Logic Design I

CSci 2021: Machine Architecture and Organization
Lecture #36, April 24th, 2015

Your instructor: Stephen McCamant

1

Brief History of Computing Machines

- **1800s: purely mechanical**
  - General-purpose computer designed, but not fully built
- **1940s: first general-purpose computers**
  - Electromechanical relays
  - Vacuum tubes
  - Idea: electrically-controlled electric switch
- **1947: transistor**
  - "Solid state": no moving parts or gases
  - Based on semiconducting materials like silicon
  - Can be used as a switch or an amplifier
  - Takes a lot to make a computer…

2

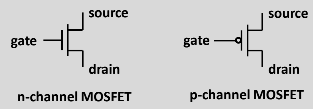
Integrated Circuits

- **Key technology for inexpensive computing**
  - Printing transistors (and other devices) on a silicon wafer
- **Low incremental production cost**
  - But the design and the factory are expensive
- **Long history of increasing density**
  - First ICs had <100 devices per chip
  - Moore's law: exponential increase in # of transistors per device
    - Doubling every 12-24 months
  - Modern CPU: tens of billions of transistors

3

MOSFETs

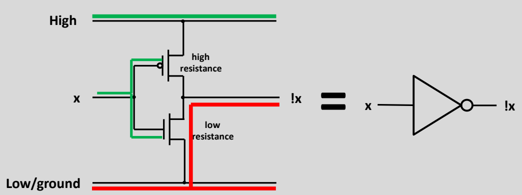
- **Modern kind of transistor used in ICs**
  - Metal-oxide-semiconductor field-effect transistor



- **Voltage at the gate determines whether current can flow between source and drain**
  - n-channel type: high voltage allows current to flow
  - p-channel type: low voltage allows current to flow

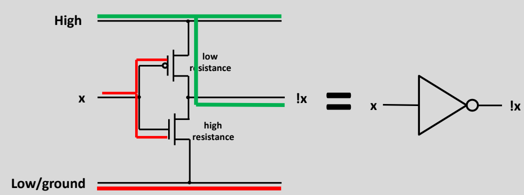
4

Transistors To Gates: CMOS Inverter



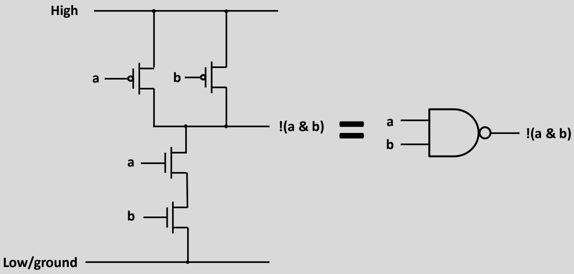
5

Transistors To Gates: CMOS Inverter



6

CMOS NAND Gate

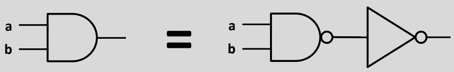


Logistics Note: (No) Readings

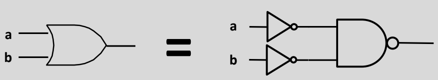
- **Most of this material is not in the textbook**
- **We've posted links to free online resources**
  - On the "Useful" page of the main course site
- **Will post specific suggestions for readings in "All About Circuits, Volume 4"**
- **But readings cover much material you don't need to know**
  - Lecture notes are guide to assignment and test coverage

Getting to AND and OR

- **This is enough to build any circuit**
- **AND = NAND + NOT**



- **OR = NOT + NOT + NAND**



One-input One-output Gates

- **What are the possibilities? 2^2 = 4 choices**

x	f(x)
0	0
1	0

Always false. Boring.

x	f(x)
0	1
1	1

Always true. Boring.

x	f(x)
0	0
1	1

Just like a wire. Boring.

x	f(x)
0	1
1	0

Inverter. Least boring.

Two-input One-output Gates (1)

- **2^4 = 16 possibilities. First some boring ones:**

	0	1
0	0	0
1	0	0

Always 0

	0	1
0	0	0
1	1	1

x

	0	1
0	1	1
1	0	0

!x

	0	1
0	1	1
1	1	1

Always 1

	0	1
0	0	1
1	0	1

y

	0	1
0	1	0
1	1	0

!y

Two-input One-output Gates (2)

- **Symmetric cases:**

	0	1
0	0	0
1	0	1

AND

	0	1
0	0	1
1	1	1

OR

	0	1
0	0	1
1	1	0

XOR, !=

	0	1
0	1	1
1	1	0

NAND

	0	1
0	1	0
1	0	0

NOR

	0	1
0	1	0
1	0	1

XNOR, ==

Two-input One-output Gates (3)

- **Asymmetric cases:**

x\y	0	1
0	1	1
1	0	1

x -> y, !x | y

x\y	0	1
0	1	0
1	1	1

y -> x, !y | x

x\y	0	1
0	0	0
1	1	0

!(x -> y), x & !y

x\y	0	1
0	0	1
1	0	0

!(y -> x), y & !x

Boolean Algebra

- **Boolean algebra**
  - Boolean functions (gates) have a nice algebraic structure
  - But it's different from the rules for arithmetic
  - Same algebraic structure applies to sets, Boolean functions
- **Boolean algebra and other notations**
  - 0 = ⊥
  - 1 = ⊤
  - & = ∧, also sometimes ·
  - | = ∨
  - ^ = ⊕
  - ! = ¬, ~, or a line above, or ' suffix
  - "+" is ambiguous: electrical engineers often use it for OR, but mathematicians use it for XOR

Boolean Identities (1)

- **(x | x) = x**
- **(x | 1) = 1**
- **(x | 0) = x**
- **| is associative**
- **| is commutative**
- **(x | !x) = 1**
- **a & (b | c) = (a & b) | (a & c)**
- **!(a & b) = (!a | !b)**
- **(x & x) = x**
- **(x & 0) = 0**
- **(x & 1) = x**
- **& is associative**
- **& is commutative**
- **(x & !x) = 0**
- **a | (b & c) = (a | b) & (a | c)**
- **!(a | b) = !a & !b**
- **Duality principle: given a formula using &, |, and !, it's also true if you swap & with | and 0 with 1**

Boolean Identities (2)

- **!!x = x**
- **^ is commutative**
- **^ is associative**
- **(x ^ x) = 0**
- **(x ^ 0) = x**
- **(x ^ 1) = !x**
- **!(a ^ b) = (!a ^ b) = (a ^ !b)**

^ forms an Abelian group with identity 0; the inverse of x is x

Universal Sets of Gates

- **A set of gates is *universal* if any Boolean function can be constructed from just gates in the set**
  - {AND, OR, NOT} is universal; proof coming later
  - {AND, NOT} and {OR, NOT} are universal
    - Use DeMorgan's laws
  - {NAND, NOT} is universal
    - Make AND from NAND and NOT
  - {NAND} is universal
    - !x = !(x & x)
  - {NOR, NOT} and {NOR} are universal
  - {AND, OR} is not universal
  - {XOR, NOT} is not universal

Truth Tables

- **Combinational circuit = Boolean function**
  - Combinational: no cycles or memory
  - Outputs are determined just by inputs
- **Finite size**
  - A Boolean function has a finite representation
  - If i input bits, 2^i possible input combinations
  - Can study by just writing the output for all possible inputs
- **Truth table**
  - Standard way to write a function
  - 2^i rows, input combinations in increasing order
  - One column per intermediate or output

Truth Table Example

a	b	c	(a & b)	(a & b) | c
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

Equivalences with a Truth Table

- **Check whether two Boolean formulas are equal**
  - Write truth table covering both
  - Check two columns have all the same entries
- **Advantages**
  - Straightforward
  - No algebraic insight needed
- **Disadvantages**
  - Effort exponential in number of input bits

Equivalence Example

a	b	c	(b & c)	a | (b & c)	(a | b)	(a | c)	(a | b) & (a | c)
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

Combinational Logic Design

- **Given: description of circuit behavior**
  - Word problem, or truth table
- **Goal: efficient circuit implementation**
  - Usually most important: fewest gates and wires
  - Secondarily: reduce number of levels (propagation delay)
- **Kinds of techniques**
  - Up to 6 inputs: pencil and paper approaches
  - Large but structured: split into repeated pieces
  - Large and unstructured: computer algorithm