Gates and Logic

Hakim Weatherspoon
CS 3410, Spring 2012
Computer Science
Cornell University

See: P&H Appendix C.2 and C.3 (Also, see C.0 and C.1)
A switch

- Acts as a *conductor* or *insulator*
- Can be used to build amazing things...
Goals for today

To understand how to program,
we will build a processor (i.e. a logic circuit)

Logic circuits
• Use P- and N-transistors to implement NAND or NOR gates
• Use NAND or NOR gates to implement the logic circuits
• Build efficient logic circuits
Better Switch

- One current controls another (larger) current

- Static Power:
  - Keeps consuming power when in the ON state

- Dynamic Power:
  - Jump in power consumption when switching
Elements

Boron

Silicon

Phosphorus
Phosphorus Doping
N-Type: Silicon + Phosphorus
Boron Doping
P-Type: Silicon + Boron
Semiconductors

p-type (Si+Boron) has mobile holes:
- low voltage (depleted) → insulator
- high voltage (mobile holes) → conductor

n-type (Si+Phosphorus) has mobile electrons:
- low voltage (mobile electrons) → conductor
- high voltage (depleted) → insulator
Bipolar Junction

P-Type

low v → insulator
high v → conductor

N-Type

low v → conductor
high v → insulator
Reverse Bias

P-Type

low $v \rightarrow$ insulator
high $v \rightarrow$ conductor

N-Type

low $v \rightarrow$ conductor
high $v \rightarrow$ insulator
Forward Bias

P-Type

N-Type

low v → insulator
high v → conductor

low v → conductor
high v → insulator
Diodes

PN Junction “Diode”

Conventions:
$\text{vdd} = \text{vcc} = +1.2\,\text{v} = +5\,\text{v} = \text{hi}$
$\text{vss} = \text{vee} = 0\,\text{v} = \text{gnd}$
PNP Junction

\[ \text{p-type} \quad \text{n-type} \quad \text{p-type} \]
Bipolar Junction Transistors

- Solid-state switch: The most amazing invention of the 1900s
  Emitter = “input”, Base = “switch”, Collector = “output”

PNP Transistor

```
C  p  n  p
B   E=vdd
```

NPN Transistor

```
vss=E  n  p  n
B     C
```

PNP Transistor

```
E=E  Vdd
B   C
```

NPN Transistor

```
C  E
B   VSS
```
Field Effect Transistors

**P-type FET**

- Connect Source to Drain when Gate = lo
- Drain must be vdd, or connected to source of another P-type transistor

**N-type FET**

- Connect Source to Drain when Gate = hi
- Source must be vss, or connected to drain of another N-type transistor
Multiple Transistors

Gate delay
- transistor switching time
- voltage, propagation, fanout, temperature, ...

CMOS design
(complementary-symmetry metal–oxide–semiconductor)
- Power consumption = dynamic + leakage

Power consumption =

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vdd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vss</td>
</tr>
</tbody>
</table>
### Digital Logic

![Logic Diagram]

**Vdd**

**in**

**out**

**Vss**

#### Voltage

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>+5v</td>
<td>0v</td>
</tr>
<tr>
<td>0v</td>
<td>+5v</td>
</tr>
</tbody>
</table>

#### Truth Table

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
NOT Gate (Inverter)

Function: NOT
- Symbol:

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Truth table
NAND Gate

Function: NAND
- Symbol:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>out</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>
NOR Gate

Function: NOR
- Symbol:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>out</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>
Building Functions

- AND:

- OR:

- NOT:
Universal Gates

NAND is universal (so is NOR)

- Can implement any function with just NAND gates
  - De Morgan’s laws are helpful (pushing bubbles)
- Useful for manufacturing

E.g.: XOR (A, B) = A or B but not both (“exclusive or”)

Proof: ?
Administrivia

Make sure you have access to CMS and Piazza.com

Lab Sections started **this** week
  • Lab0 turned in during Section
  • Bring laptop to section, if possible (not required)
  • Lab1 available Monday next week (due following Monday)
  • Group projects start in week 4 (partner in same section)

Homework1 available Monday
  • Due following Monday

Office hours start next week
  • More information available on website by this weekend

Clickers not required, bring to every lecture
  • Participation, not attendance
Logic Equations

Some notation:
• constants: true = 1, false = 0
• variables: a, b, out, ...
• operators:
  • AND(a, b) = a b = a & b = a ∧ b
  • OR(a, b) = a + b = a | b = a ∨ b
  • NOT(a) = ā = !a = ¬a
Identities

Identities useful for manipulating logic equations

– For optimization & ease of implementation

\begin{align*}
  a + 0 &= a \\
  a + 1 &= 1 \\
  a + \overline{a} &= 1 \\
  a \overline{0} &= 0 \\
  a \overline{1} &= a \\
  a \overline{\overline{a}} &= 0 \\
  (a + b) &= \overline{\overline{a} \overline{b}} \\
  (a \overline{b}) &= \overline{\overline{a} + b} \\
  a + a \overline{b} &= a \\
  a(b+c) &= a \overline{b} + ac \\
  a(b+c) &= \overline{a} + \overline{b}c
\end{align*}
Logic Manipulation

- functions: gates ↔ truth tables ↔ equations
- Example:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Logic Minimization

• A common problem is how to implement a desired function most efficiently
• One can derive the equation from the truth table

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>minterm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>abc</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>abc</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>abc</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>abc</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>abc</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>abc</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>abc</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>abc</td>
</tr>
</tbody>
</table>

for all outputs that are 1, take the corresponding minterm
Obtain the result in “sum of products” form

• How does one find the most efficient equation?
  – Manipulate algebraically until satisfied
  – Use Karnaugh maps (or K maps)
Karnaugh maps

- Encoding of the truth table where adjacent cells differ in only one bit.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</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>1</td>
</tr>
</tbody>
</table>

Truth table for AND

00 01 11 10
0 0 1 0

Corresponding Karnaugh map
Bigger Karnaugh Maps

3-input function

4-input function

00 01 11 10
0 1
00 01 11 10
00 01 11 10
00 01 11 10
Minimization with Karnaugh maps (1)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</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>0</td>
</tr>
</tbody>
</table>

Sum of minterms yields

- $\overline{abc} + \overline{abc} + \overline{abc} + \overline{abc}$
Minimization with Karnaugh maps (2)

Sum of minterms yields

\[ \text{abc} + \text{abc} + \text{abc} + \text{abc} \]

Karnaugh maps identify which inputs are (ir)relevant to the output
Minimization with Karnaugh maps (2)

Sum of minterms yields
- \( \overline{abc} + \overline{abc} + \overline{abc} + \overline{abc} \)

Karnaugh map minimization
- Cover all 1’s
- Group adjacent blocks of \( 2^n \) 1’s that yield a rectangular shape
- Encode the common features of the rectangle
  - \( \text{out} = \overline{ab} + \overline{ac} \)
Karnaugh Minimization Tricks (1)

- Minterms can overlap
  - \( \text{out} = b\bar{c} + a\bar{c} + ab \)

- Minterms can span 2, 4, 8 or more cells
  - \( \text{out} = \bar{c} + ab \)
• The map wraps around
  – out = \overline{bd}

  – out = \overline{bd}
Karnaugh Minimization Tricks (3)

- "Don’t care" values can be interpreted individually in whatever way is convenient:
  - assume all x’s = 1
  - out = d
  - assume middle x’s = 0
  - assume 4th column x = 1
  - out = \text{bd}
Multiplexer

• A multiplexer selects between multiple inputs
  – out = a, if d = 0
  – out = b, if d = 1

• Build truth table
• Minimize diagram
• Derive logic diagram
Multiplexer Implementation

- Build a truth table
  \[ = \text{abd} + \text{abd} + \text{a bd} + \text{a b d} \]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>d</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Multiplexer Implementation

- Build the Karnaugh map

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>d</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Multiplexer Implementation**

- Derive Minimal Logic Equation

\[ \text{out} = ad + bd \]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>d</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Multiplexer Implementation

- Derive Minimal Logic Equation

\[
\text{out} = \overline{a}d + bd
\]
Summary

• We can now implement any logic circuit
  – Can do it efficiently, using Karnaugh maps to find the minimal terms required
  – Can use either NAND or NOR gates to implement the logic circuit
  – Can use P- and N-transistors to implement NAND or NOR gates