CSCI-GA.1144-001
PAC II

Lecture 2: Digital Logic

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
It all starts with TRANSISTORS
Transistor: Building Block of Computers

- Microprocessors contain millions of transistors
- Logically, each transistor acts as a switch
- Combined to implement logic functions
  - AND, OR, NOT
- Combined to build higher-level structures
  - Adder, multiplexer, decoder, register, ...
- Combined to build processor
**Simple Switch Circuit**

- **Switch open**:  
  - No current through circuit  
  - Light is off

- **Switch closed**:  
  - Short circuit across switch  
  - Current flows  
  - Light is on

*Switch-based circuits* can easily represent two states: on/off, open/closed, voltage/no voltage.
Switch-based circuits can easily represent two states: on/off, open/closed, voltage/no voltage.

Simple Switch Circuit

MOS Transistors:
- n-type
- p-type

This is exactly what a transistor does!
n-type MOS Transistor

- **n-type**
  - when Gate has **positive** voltage, short circuit between #1 and #2 (switch **closed**)
  - when Gate has **zero** voltage, open circuit between #1 and #2 (switch **open**)

![Diagram of n-type MOS Transistor]
p-type MOS Transistor

- **p-type** is complementary to n-type
  - when Gate has **positive** voltage, open circuit between #1 and #2 (switch **open**)
  - when Gate has **zero** voltage, short circuit between #1 and #2 (switch **closed**)

![Diagram of p-type MOS Transistor]

- Gate = 1
- Gate = 0
Logic Gates

- Use switch behavior of MOS transistors to implement logical functions: AND, OR, NOT.

Transistors  Building Blocks  Logic Gates
Inverter (NOT Gate)

Truth table

<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>

$P$-type

$N$-type
NOR Gate

A
B
C

A = 0
B = 1
C = 0

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</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>
Add inverter to NOR.
NAND Gate (AND-NOT)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</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>
AND Gate

Add inverter to NAND.

<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>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>
Basic Logic Gates

**NOT**

**OR**

**NOR**

**AND**

**NAND**
DeMorgan's Law

- Converting AND to OR (with some help from NOT)
- Consider the following gate:

\[ \overline{A \cdot B} \]

To convert AND to OR (or vice versa), invert inputs and output.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>( A' )</th>
<th>( B' )</th>
<th>( A \cdot B' )</th>
<th>( A' \cdot B' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</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>
<td>1</td>
</tr>
</tbody>
</table>
DeMorgan's Law

- not(A and B) = (not A) or (not B)

\[ A \text{ and } B = \overline{A} \text{ or } \overline{B} \]

- not(A or B) = (not A) and (not B)

\[ A \text{ or } B = \overline{A} \text{ and } \overline{B} \]
More than 2 Inputs?

- AND/OR can take any number of inputs.
  - AND = 1 if all inputs are 1.
  - OR = 1 if any input is 1.
  - Similar for NAND/NOR.

- Can implement with multiple two-input gates, or with single CMOS circuit.
A logic function can be represented as
- a truth table
- a logic expression
- a logic circuit

Example

\[ f = a.(b.c + d) + \overline{a}.c = a.b.c + a.d + \overline{a}.c \]
Building Functions from Logic Gates

• **Combinational Logic Circuit**
  – output depends only on the current inputs
  – stateless

• **Sequential Logic Circuit**
  – output depends on the sequence of inputs (past and present)
  – stores information (state) from past inputs

Transistors ————————> Logic gates ————————> Functions
Combinational Circuits
Decoder

• \( n \) inputs, \( 2^n \) outputs
  – exactly one output is 1 for each possible input pattern

2-bit decoder

2x4 Decoder
Decoder

- \( n \) inputs, \( 2^n \) outputs
  - exactly one output is 1 for each possible input pattern

2-bit decoder
When do we use decoders?
Multiplexer (MUX)

- $n$-bit selector and $2^n$ inputs, one output
  - output equals one of the inputs, depending on selector

4-to-1 MUX

A, if $S=00$
B, if $S=01$
C, if $S=10$
D, if $S=11$
Full Adder

- Add two bits and carry-in, produce one-bit sum and carry-out.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>A</th>
<th>B</th>
<th>C_in</th>
<th>S</th>
<th>C_out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<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>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Four-bit Adder

Called Ripple-carry adder.
Logical Completeness

• We can implement any logic function with \{AND, OR, NOT\} gates.

→ Set of \{AND, OR, NOT\} is logically complete

• Which other gates are logical complete?
Combinational vs. Sequential

• **Combinational Circuit**
  – always gives the same output for a given set of inputs
    • ex: adder always generates sum and carry, regardless of previous inputs

• **Sequential Circuit**
  – stores information
  – output depends on stored information (state) plus input
    • so a given input might produce different outputs, depending on the stored information
  – *example*: ticket counter
    • advances when you push the button
    • output depends on previous state
  – useful for building “memory” elements and “state machines”
Sequential Circuits
R-S Latch: Simple Storage Element

- R is used to “reset” or “clear” the element - set it to zero.
- S is used to “set” the element - set it to one.

If both R and S are one, out could be either zero or one.
  - “quiescent” state -- holds its previous value
  - note: if a is 1, b is 0, and vice versa
R-S Latch Summary

- **R = S = 1**
  - hold current value in latch
- **S = 0, R=1**
  - set value to 1
- **R = 0, S = 1**
  - set value to 0

- **R = S = 0**
  - both outputs equal one
  - final state determined by electrical properties of gates
  - *Don’t do it!*
Gated D-Latch

• Two inputs: D (data) and WE (write enable)
  – when $WE = 1$, latch is set to value of D
    • $S = \text{NOT}(D), R = D$
  – when $WE = 0$, latch holds previous value
    • $S = R = 1$
Register

• A register stores a multi-bit value.
  – We use a collection of D-latches, all controlled by a common WE.
Representing Multi-bit Values

- Number bits from right (0) to left (n-1)
- Use brackets to denote range: \( D[l:r] \) denotes bit \( l \) to bit \( r \), from left to right

\[
A = \overbrace{0101001101010101}^{15} \overbrace{0101010101010101}^{0}
\]

\[
A[14:9] = \underbrace{101001}_{15} \quad A[2:0] = \underbrace{101}_{0}
\]
Memory

• Now that we know how to store bits, we can build a memory - a logical $k \times m$ array of stored bits.

**Address Space:**
number of locations (usually a power of 2)

$k = 2^n$
locations

**Addressability:**
number of bits per location (e.g., byte-addressable)

$m$ bits
$2^2 \times 3$ Memory

One gated D-latch
Reading A Location in Memory

A = 11
WE = 0
More Memory Details

• This is not the way actual memory is implemented.
  – fewer transistors, much more dense,
    relies on electrical properties

• But the logical structure is very similar.
  – address decoder
  – word select line
  – word write enable

• Two basic kinds of **RAM** (Random Access Memory)

• **Static RAM** (SRAM)
  – fast, maintains data as long as power applied

• **Dynamic RAM** (DRAM)
  – slower but denser, bit storage decays - must be periodically refreshed

Also, non-volatile memories: ROM, PROM, flash, ...
State Machine

- Another type of sequential circuit
  - Combines combinational logic with storage
  - "Remembers" state, and changes output (and state) based on inputs and current state
Example: Combinational vs. Sequential Lock

• Two types of “combination” locks

**Combinational**
Success depends only on the **values**, not the order in which they are set.

**Sequential**
Success depends on the **sequence** of values (e.g., R-13, L-22, R-3).
The state of a system is a snapshot of all the relevant elements of the system at the moment the snapshot is taken.

Examples:

- The state of a basketball game can be represented by the scoreboard.
  - Number of points, time remaining, possession, etc.

- The state of a tic-tac-toe game can be represented by the placement of X’s and O’s on the board.
State of Sequential Lock

• Our lock example has four different states, labelled A-D:

  A: The lock is not open, and no relevant operations have been performed.
  
  B: The lock is not open, and the user has completed the R-13 operation.
  
  C: The lock is not open, and the user has completed R-13, followed by L-22.
  
  D: The lock is open.
State Diagram

- Shows **states** and **actions** that cause a **transition** between states.
Finite State Machine

- A description of a system with the following components:

1. A finite number of states
2. A finite number of external inputs
3. A finite number of external outputs
4. An explicit specification of all state transitions
5. An explicit specification of what determines each external output value

- Often described by a state diagram.
  - Inputs trigger state transitions.
  - Outputs are associated with each state (or with each transition).
Implementing a Finite State Machine

- **Combinational logic**
  - Determine outputs and next state.
- **Storage elements**
  - Maintain state representation.
Storage: Master-Slave Flipflop

- A pair of gated D-latches, to isolate next state from current state.

During 1st phase (clock=1), previously-computed state becomes current state and is sent to the logic circuit.

During 2nd phase (clock=0), next state, computed by logic circuit, is stored in Latch A.
Storage

• Each master-slave flipflop stores one state bit.

• The number of storage elements (flipflops) needed is determined by the number of states (and the representation of each state).

• Example:
  – Sequential lock
    • Four states - two bits
Complete Example

- A blinking traffic sign
  - No lights on
  - 1 & 2 on
  - 1, 2, 3, & 4 on
  - 1, 2, 3, 4, & 5 on
  - (repeat as long as switch is turned on)
Traffic Sign State Diagram

Transition on each clock cycle.
Traffic Sign Truth Tables

Outputs
(depend only on state: $S_1S_0$)

<table>
<thead>
<tr>
<th>$S_1$</th>
<th>$S_0$</th>
<th>Z</th>
<th>Y</th>
<th>X</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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Next State: $S_1'S_0'$
(depend on state and input)

<table>
<thead>
<tr>
<th>$In$</th>
<th>$S_1$</th>
<th>$S_0$</th>
<th>$S_1'$</th>
<th>$S_0'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Switch

Whenever $In=0$, next state is 00.
Traffic Sign Logic

In

Storage Element 0

Storage Element 1

Clock

Z
Y
X

S₀'
S₁'

Master-slave flipflop
From Logic to Data Path

- The data path of a computer is all the logic used to process information.

- **Combinational Logic**
  - Decoders -- convert instructions into control signals
  - Multiplexers -- select inputs and outputs
  - ALU (Arithmetic and Logic Unit) -- operations on data

- **Sequential Logic**
  - State machine -- coordinate control signals and data movement
  - Registers and latches -- storage elements
Conclusions

• MOS transistors are used as switches to implement logic gates.

• Logic gates $\rightarrow$ digital structures

• Digital structures:
  – Combinational circuit (e.g. MUXes, decoders, ...)
  – Sequential circuits (flip-flops, registers, memory elements, ...)

• Next lecture we will see how to use digital structures to build a processor.