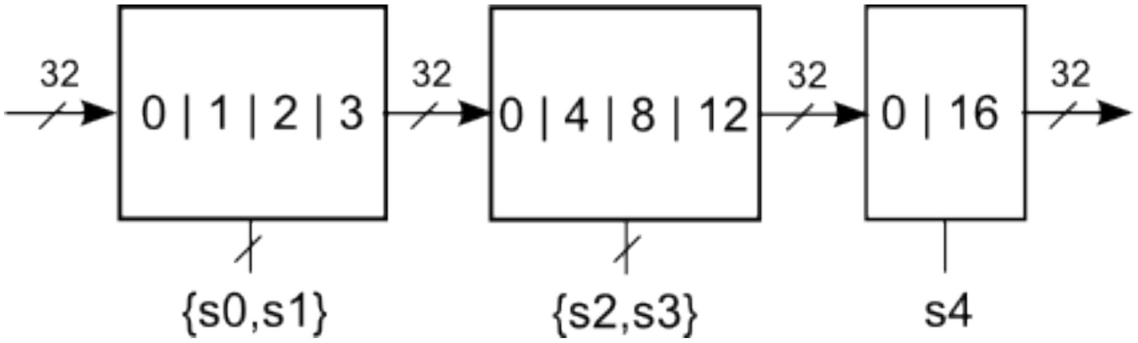


Efficient with 2to1 multiplexors, for instance, 3LUTs.

Virtex6 has 6LUTs. Naturally makes 4to1 muxes:



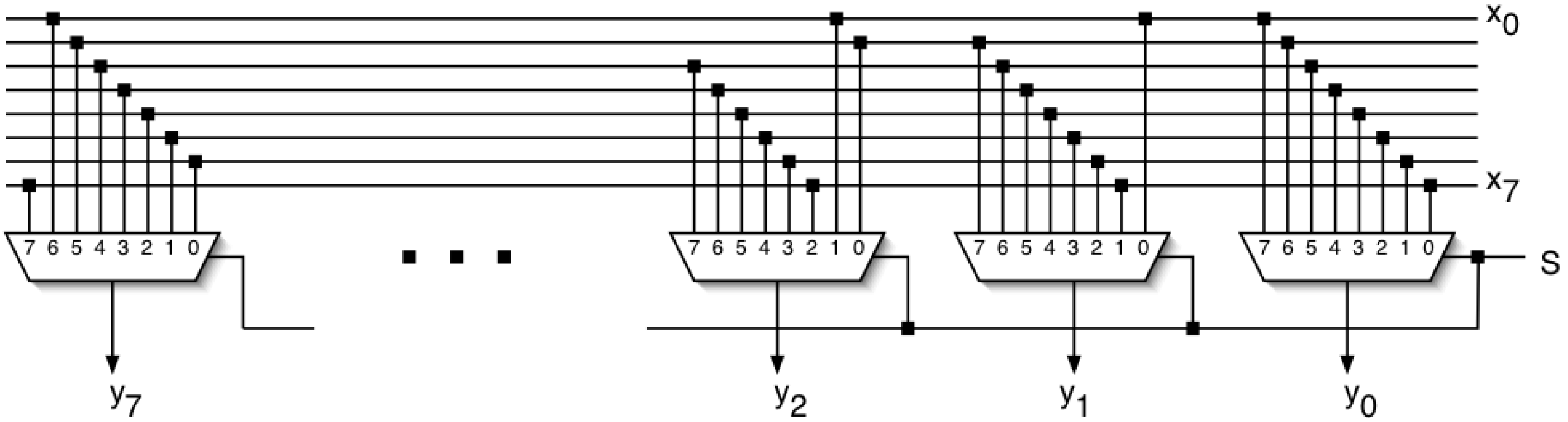
Reorganize shifter to use 4to1 muxes.



Final stage uses F7 mux

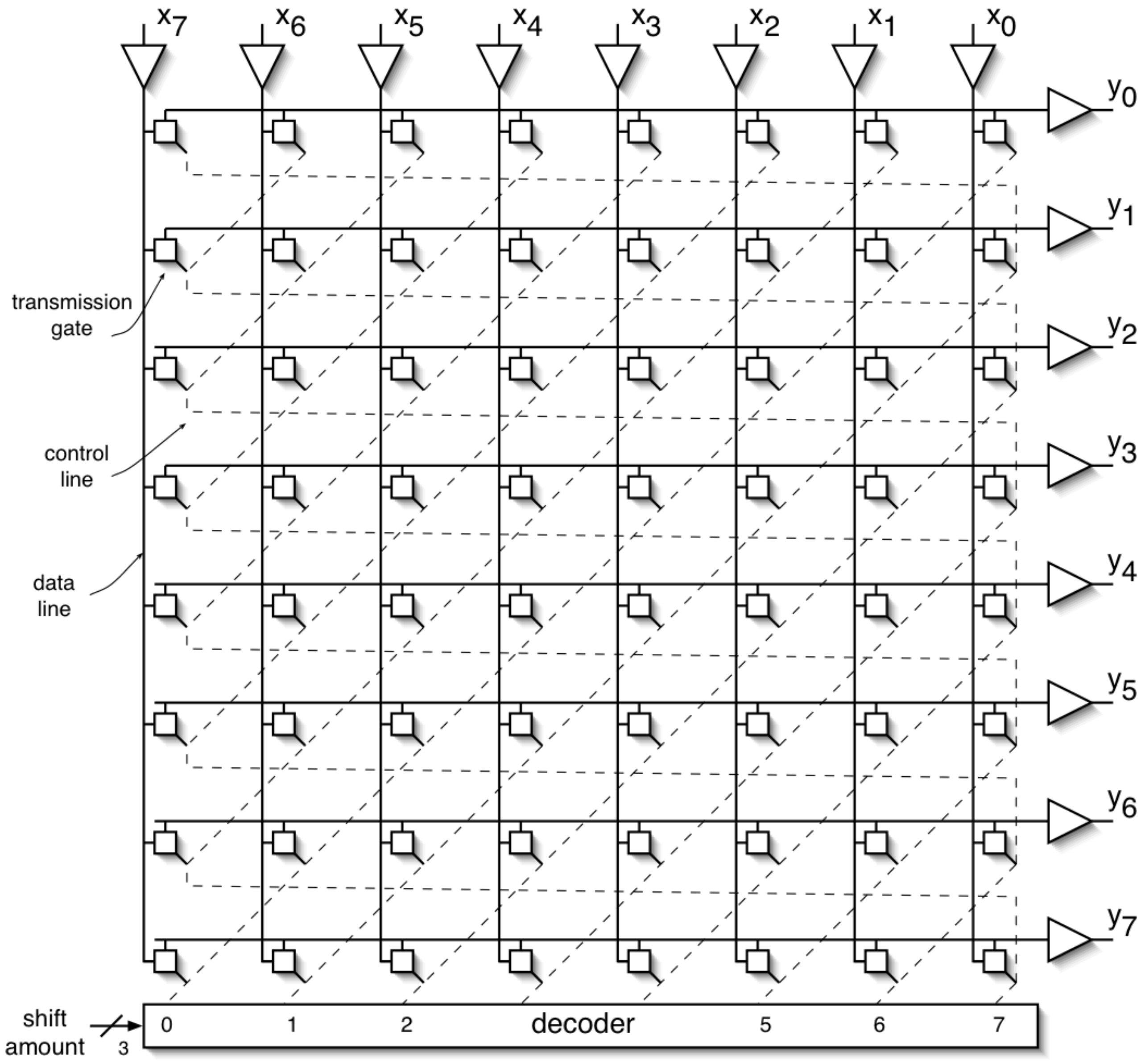
# “Improved” Shifter / Rotator

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



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

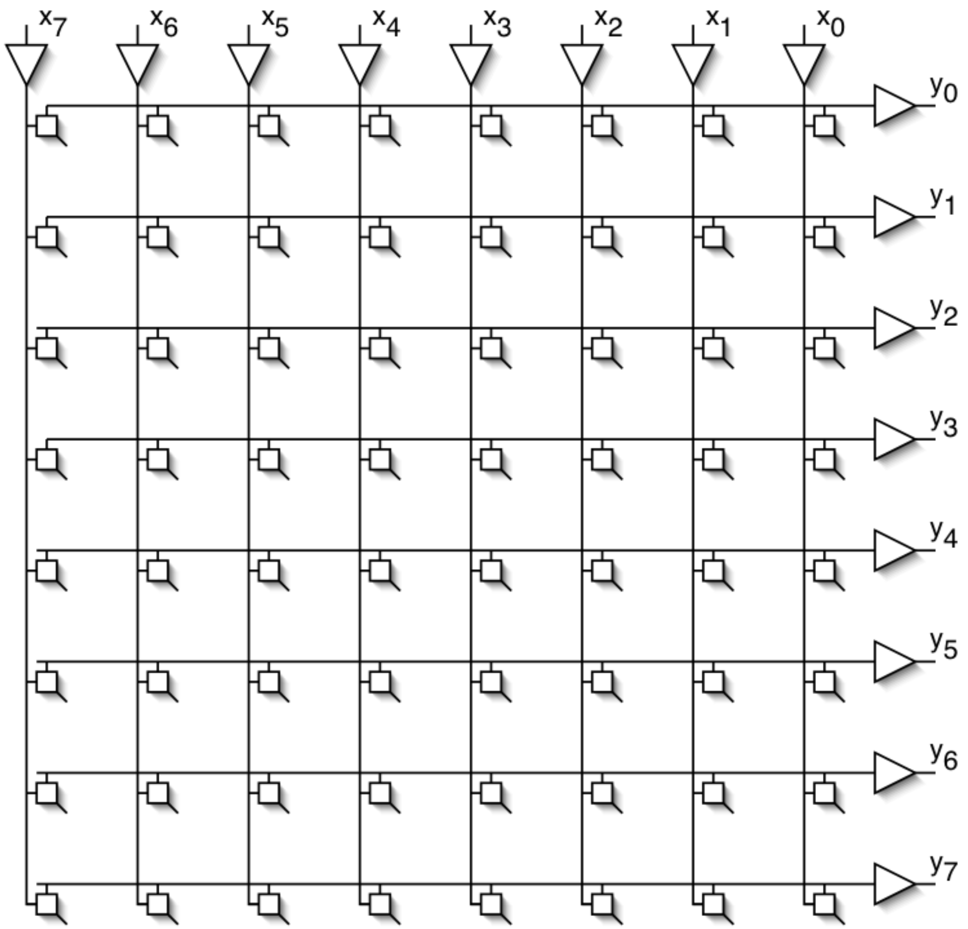
# Barrel Shifter



Cost/delay?

- (don't forget the decoder)

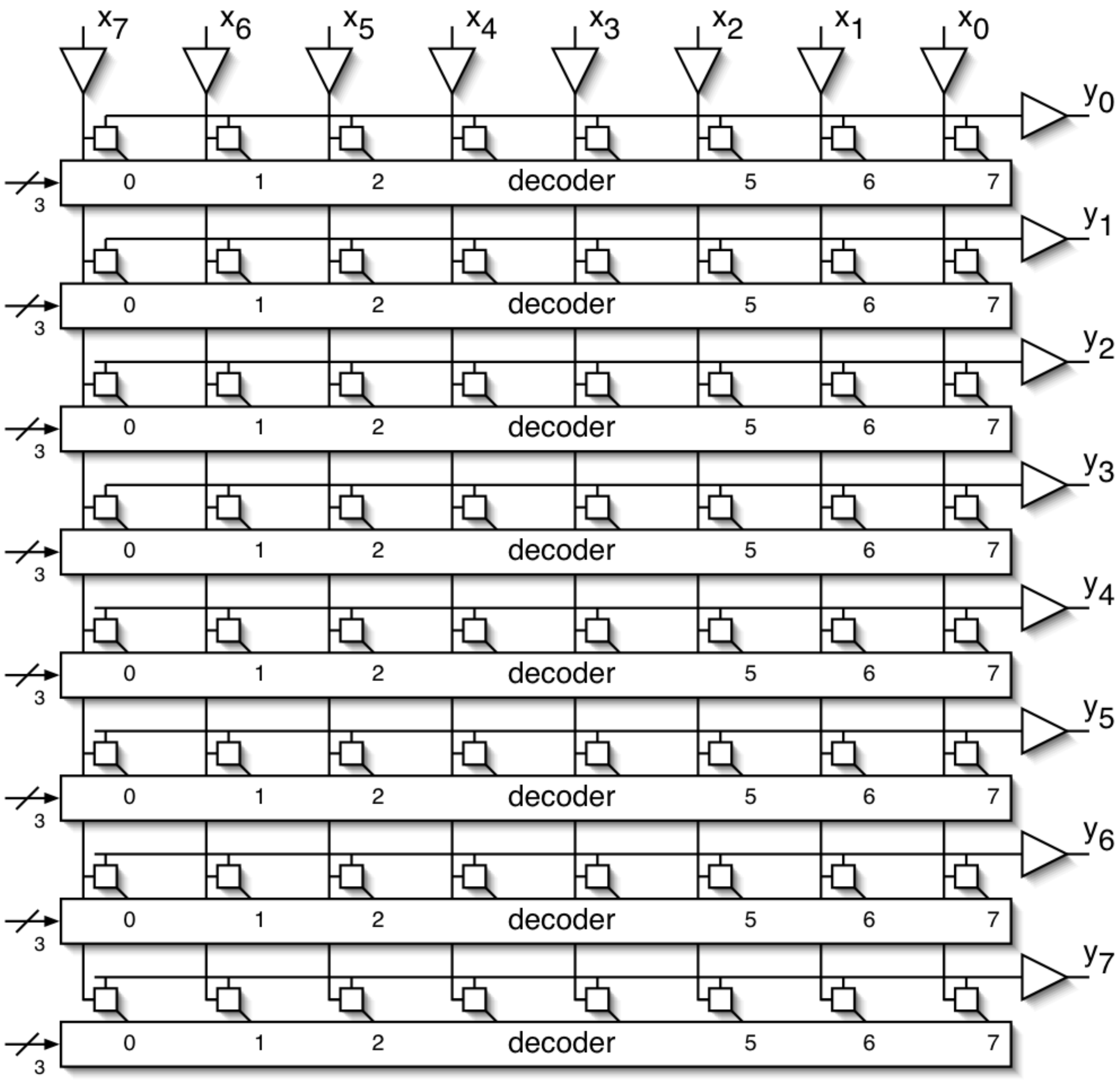
# Connection Matrix



Generally useful structure:

- $N^2$ control points.
- What other interesting functions can it do?

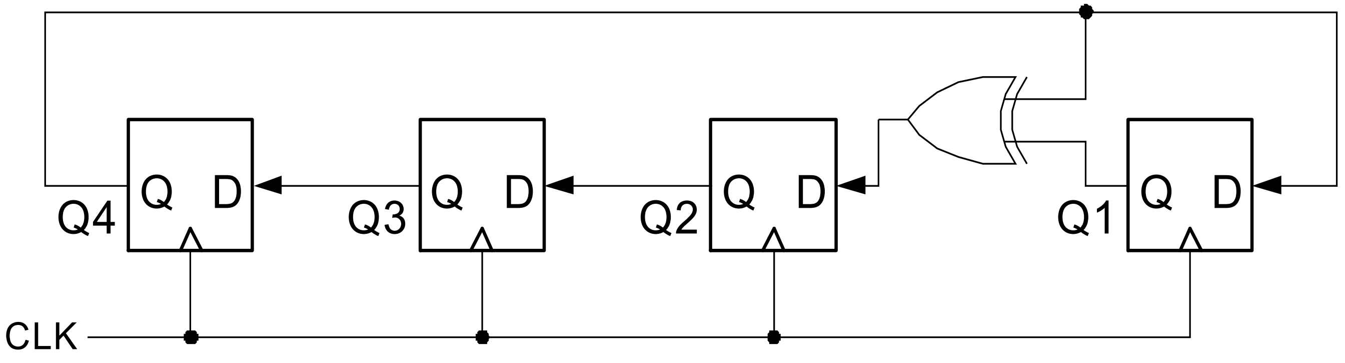
# Cross-bar Switch



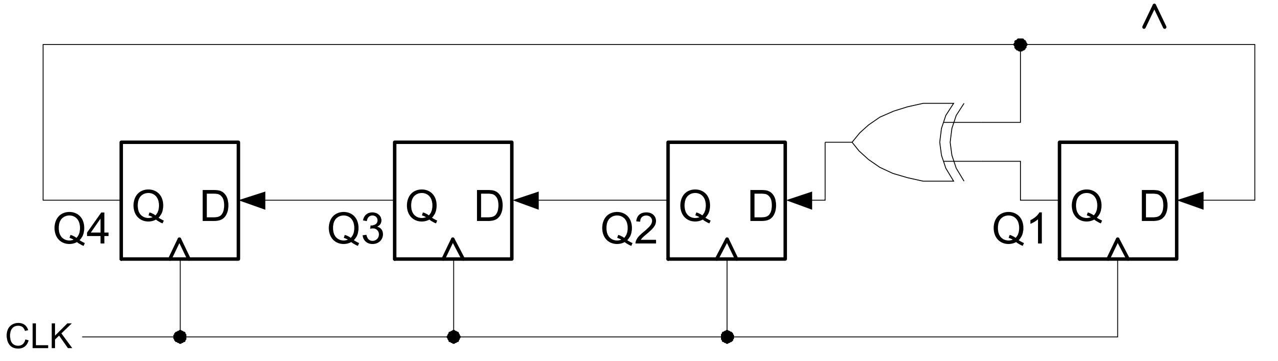
- Nlog(N) control signals.
- Supports all interesting permutations
  - All one-to-one and one-to-many connections.
- Commonly used in communication hardware (switches, routers).

# 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.)

```
    0 0 0 1 0
xor 0 0 0 0 0
    0 0 0 1 0 0
  xor 0 0 0 0 0
      0 0 1 0 0 0
    xor 0 0 0 0 0
        0 1 0 0 0 0
      xor 1 0 0 1 1
          0 0 0 1 1 0
        xor 0 0 0 0 0
            0 0 1 1 0 0
          xor 0 0 0 0 0
              0 1 1 0 0 0
            xor 1 0 0 1 1
                0 1 0 1 1
```

Q4 Q3 Q2 Q1

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:

    $T > T_{\text{2-input-xor}}$ + 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:

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

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

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

# 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.*

# Galois Fields - The theory behind LFSRs

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

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

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

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

# Primitive Polynomials

$x^2 + x + 1$

$x^3 + x + 1$

$x^4 + x + 1$

$x^5 + x^2 + 1$

$x^6 + x + 1$

$x^7 + x^3 + 1$

$x^8 + x^4 + x^3 + x^2 + 1$

$x^9 + x^4 + 1$

$x^{10} + x^3 + 1$

$x^{11} + x^2 + 1$

$x^{12} + x^6 + x^4 + x + 1$

$x^{13} + x^4 + x^3 + x + 1$

$x^{14} + x^{10} + x^6 + x + 1$

$x^{15} + x + 1$

$x^{16} + x^{12} + x^3 + x + 1$

$x^{17} + x^3 + 1$

$x^{18} + x^7 + 1$

$x^{19} + x^5 + x^2 + x + 1$

$x^{20} + x^3 + 1$

$x^{21} + x^2 + 1$

$x^{22} + x + 1$

$x^{23} + x^5 + 1$

$x^{24} + x^7 + x^2 + x + 1$

$x^{25} + x^3 + 1$

$x^{26} + x^6 + x^2 + x + 1$

$x^{27} + x^5 + x^2 + x + 1$

$x^{28} + x^3 + 1$

$x^{29} + x + 1$

$x^{30} + x^6 + x^4 + x + 1$

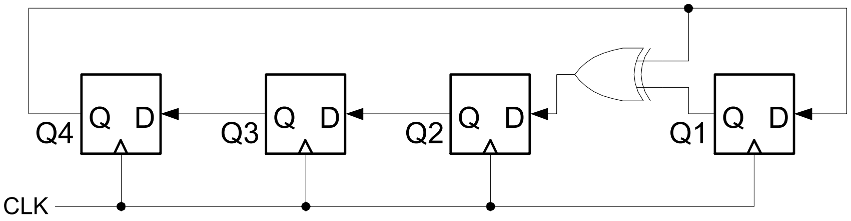
$x^{31} + x^3 + 1$

$x^{32} + x^7 + x^6 + x^2 + 1$

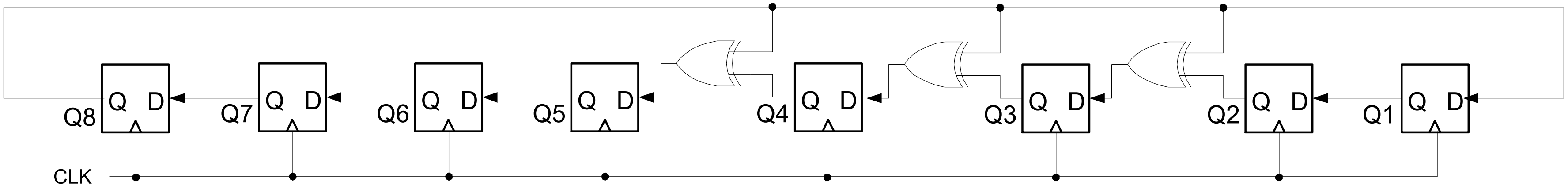
| **Galois Field** | | **Hardware** |
|---|---|---|
| Multiplication by $x$ | ⇔ | shift left |
| Taking the result mod $p(x)$ | ⇔ | XOR-ing with the coefficients of $p(x)$ when the most significant coefficient is 1. |
| Obtaining all $2^n-1$ non-zero elements by evaluating $x^k$ for $k = 1, \ldots, 2^n-1$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

# 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 $x^k + ... + 1$.
- 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

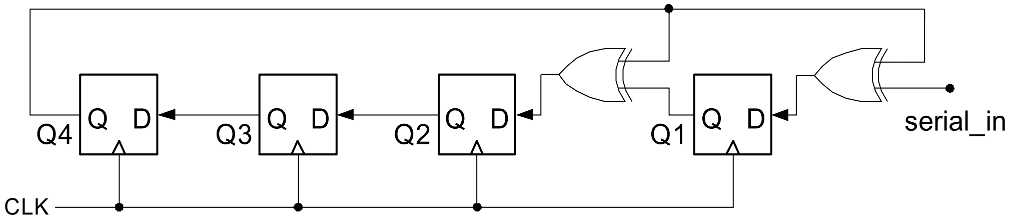


- To build an 8-bit LFSR, use the primitive polynomial $x^8 + x^4 + x^3 + x^2 + 1$ and connect xors between FF2 and FF3, FF3 and FF4, and FF4 and FF5.



# Error Correction with LFSRs

```
              11 message bits   4 check bits
bit sequence:  1 1 0 0 1 0 0 0 1 1 1 0 0 0 0
       0 0 0 0 1
   xor 0 0 0 0 0
       0 0 0 0 1 1
     xor 0 0 0 0 0
         0 0 0 1 1 0
       xor 0 0 0 0 0
           0 0 1 1 0 0
         xor 0 0 0 0 0
             0 1 1 0 0 1
           xor 1 0 0 1 1
               0 1 0 1 0 0
             xor 1 0 0 1 1
                 0 0 1 1 1
                                . . . . . . . . . . .
                                       1 0 1 0
```



# 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 patters 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.

$$1\,1\,0\,0\,1\,0\,0\,0\,1\,1\,1\quad 0\,0\,0\,0 \;\Rightarrow\; 1\,0\,1\,0$$

$$1\,1\,0\,0\,1\,0\,0\,0\,1\,1\,1\quad 1\,0\,1\,0 \;\Rightarrow\; 0\,0\,0\,0$$

- 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.