MOS Transistors

- Silicon substrate doped with impurities
- Adding or cutting away insulating glass (SiO₂)
- Adding wires made of polycrystalline silicon (polysilicon, poly) or metal, insulated from the substrate by SiO₂

MOS Transistor Switches

N-switch

\[
\begin{array}{c}
S = 0 \\
0 \\
S = 1 \\
1 (degraded)
\end{array}
\]

Good 0, Poor 1
MOS Transistor Switches

P-switch

CMOS switch

(Transmission gate)

Signal Strength

- *Strength* of signal
  - How close it approximates ideal voltage source
- $V_{DD}$ and GND rails are strongest 1 and 0
- nMOS pass strong 0
  - But degraded or weak 1
- pMOS pass strong 1
  - But degraded or weak 0
- Thus nMOS are best for pull-down network
Pass Transistors

- Transistors can be used as switches

\[
\begin{array}{ccc}
\text{g} & \text{s} & \text{d} \\
\text{g} = 0 & \text{Input} & \text{Output} \\
\text{g} = 1 & \text{g} = 1 & \text{strong 0} \\
\text{g} = 0 & \text{g} = 1 & \text{ degraded 1} \\
\text{g} = 1 & \text{g} = 0 & \text{output} \\
\text{g} = 0 & \text{g} = 0 & \text{strong 1} \\
\end{array}
\]

Transmission Gates

- Pass transistors produce degraded outputs
- *Transmission gates* pass both 0 and 1 well

\[
\begin{array}{ccc}
\text{g} & \text{a} & \text{b} \\
\text{g} = 0, \text{gb} = 1 & \text{Input} & \text{Output} \\
\text{g} = 1, \text{gb} = 0 & \text{g} = 1, \text{gb} = 0 & \text{strong 0} \\
\text{g} = 1, \text{gb} = 0 & \text{g} = 1, \text{gb} = 0 & \text{strong 1} \\
\end{array}
\]
Complementary CMOS

- Complementary CMOS logic gates
  - nMOS pull-down network
  - pMOS pull-up network
  - a.k.a. static CMOS

<table>
<thead>
<tr>
<th></th>
<th>Pull-up OFF</th>
<th>Pull-up ON</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pull-down OFF</td>
<td>Z (float)</td>
<td>1</td>
</tr>
<tr>
<td>Pull-down ON</td>
<td>0</td>
<td>X (not allowed)</td>
</tr>
</tbody>
</table>

Series and Parallel

- nMOS: 1 = ON
- pMOS: 0 = ON
- Series: both must be ON
- Parallel: either can be ON
Conduction Complement

- Complementary CMOS gates always produce 0 or 1
- Ex: NAND gate
  - Series nMOS: \( Y = 0 \) when both inputs are 1
  - Thus \( Y = 1 \) when either input is 0
  - Requires parallel pMOS

- Rule of Conduction Complements
  - Pull-up network is complement of pull-down
  - Parallel -> series, series -> parallel

---

CMOS Logic Gates-1

<table>
<thead>
<tr>
<th>Inverter</th>
<th>2-input NAND</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Inverter Diagram" /></td>
<td><img src="image2" alt="2-input NAND Diagram" /></td>
</tr>
</tbody>
</table>

### Pull-up network truth table

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Z</td>
</tr>
</tbody>
</table>

### Pull-down network truth table

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### NAND truth table

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
CMOS Logic Gates-2

2-input NOR

Pull-down truth table
\[
\begin{array}{c|c|c}
\text{a} & \text{b} & \text{z} \\
0 & 0 & Z \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 0 \\
\end{array}
\]

Pull-up truth table
\[
\begin{array}{c|c|c}
\text{a} & \text{b} & \text{z} \\
0 & 0 & 1 \\
0 & 1 & Z \\
1 & 0 & Z \\
1 & 1 & Z \\
\end{array}
\]

• There is always (for all input combinations) a path from either 1 or 0 to the output
• No direct path from 1 to 0 (low power dissipation)
• Fully restored logic
• No ratio-ing is necessary (ratio-less logic)
• Generalize to n-input NAND and n-input NOR?

CMOS Compound (Complex) Gates-1

• What function is implemented by this circuit?
Compound Gates-2

How to implement \( F = ab + bc + ca \)?

\[
\begin{align*}
F &= ab + bc + ca \\
&= \overline{a} \cdot b \cdot c + a \cdot \overline{b} \cdot c + a \cdot b \cdot \overline{c}
\end{align*}
\]

Compound Gates

- *Compound gates* can do any inverting function

- Ex:

\[
Y = \overline{A} \cdot B + C \cdot D \quad \text{(AND-AND-OR-INVERT, AOI22)}
\]

- (a)
- (b)
- (c)
- (d)
- (e)
- (f)
Example: O3AI

\[ Y = (A + B + C) \cdot D \]

- Transmission gate implementation (4 transistors)
- Assume \( \overline{s} \) is available

\[ F = a \overline{s} + b \overline{s} \]

- Complex gate implementation based on \( F = a \overline{s} + b \overline{s} \) requires 10 transistors
And-Or-Invert (AOI) Gates

Or-And-Invert (OAI) Gate

• Generally, complex CMOS gates can be derived directly from maxterms of the function (as in a Karnaugh map)
Designing Complex Gates Using K-Maps

\[ F = ab + bd + bc \]
\[ F = b(a+d+c) \]

\[ F = b + \bar{a}cd \] (how many transistors?)

Graph Models

How to generate pull-up circuit from the pull-down circuit?

- **Draw pull-down graph for pull-down circuit**
  - Every vertex is a source-drain connection
  - Every edge is an nMOS transistor

- **Generate pull-up graph from the pull-down graph**
  - Add vertex for every “region” of pull-down graph
  - Add edge between vertices lying in adjacent “regions”
  - Pull-up circuit corresponds to pull-up graph
Graph Models

- Pull-down circuit
  - a --| |-- I_1
  - b     | |-- I_2
  - c
  - Gnd

- Pull-up circuit
  - a
  - b
  - c

- Tristates
  - Tristate buffer produces Z when not enabled

<table>
<thead>
<tr>
<th>EN</th>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Nonrestoring Tristate

- Transmission gate acts as tristate buffer
  - Only two transistors
  - But nonrestoring
    - Noise on A is passed on to Y

Tristate Inverter

- Tristate inverter produces restored output
  - Violates conduction complement rule
  - Because we want a Z output
Multiplexers

• 2:1 multiplexer chooses between two inputs

<table>
<thead>
<tr>
<th>S</th>
<th>D1</th>
<th>D0</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>

Gate-Level Mux Design

\[ Y = SD_1 + \overline{SD}_0 \] (too many transistors)

• How many transistors are needed? 20
Transmission Gate Mux

- Nonrestoring mux uses two transmission gates
  - Only 4 transistors

Inverting Mux

- Inverting multiplexer
  - Use compound AOI22
  - Or pair of tristate inverters
  - Essentially the same thing

- Noninverting multiplexer adds an inverter
CMOS Exclusive-Nor Gate

- 8-transistor implementation

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>TG₁</th>
<th>TG₂</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>nonconducting</td>
<td>conducting</td>
<td>B (1)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>nonconducting</td>
<td>conducting</td>
<td>B (0)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>conducting</td>
<td>nonconducting</td>
<td>B (0)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>conducting</td>
<td>nonconducting</td>
<td>B (1)</td>
</tr>
</tbody>
</table>

- Better, 6-transistor implementation is possible!

4:1 Multiplexer

- 4:1 mux chooses one of 4 inputs using two selects
  - Two levels of 2:1 muxes
  - Or four tristates
D Latch

- When CLK = 1, latch is *transparent*
  - D flows through to Q like a buffer
- When CLK = 0, the latch is *opaque*
  - Q holds its old value independent of D
- a.k.a. *transparent latch* or *level-sensitive latch*

Memory Elements: Latches and Flip-Flops

- Difference between a latch and a flip-flop?

- How to design negative level-sensitive D-latch?

*Recirculating latch*
D Latch Design

- Multiplexer chooses D or old Q

Memory Elements: Latches and Flip-Flops

Clock = 1
- Q changes faster than Q

Clock = 0
- Q changes slower than Q
D Latch Operation

- When CLK rises, D is copied to Q
- At all other times, Q holds its value
- a.k.a. positive edge-triggered flip-flop, master-slave flip-flop

D Flip-flop

- When CLK rises, D is copied to Q
- At all other times, Q holds its value
- a.k.a. positive edge-triggered flip-flop, master-slave flip-flop
**D Flip-flop Design**

- Built from master and slave D latches

![D Flip-flop Diagram]

**Flip-Flop Design**

- Positive edge-triggered D flip-flop
- How do we obtain the \( \overline{Q} \) output?

![Flip-Flop Diagram]
D Flip-flop Operation

Race Condition

- Back-to-back flops can malfunction from clock skew
  - Second flip-flop fires late
  - Sees first flip-flop change and captures its result
  - Called hold-time failure or race condition
Nonoverlapping Clocks

- Nonoverlapping clocks can prevent races
  - As long as nonoverlap exceeds clock skew
- We will use them in this class for safe design
  - Industry manages skew more carefully instead

Design Representation Levels

- Design domains
  - Behavioral
  - Structural
  - Physical
- Gajski and Kuhn’s Y-chart (layered like an onion)
- Hardware description languages commonly used at behavioral level, e.g. VHDL, Verilog
- Example: Consider the carry function \( c_o = ab + bc + c_ia \)
Verilog Example (Behavioral)

Boolean equation form:

```verilog
module carry (co, a, b, ci);
output co;
input a, b, ci;
assign
co = (a & b) | (a & ci) | (b & ci);
end module
```

Boolean truth table form:

```verilog
primitive carry (co, a, b, ci);
output co;
input a, b, ci;
table
// a   b   c   co
1   1  ?   1;
1   ?  1   1;
?   1  1   1;
0   0  ?   0;
0   ?  0   0;
?   0  0   0;
end table
end module
```

Timing information:

```verilog
module carry (co, a, b, ci);
output co;
input a, b, ci;
Wire #10 co = (a & b) | (a & ci) | (b & ci);
end module
```

c_o changes 10 time units after a, b, or c changes

Verilog Example (Structural)

Structural representation of 4-bit adder (top-down):

```verilog
module add4 (s, c4, ci, a, b);
output [3:0] s;
output [3:0] c4;
input [3:0] a, b;
input ci;
wire [2:0] c_o;
add a0 (c_o[0], s[0], a[0], b[0], ci);
add a1 (c_o[1], ... , b[1], c_o[0]);
add a2 (c_o[2], ... , c_o[1]);
add a3 (c4, s[3], a[3], b[3], c_o[2]);
end module
```

Technology-independent

```verilog
module add (co, s, a, b, ci);
output s, co;
input a, b, ci;
sum s1 (s, a, b, ci);
carry c1 (co, a, b, ci);
end module
```

```verilog
module carry (co, a, b, ci);
output co;
input a, b, ci;
wire x, y, z;
and g1 (y, z, b);
and g2 (z, b, ci);
and g3 (z, a, ci);
or g4 (co, x, y, z);
end module
```

Technology-independent