Lecture 3

Switch Logic and Gates

C.K. Ken Yang
University of California at Los Angeles
yang@ee.ucla.edu

Overview

Reading
 – W&H: 1.4

Introduction
Transistors can be modeled to be switched resistors and capacitors. Two flavors of transistors are needed in order to pass both 0 and 1 signals. This lecture addresses how we can combine these switches to build logic networks. We start with switching networks. Although they suffice for logic, they permit signals to be passed in both directions. CMOS logic gates are introduced to build an arbitrary boolean function. Gates also decouple the inputs from the outputs such that the output is only a function of the inputs. Practically, the kinds of CMOS gates that you can construct are limited by the need for stacks of series transistors, and their effect on gate performance.
Switch Networks

• The function of a switch network is true when the two terminals of the network are connected together. Since for parallel switches the terminals are connected if either switch is conducting, the function is OR. For series switches the network is conducting only if both switches are conducting, hence an AND.
  – If A+B, Y=X

General Switch Networks

• More complex connections are possible
  – Many possible implementation of a single function.
• Composition rules are simple. Use a recursive definition:
  – Parallel combination of switch networks yields an OR of the component switch networks' functions
  – Series combination of switch networks yields on AND of the component switch networks' functions.
Switch Logic

Using switch-networks we can build up a simple kind of logic. The basic idea is to use switches (controlled by switching inputs) to route one of several inputs to the output. There are two rules you must follow for switch logic to work:

- The primary output must always be connected to one of the inputs
  - (the OR of all the switch-networks to output must be 1)
- Two (or more) inputs must not be connected together
  - (the AND of any two of the switch-networks to output must be 0)
  - (unless they are both constants and have the same value)

- For now we will assume that both true and complement values of the inputs are available. A little later we will talk about how to make inverters to generate the complements.

Multiplexer

- A very useful switch network in an input multiplexer. It simply selects one of the inputs to the output. This structure can be used to easily map any logical function into switch logic -- all that needs to be done is present the right constant vector to the inputs of the multiplexer.

Notice that the switch networks are exclusive of each other (AND is 0), and that the OR of all the terms is 1.

The layout shown is NOT a good way to build this function. It uses diffusion wires, which we will see later is not a good choice.
**Muxes**

- For some functions you can do better than just using constants and a multiplexor. You can implement an XOR gate in only two switches (if you assume that both the inputs and their complements are available).
- Inputs are not just connected to the switching input (gate), but also to the conducting inputs (source/drain).

![Mux Diagram](image)

---

**Parity Function**

- A more complex switch logic function:
  - \( A \ XOR \ B \ XOR \ C \ XOR \ D \ ...

- Try to minimize the work you need to do, so try for an iterating structure

![Parity Diagram](image)

- Parity of \((..., A_n, A_{n+1}) = Parity(..., A_n) \ XOR \ A_{n+1}\)
- This looks promising but the XOR switch logic needs both \(In\) and \(In_b\)
- We need to build both \(XOR\) and \(XNOR\)
Parity

Each stage looks like

Parity of Three Inputs

For more inputs, this structure can be further cascaded.
- Elegant solution but not practical as is for high performance.
Tally Function

- Even more complex function can be implemented in switch logic. The function counts the number of ones in the input word:
- \( Z_n \) is 1 if there are \( N \) 1’s in the input word
- For an \( n \)-bit number there are \( N+1 \) outputs
  - \( Z_0, Z_1, \ldots, Z_n \)
  - One-hot outputs

Example:

<table>
<thead>
<tr>
<th>Input</th>
<th>( Z_0 )</th>
<th>( Z_1 )</th>
<th>( Z_2 )</th>
<th>( Z_3 )</th>
<th>( Z_4 )</th>
<th>( Z_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>00001</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Tally Function Implementation

- The easiest way to solve this is to solve iteratively (like parity):
  - \( T_n = f(T_{n-1}, \text{Input}_n) \)
- Here each stage is a little different (since it must produce a different number of outputs)
  - How to build a stage?
    - If bit is one, increment count by shifting \( Z_n \) by 1
    \[ Z_n \rightarrow Z_n + 1 \]
    - If bit is zero \( Z_n \) remain the same
    \[ Z_n \rightarrow Z_n \]
  - So, one implementation is just a shifter!
Tally Function Switch Implementation

- Shown below is the tally function for one bit. It has two outputs, \( Z_0 \) and \( Z_1 \).
- When the data is 0, the diagonal transistors are off, and the horizontal path (complementary switches) are on.
  - Output \( Z_1 \) is set to 0
  - Output \( Z_0 \) is set to 1
- When the data is 1, the diagonal transistors are on, and the horizontal switches are off
  - Output \( Z_1 \) is set to 1
  - Output \( Z_0 \) is set to 0
- Note: Each output is always driven by one and only one value. (Switch logic rule)

Two Input Tally

- Simple generalization of the one input case.
- Built by cascading 1 input tally functions.
- Size of circuit is \( O(n^2) \) where \( n \) is the number of data inputs.
- Simple cell (two transistors) can be replicated to build larger circuits

Switches are set for 1, 1, so the diagonal path is connected (\( Z_2 = 1 \))
Two Input Tally: Example of a Route Path

- Circuit works by routing inputs either across or up

![Diagram showing two input tally]

Two Basic Abstractions

To help us visualize complex logic, two useful abstractions are used:

- Digital Abstraction
  - Signals are either 1 or 0 represented by voltages
  - As long as signals are above a $V_{\text{threshold\_HIGH}}$, it is a 1, and below a $V_{\text{threshold\_LOW}}$, it is a 0.

- Gate Abstraction
  - Logical functions can be encapsulated and isolated as Gates.
    - Output is a logical function of N-inputs.
  - Good gate characteristics:
    - Regenerates to Digital Values (even if inputs are degenerated)
    - Unidirectional (outputs do not effect inputs.)
Switches Using Only One Transistor Flavor

- If you connect this degraded output to the gate of another nMOS switch, you would get an output that is degraded by 2 $V_T$. This may be too low to detect as a high output.
  - We need to provide signals that satisfy the digital abstraction of 1’s and 0’s.

- Passing a logic 0 is much easier, since then the transistor is always on ($V_{GS} = V_{DD}$). nMOS devices don’t degrade low levels.
- The same condition occurs for pMOS passing a logic 0 (see last lecture.)

Transmission Gates

- By using both nMOS and pMOS neither output is degraded
- But you need the true and complement of the control signal
- Using transmission gates, you don’t degrade logic levels
  - Simpler design issues, since there is only full swing signal
  - Full signals can propagate in both directions (no in or out)
- For most CMOS designs you are not allowed to have degraded levels even though circuits may be faster.
  - Robustness versus Area/Power/Performance tradeoff
Transmission Gate Model

- Modeled with 2 resistors in parallel to represent the two transistors.
  - Both are ON simultaneously.
  - \( R_P \) and \( R_N \) depends on passing a HIGH or LOW value.

\[
\begin{align*}
  R_P &= R_{\text{pull up PMOS}} \\
  R_N &= R_{\text{pull up NMOS}} \\
  R_P &= R_{\text{pull down PMOS}} \quad \text{(higher R)} \\
  R_N &= R_{\text{pull down NMOS}} \quad \text{(lower R)}
\end{align*}
\]

CMOS Switch Networks

- In general one needs to use full CMOS transmission gates
  - Two control lines per switch
  - No degraded levels

Examples:

- A AND B
- A OR B

If the switch network only connects to a constant (Vdd or Gnd) then you don’t need both transistors. Connections to \( V_{\text{dd}} \) only need pMOS, and connections to \( V_{\text{ss}} \) only need nMOS.

Courtesy of MAH

M216A Lecture 3
Switch Logic

- Example: 2-1 Mux:

CMOS switch logic need a large number of control wires
- Each control is needed in true and complement form
- For 2-1 Mux this works out well, but for a 3-1 mux, this means 6 ctrls
  - SelA, SelB, SelC and their complements

Inverters

- Switch logic has several limitations. An important one is that it can’t invert signals! To do useful stuff we clearly need (at least) inverters.
- To build an inverter with switch networks, you want to connect the output to $V_{\text{GND}}$ when the input is high (a switch) and connect the output to $V_{\text{DD}}$ when the input is low (another switch).
- These switches DO NOT need to be full T-gates!
  - Pull up only needs pMOS (passes 1 well), input LOW to pass HIGH (perfect!)
  - Pull down only needs nMOS (passes 0 well)
Transmission Gate Logic Networks

Advantages
- Small and dense.
- Less capacitance.

Disadvantages
- Signal is not regenerated.
  - Transmission Gates are resistors, a weakened input signal results in a weakened output.
  - Noise (such as switching noise) can degrade the signal.
- Signal propagates in both directions.
  - Noise at the output can effect the input.
  - Outputs can drive to many inputs… so robustness is an issue.
- Signal propagates through lots of R’s (potentially large RC delay)

Static CMOS Gates

- Transmission gates allow inputs to connect to both the conducting input of a switch (S/D) and the switching input (Gate)
  - Causes the problems.
- If conducting inputs only connect to 1’s and 0’s, then many of the problems go away.
  - Degraded inputs can still turn on a transmission gate (just not as low of a resistance)
  - Noise on the output only affects the supply rails (V_{DD} and V_{GND}) and not the inputs.
    - Unidirectional.
    - If transmission gates are only connected to one supply, then the gate can be simplified.
NOR Gate Example

<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>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

NOR Output

Intuitive NOR Gate

- This is intuitively correct.
- \(F(\text{NOR}) = (A+B)'\)
  - Output is low when either \(A\) or \(B\) is high.
    - NMOS’s form an OR network to \(V_{\text{GND}}\).
  - Output is high when \(A\) and \(B\) are both low
    - PMOS’s form an AND network to \(V_{\text{DD}}\).
NAND Gate

- **NAND**
  - Output is low when A **and** B are both high
  - Output is high when either A **or** B is low

![NAND Gate Diagram]

CMOS Buffer

- Can build any gate using switches, even a buffer
  - But turns out to be an inverter of the inverted input
  - All CMOS gates invert (an inverting logic family”)

![CMOS Buffer Diagram]

For a non-inverting buffer, we need to use 2 inverters.
Building General Static CMOS Gates

- This type of gate is known as static CMOS logic gates.
- This type of LOGIC FAMILY can implement any function.
  - Each gate is inverting
  - Use inverters to invert outputs or inputs.
- Need to build 2 networks: a pull-up and a pull-down network.
  - Pull-up with PMOS, and pull-down with NMOS
  - The two networks are duals
    - series pull-down -> parallel pull-up
    - parallel pull-down-> series pull-up
- Easier to build N tree first, because easy to forget P tree inverts inputs.

CMOS Gates

- To build a logic gate \( g(x_1, \ldots, x_n) = f'(x_1, \ldots, x_n) \), need to build two switch networks:
  - Pull-up network
    - Connects the output to \( V_{DD} \) when \( f \) is FALSE
    - \( \beta \) pMOS only, since only passes 1
  - Pull-down network
    - Connects the output to \( V_{GND} \) when \( f \) is TRUE
    - \( \alpha \) nMOS only, since only passes 0
- Pull-down
  - \( \alpha(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \)
- Pull-up
  - \( \beta(x_1', \ldots, x_n') = f'(x_1, \ldots, x_n) \) (since pMOS invert inputs)

Notice that the constraints on the two switch networks is just what we talked about for switch logic. The output must be driven \( (\alpha + \beta) = 1 \), and there can’t be conflicts \( (\alpha \beta) = 0 \).
Duality

The pull-up network and pull-down network are duals of each other.

- Dual of a function:
  - Exchange ANDs and ORs

- Example Duals
  - $A \cdot B ; A + B$
  - $(A + B) \cdot C ; (A \cdot B) + C$

For switch networks

- AND = series switches
- OR = parallel switches

- So
- Parallel pull-down, serial pull-up and vice versa

Why?

De Morgan’s Law

- Remember DeMorgan’s Law?
- $(a + b)' = a' \cdot b'$
- $(a \cdot b)' = a' + b'$

- More generally the complement of a function can be obtained by replacing each variable / element with its complement, and exchanging the AND and OR operations

- One of the most useful rules in boolean algebra
- Can apply to arbitrarily complex expressions.
- If element is not a single variable, then apply recursively to the expressions:
  - $((A+B) \cdot C)' = (A + B)' + C' = (A' \cdot B') + C'$
  - $((A \cdot B) + (C \cdot D))' = (A \cdot B)' \cdot (C \cdot D)' = (A' + B') \cdot (C' + D')$
**Simple Example**

- Function: And-Or-Invert (AOI): \( g = (AB+C)' \)
- Pull-down implement: \( f_N = AB+C \)
- Pull-up implement DUAL of \( f_N \)
  - \( f_P = (A'+B')C' \) - inverted inputs
  - The network that you build is \((A+B)C\)

**More Examples**

- 3 input function, \( g = a(b+c) + bc \)
  - For pull-down, \( f = (a'+b'c')(b'+c') \)
  - For pull-up, \( f' = a''(b''+c'')+b''c'' \)

- 5 input function, \( g = ab + (cde)' \)
  - For pull-down, \( f = (a'+b')(cde) \)
  - For pull-up, \( f' = a''b'' + (c' + d' + e') \)

- Note that the depth of the stacking is approximately the number of inputs.
Many Implementation for the Same Function

- Building CMOS gates is a logic minimization problem.
  - Reducing redundancies
  - Example: \( f_b = abc + ab + acd + bd + bcd \)
    - Equiv \( f_b = ab + acd + bd \)
    - Equiv \( f_b = b(a+d) + acd = a(b + cd) + bd \)
- Same expression can be implemented in several ways
  - Ordering of the AND.

- Is one better?
  - Generally better to reduce redundancies (each variable instance is a device).
  - Some inputs will be faster than others.
    - Ones closer to the output (see why later)

Mixing Static CMOS with Transmission Gate Logic

Transmission Gate Switch Logic can be made to satisfy Gate Abstraction

- Use static CMOS gates outputs to drive a transmission gate switch network (conducting inputs).
- Output of network drives a static CMOS gate.
- Example: a 2:1 Mux
Why Static CMOS Gates?

- Inputs of gates are capacitive.
- If one gate always drive another gate
  - Outputs are full digital values
    - Driven by a low resistance to a supply rail.
  - No static power dissipation.
    - The design needs to obey switch network rule
    - Output is always and only connected to ONE conducting input.
    - Dynamic energy is needed to charge or discharge the capacitance.
- Logical functionality is guaranteed by the network.
  - No dependence on size of device etc.
- Good for Digital and Gate Abstractions.

Complex Static CMOS Gates

- In theory can build any arbitrary logic function in a single gate
  - Take the complement of the function
  - Build a switch network out of nMOS devices and connect between $V_{\text{GND}}$ and Out
  - Build the dual switch network out of pMOS devices and connect between $V_{\text{DD}}$ and Out
- In practice the number of practical gate types is limited
  - Want a finite number of gate types (need to design/test/layout them)
  - One complex gate can be SLOWER than several smaller gates connected together to implement the same function.
- To understand why we need to understand delay of a gate.
- But first, let’s look at how to layout a gate and the other device: the wire.