Cross-coupled NOR gates

- If both $R=0$ and $S=0$, then cross-coupled NORs are equivalent to a stable latch:

<table>
<thead>
<tr>
<th>NOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

- If either $R$ or $S$ becomes $=1$, then the state may change:

  - Case 1: $R=0$, $S=0$, $Q$ changes
  - Case 2: $R=0$, $S=0$, $Q'$ changes

- What happens if $R$ or $S$ or both become $=1$?
Asynchronous State Transition Diagram

Transitions triggered by input changes.

SR Latch:

<table>
<thead>
<tr>
<th>SR</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>hold</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>indeterminate</td>
</tr>
</tbody>
</table>

- S is “set” input
- R is “reset” input

QQ’=00 is often called a “forbidden state”

Nand-gate based SR latch

(a) Logic diagram

(b) Function table

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>Q’</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1 (after S = 1, R = 0)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0 (after S = 0, R = 1)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 5-4 SR Latch with NAND Gates

- Same behavior as cross-coupled NORs with inverted inputs.
Level-sensitive SR Latch

- The input “C” works as an “enable” signal, latch only changes output when C is high.
- Usually connected to clock.
- Generally, it is not a good idea to use a clock as a logic signal (into gates etc.). This is a special case.

D-latch

Compare to transistor version:

All state elements could be built using logic gates.
Flip-flops

- Add logic to eliminate "indeterminate" action of RS FF.
- New action is "toggle"
- J = "jam"
- K = "kill"

J-K FF

<table>
<thead>
<tr>
<th>J</th>
<th>K</th>
<th>Q(t)</th>
<th>Q(t+Δ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>reset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>set</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>toggle</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
# Storage Element Taxonomy

<table>
<thead>
<tr>
<th></th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Level-sensitive</td>
<td>Edge-triggered</td>
</tr>
<tr>
<td>D-type</td>
<td>★</td>
<td>✔</td>
</tr>
<tr>
<td>JK-type</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>RS-type</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

★ “natural” form
✔ “possible” form

## Design Example with RS FF

- With D-type FF state elements, new state is computed based on inputs & present state bits - reloaded each cycle.
- With RS (or JK) FF state elements, inputs are used to determine conditions under which to set or reset state bits.
- Example: bit-serial adder (LSB first)

![Diagram](image-url)
Bit-serial adder with RS FF

- RS FF stores the carry:

<table>
<thead>
<tr>
<th>a b c_i</th>
<th>c_{i+1}</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0</td>
<td>Carry kill a'b'</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 0</td>
<td>Carry generate ab</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1</td>
<td></td>
</tr>
</tbody>
</table>

**FIFOs**
First-in-first-out (FIFO) Memory

• Used to implement queues.
• These find common use in computers and communication circuits.
• Generally, used for rate matching data producer and consumer:

  - Producer can perform many writes without consumer performing any reads (or vice versa). However, because of finite buffer size, on average, need equal number of reads and writes.
• Typical uses:
  - interfacing I/O devices. Example network interface. Data bursts from network, then processor bursts to memory buffer (or reads one word at a time from interface). Operations not synchronized.
  - Example: Audio output. Processor produces output samples in bursts (during process swap-in time). Audio DAC clocks it out at constant sample rate.

FIFO Interfaces

• After write or read operation, FULL and EMPTY indicate status of buffer.
• Used by external logic to control own reading from or writing to the buffer.
• FIFO resets to EMPTY state.
• HALF FULL (or other indicator of partial fullness) is optional.

• Address pointers are used internally to keep next write position and next read position into a dual-port memory.

  - If pointers equal after write ⇒ FULL:
  - If pointers equal after read ⇒ EMPTY:
FIFO Implementation Details

- Assume, dual-port memory with asynchronous read, synchronous write.
- Binary counter for each of read and write address. CEs controlled by WE and RE.
- Equal comparator to see when pointers match.
- Flip-flop each for FULL and EMPTY flags:

<table>
<thead>
<tr>
<th>WE</th>
<th>RE</th>
<th>equal</th>
<th>EMPTY&lt;sub&gt;i&lt;/sub&gt;</th>
<th>FULL&lt;sub&gt;i&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>EMPTY&lt;sub&gt;i-1&lt;/sub&gt;</td>
<td>FULL&lt;sub&gt;i-1&lt;/sub&gt;</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>EMPTY&lt;sub&gt;i-1&lt;/sub&gt;</td>
<td>FULL&lt;sub&gt;i-1&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

- Control logic with truth-table shown to left.

Xilinx Virtex5 FIFOs

- Virtex5 BlockRAMS include special logic for FIFOs.
- Details in User Guide (ug190).
- Take advantage of separate dual ports and independent ports clocks.
Error Correction Codes (ECC)

• Memory systems generate errors (accidentally flipped-bits)
  – DRAMs store very little charge per bit
  – “Soft” errors occur occasionally when cells are struck by alpha particles or other environmental upsets.
  – Less frequently, “hard” errors can occur when chips permanently fail.

• Where “perfect” memory is required
  – servers, spacecraft/military computers, …

• Memories are protected against failures with ECCs

• Extra bits are added to each data-word
  – extra bits are used to detect and/or correct faults in the memory system
  – in general, each possible data word value is mapped to a unique “code word”. A fault changes a valid code word to an invalid one - which can be detected.
Simple Error Detection Coding

Parity Bit

- Each data value, before it is written to memory is “tagged” with an extra bit to force the stored word to have even parity:

\[ b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 p \]

- Each word, as it is read from memory is “checked” by finding its parity (including the parity bit):

\[ b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 p \]

- A non-zero parity indicates an error occurred:
  - two errors (on different bits) is not detected (nor any even number of errors)
  - odd numbers of errors are detected.

Hamming Error Correcting Code

- Use more parity bits to pinpoint bit(s) in error, so they can be corrected.

- Example: Single error correction (SEC) on 4-bit data
  - use 3 parity bits, with 4-data bits results in 7-bit code word
  - 3 parity bits sufficient to identify any one of 7 code word bits
  - overlap the assignment of parity bits so that a single error in the 7-bit word can be corrected

- Procedure: group parity bits so they correspond to subsets of the 7 bits:
  - \( p_1 \) protects bits 1,3,5,7
  - \( p_2 \) protects bits 2,3,6,7
  - \( p_3 \) protects bits 4,5,6,7

<table>
<thead>
<tr>
<th>Bit position number</th>
<th>001 = 1(_{10})</th>
<th>011 = 3(_{10})</th>
<th>101 = 5(_{10})</th>
<th>111 = 7(_{10})</th>
<th>010 = 2(_{10})</th>
<th>011 = 3(_{10})</th>
<th>110 = 6(_{10})</th>
<th>111 = 7(_{10})</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1 )</td>
<td>( p_2 )</td>
<td>( p_3 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note: number bits from left to right.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hamming Code Example

1 2 3 4 5 6 7
p_1  p_2  d_1  p_3  d_2  d_3  d_4

- Note: parity bits occupy power-of-two bit positions in code-word.
- On writing to memory:
  - parity bits are assigned to force even parity over their respective groups.
- On reading from memory:
  - check bits (c_3,c_2,c_1) are generated by finding the parity of the group and its parity bit. If an error occurred in a group, the corresponding check bit will be 1, if no error the check bit will be 0.
  - check bits (c_3,c_2,c_1) form the position of the bit in error.

  Example: c = c_3c_2c_1 = 101
  - error in 4,5,6, or 7 (by c_3=1)
  - error in 1,3,5, or 7 (by c_1=1)
  - no error in 2, 3, 6, or 7 (by c_2=0)
  - Therefore error must be in bit 5.
  - Note the check bits point to 5

  By our clever positioning and assignment of parity bits, the check bits always address the position of the error!

  c=000 indicates no error

  Review procedure with example

Hamming Error Correcting Code

- Overhead involved in single error correction code:
  - let p be the total number of parity bits and d the number of data bits in a p + d bit word.
  - If p error correction bits are to point to the error bit (p + d cases) plus indicate that no error exists (1 case), we need:
    \[ 2^p \geq p + d + 1, \]
    thus \( p \geq \log(p + d + 1) \)
    for large \( d \), \( p \) approaches \( \log(d) \)
  - Adding on extra parity bit covering the entire word can provide double error detection

1 2 3 4 5 6 7 8
p_1  p_2  d_1  p_3  d_2  d_3  d_4  p_4

  On reading the C bits are computed (as usual) plus the parity over the entire word, P:

C=0  P=0, no error
C!=0 P=1, correctable single error
C!=0 P=0, a double error occurred
C=0  P=1, an error occurred in p_4 bit

Typical modern codes in DRAM memory systems:
64-bit data blocks (8 bytes) with 72-bit code words (9 bytes).
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:
    \[ T > T_{\text{2-input-xor}} + \text{clock overhead} \]
  - 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 perform 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:
  $$\begin{array}{c}
x^4 + x^3 + x + 1 \\
+ x^4 + x^2 + x \\
\hline
x^3 + x^2 + 1
\end{array}$$
- “Multiply”: multiplying by $x^n$ is like shifting to the left.
Galois Fields - The theory behind LFSRs

- These polynomials form a \textit{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 \textit{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 \( GF(16) \). \textit{Example on next slide.}

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

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

\[ \begin{align*}
\alpha^4 &= x^4 \ mod \ x^4 + x + 1 \\
&= x^4 \ xor \ x^4 + x + 1 \\
&= x + 1
\end{align*} \]

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

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

**Galois Field**

- 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 \(k = 1, \ldots, 2^n - 1\)

### 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 \(x^0 = 1\) 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 \(x^4 + x + 1\)
  - \(x^4 ⇔ FF4’s\) Q output
  - \(x ⇔\) XOR between FF1 and FF2
  - \(I ⇔ FF1’s\) 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.