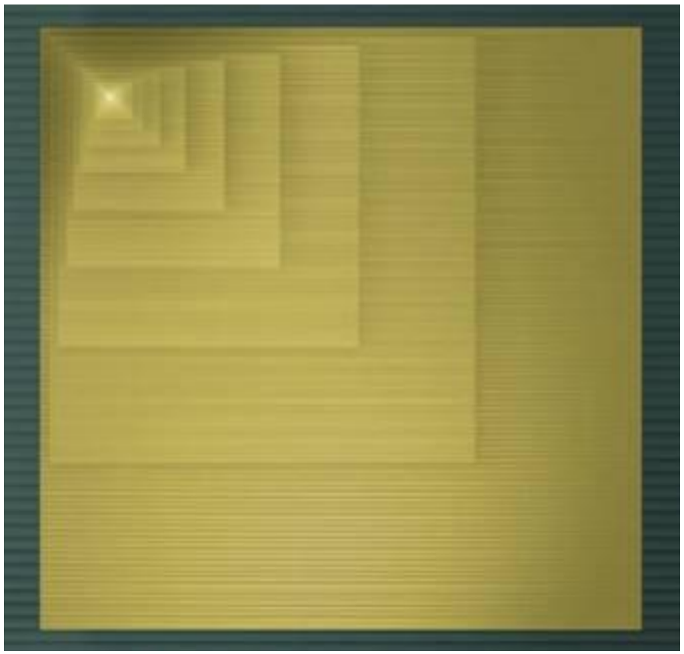


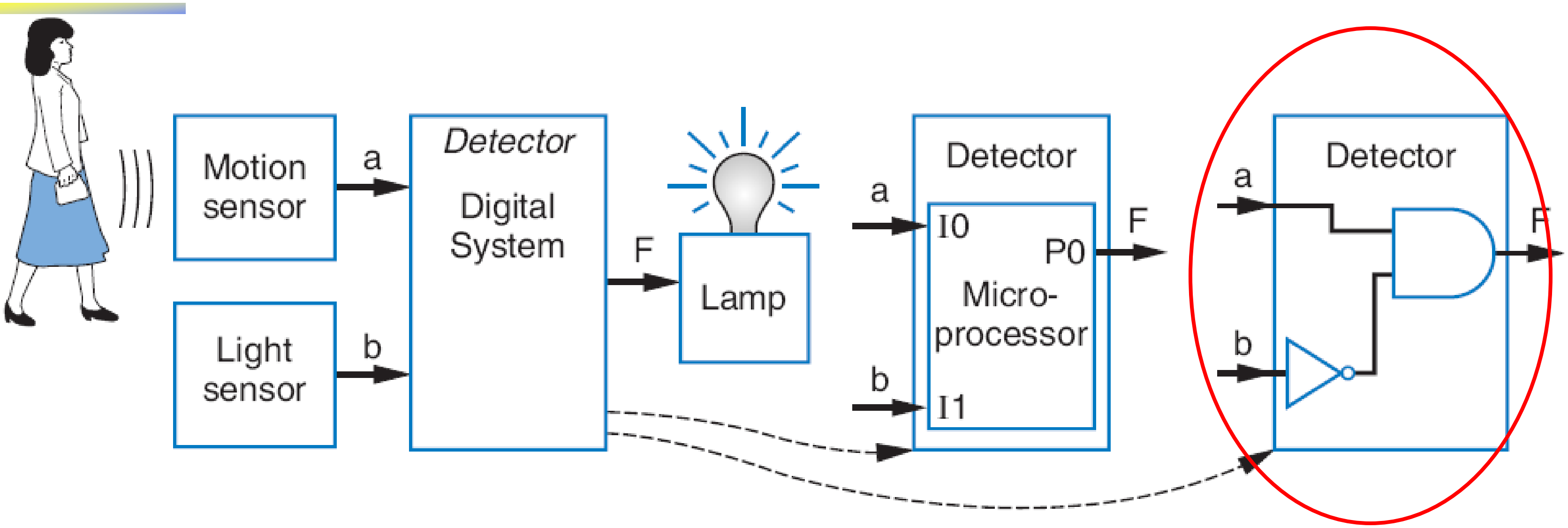
Digital Design

Chapter 2:

Combinational Logic Design

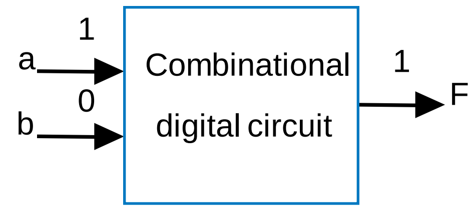
Slides to accompany the textbook *Digital Design*, First Edition,
by Frank Vahid, John Wiley and Sons Publishers, 2007.
http://www.ddvahid.com

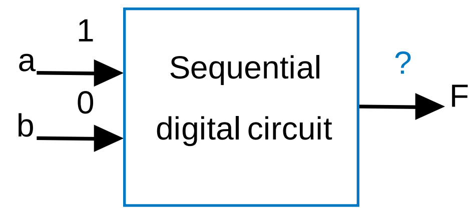
Introduction



Digital circuit

- Let's learn to design digital circuits
- We'll start with a simple form of circuit:
  - ***Combinational circuit***
    - A digital circuit whose outputs depend solely on the *<u>present</u> **<u>combination</u>** of the circuit inputs' values*

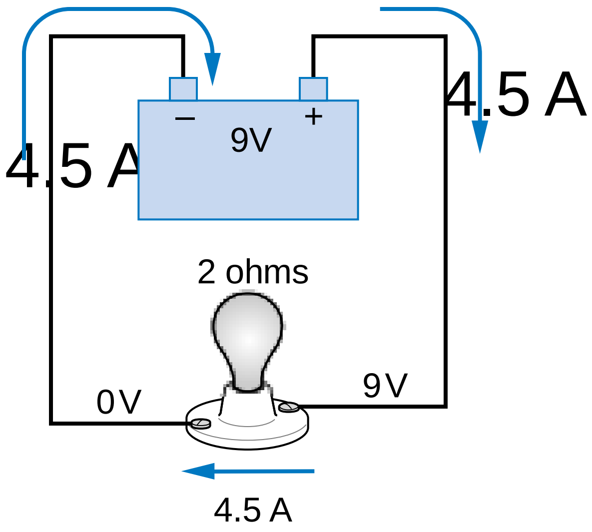




Note: Slides with animation are denoted with a small red "a" near the animated items

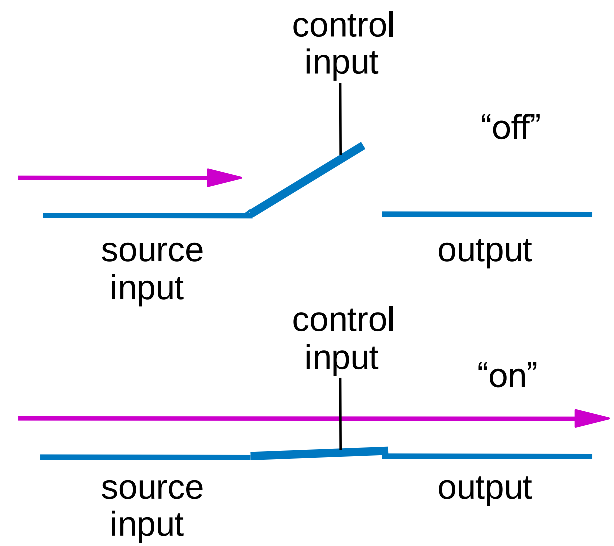
Switches

- Electronic switches are the basis of binary digital circuits
  - Electrical terminology
    - ***Voltage***: Difference in electric potential between two points
      - Analogous to water pressure
    - ***Current***: Flow of charged particles
      - Analogous to water flow
    - ***Resistance***: Tendency of wire to resist current flow
      - Analogous to water pipe diameter
    - V = I * R (Ohm's Law)

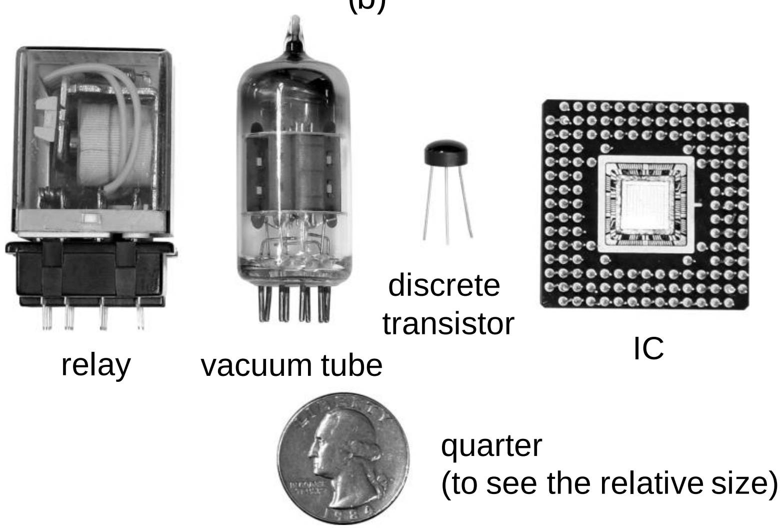


Switches

- A switch has three parts
  - Source input, and output
    - Current wants to flow from source input to output
  - Control input
    - Voltage that controls whether that current can flow
- The amazing shrinking switch
  - 1930s: Relays
  - 1940s: Vacuum tubes
  - 1950s: Discrete transistor
  - 1960s: Integrated circuits (ICs)
    - Initially just a few transistors on IC
    - Then tens, hundreds, thousands...

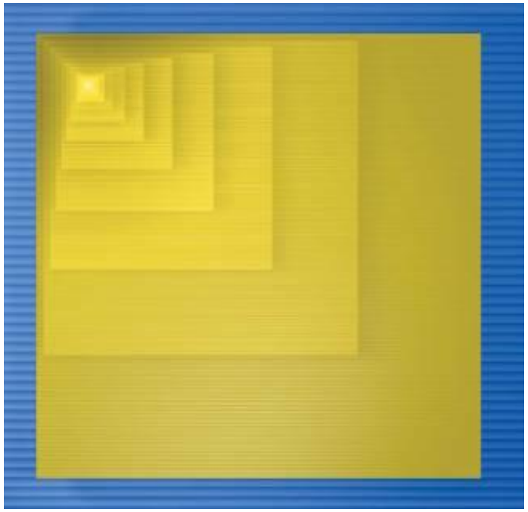


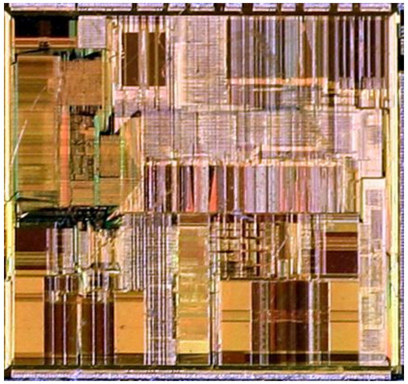
(b)



Moore's Law

- IC capacity doubling about every 18 months for several decades
  - Known as "Moore's Law" after Gordon Moore, co-founder of Intel
    - Predicted in 1965 predicted that components per IC would double roughly every year or so
  - Book cover depicts related phenomena
    - For a particular number of transistors, the IC shrinks by half every 18 months
      - Notice how much shrinking occurs in just about 10 years
      - Enables incredibly powerful computation in incredibly tiny devices
  - Today's ICs hold *billions* of transistors
    - The first Pentium processor (early 1990s) needed only 3 million

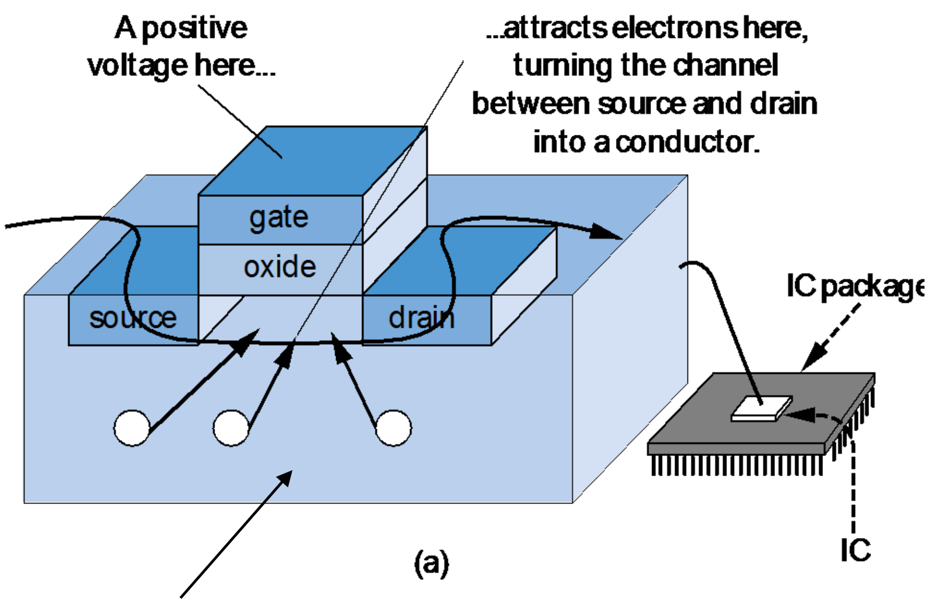




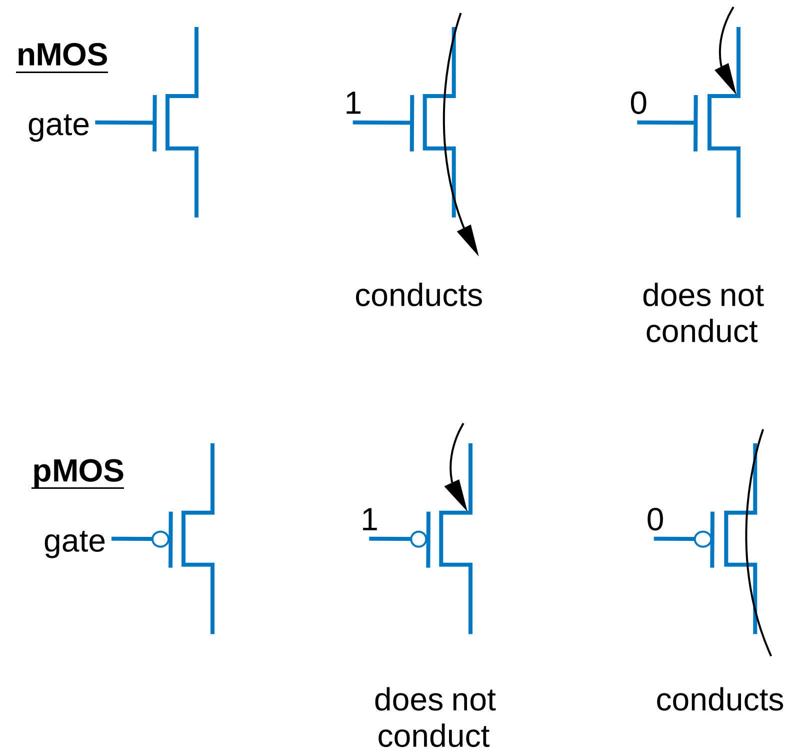
An Intel Pentium processor IC having millions of transistors

The CMOS Transistor

- CMOS transistor
  - Basic switch in modern ICs



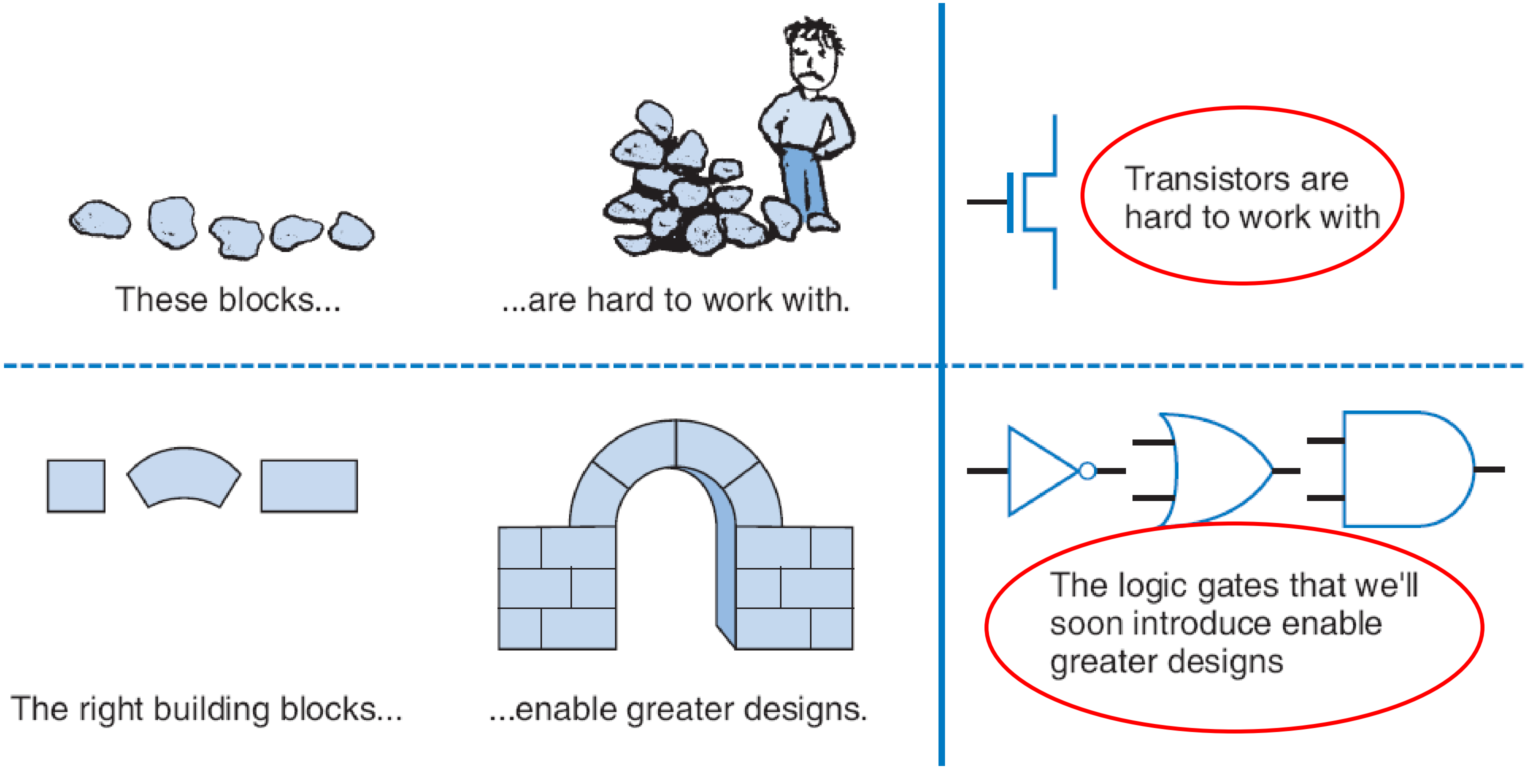
Silicon -- not quite a conductor or insulator:
Semiconductor



Boolean Logic Gates

Building Blocks for Digital Circuits

(Because Switches are Hard to Work With)



- "Logic gates" are better digital circuit building blocks than switches (transistors)
  - Why?...

Boolean Algebra and its Relation to Digital Circuits

- To understand the benefits of “logic gates” vs. switches, we should first understand Boolean algebra
- “Traditional” algebra
  - Variable represent real numbers
  - Operators operate on variables, return real numbers
- ***Boolean Algebra***
  - Variables represent 0 or 1 only
  - Operators return 0 or 1 only
  - Basic operators
    - AND: *a AND b* returns 1 only when both a=1 and b=1
    - OR: *a OR b* returns 1 if either (or both) a=1 or b=1
    - NOT: *NOT a* returns the opposite of a (1 if a=0, 0 if a=1)

a	b	AND
0	0	0
0	1	0
1	0	0
1	1	1

a	b	OR
0	0	0
0	1	1
1	0	1
1	1	1

a	NOT
0	1
1	0

Boolean Algebra and its Relation to Digital Circuits

- Developed mid-1800's by George Boole to formalize human thought
  - Ex: "I'll go to lunch if Mary goes OR John goes, AND Sally does not go."
    - Let F represent my going to lunch (1 means I go, 0 I don't go)
    - Likewise, m for Mary going, j for John, and s for Sally
    - Then **F = (m OR j) AND NOT(s)**
  - Nice features
    - Formally evaluate
      - m=1, j=0, s=1 --> F = (1 OR 0) AND NOT(1) = 1 AND 0 = **<u>0</u>**
    - Formally transform
      - F = (m and NOT(s)) OR (j and NOT(s))
        - » Looks different, but same function
        - » We'll show transformation techniques soon

a	b	AND
0	0	0
0	1	0
1	0	0
1	1	1

a	b	OR
0	0	0
0	1	1
1	0	1
1	1	1

a	NOT
0	1
1	0

Evaluating Boolean Equations

- Evaluate the Boolean equation **F = (a AND b) OR (c AND d)** for the given values of variables a, b, c, and d:
  - Q1: a=1, b=1, c=1, d=0.
    - Answer: F = (1 AND 1) OR (1 AND 0) = 1 OR 0 = 1.
  - Q2: a=0, b=1, c=0, d=1.
    - Answer: F = (0 AND 1) OR (0 AND 1) = 0 OR 0 = 0.
  - Q3: a=1, b=1, c=1, d=1.
    - Answer: F = (1 AND 1) OR (1 AND 1) = 1 OR 1 = 1.

a	b	AND
0	0	0
0	1	0
1	0	0
1	1	1

a	b	OR
0	0	0
0	1	1
1	0	1
1	1	1

a	NOT
0	1
1	0

Converting to Boolean Equations

- Convert the following English statements to a Boolean equation
  - Q1. a is 1 and b is 1.
    - Answer: F = a AND b
  - Q2. either of a or b is 1.
    - Answer: F = a OR b
  - Q3. both a and b are not 0.
    - Answer:
      - (a) Option 1: F = NOT(a) AND NOT(b)
      - (b) Option 2: F = a OR b
  - Q4. a is 1 and b is 0.
    - Answer: F = a AND NOT(b)

Converting to Boolean Equations

- Q1. A fire sprinkler system should spray water if high heat is sensed and the system is set to enabled.
  – Answer: Let Boolean variable h represent “high heat is sensed,” e represent “enabled,” and F represent “spraying water.” Then an equation is: F = h AND e.
- Q2. A car alarm should sound if the alarm is enabled, and either the car is shaken or the door is opened.
  – Answer: Let a represent “alarm is enabled,” s represent “car is shaken,” d represent “door is opened,” and F represent “alarm sounds.” Then an equation is: F = a AND (s OR d).
  – (a) Alternatively, assuming that our door sensor d represents “door is closed” instead of open (meaning d=1 when the door is closed, 0 when open), we obtain the following equation: F = a AND (s OR NOT(d)).

Relating Boolean Algebra to Digital Design

Boolean algebra (mid-1800s)

Boole's intent: formalize human thought

Switches (1930s)

For telephone switching and other electronic uses

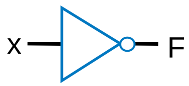
Shannon (1938)

Showed application of Boolean algebra to design of switch-based circuits

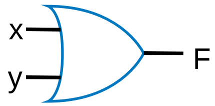
Digital design

Symbol

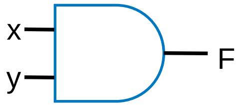
NOT



OR



AND



Truth table

NOT:

x	F
0	1
1	0

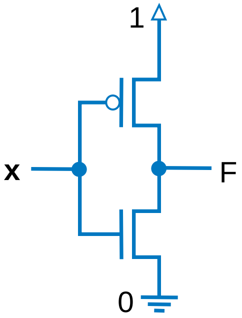
OR:

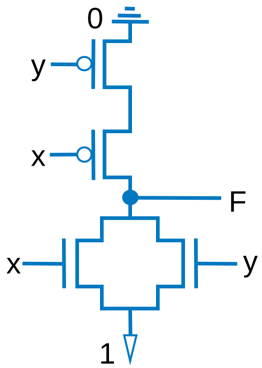
x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

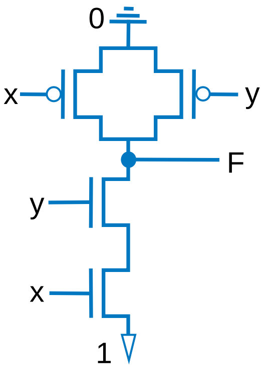
AND:

x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

Transistor circuit



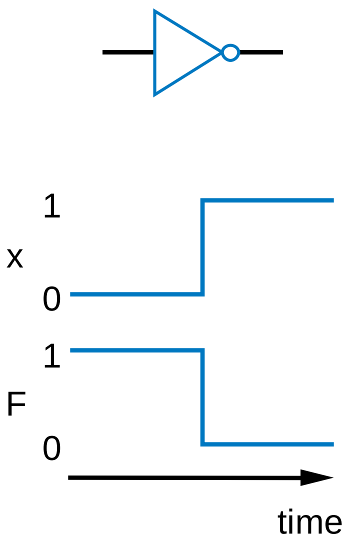


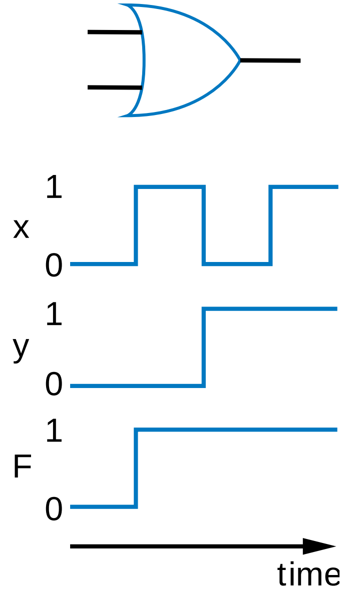


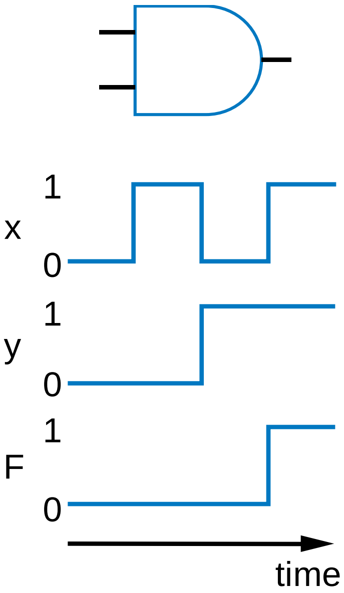
- Implement Boolean operators using transistors
  - Call those implementations ***logic gates***.
  - ***<u>Let's us build circuits by doing math</u>*** -- powerful concept

<u>Note</u>: These OR/AND implementations are inefficient; we'll show why, and show better ones later.

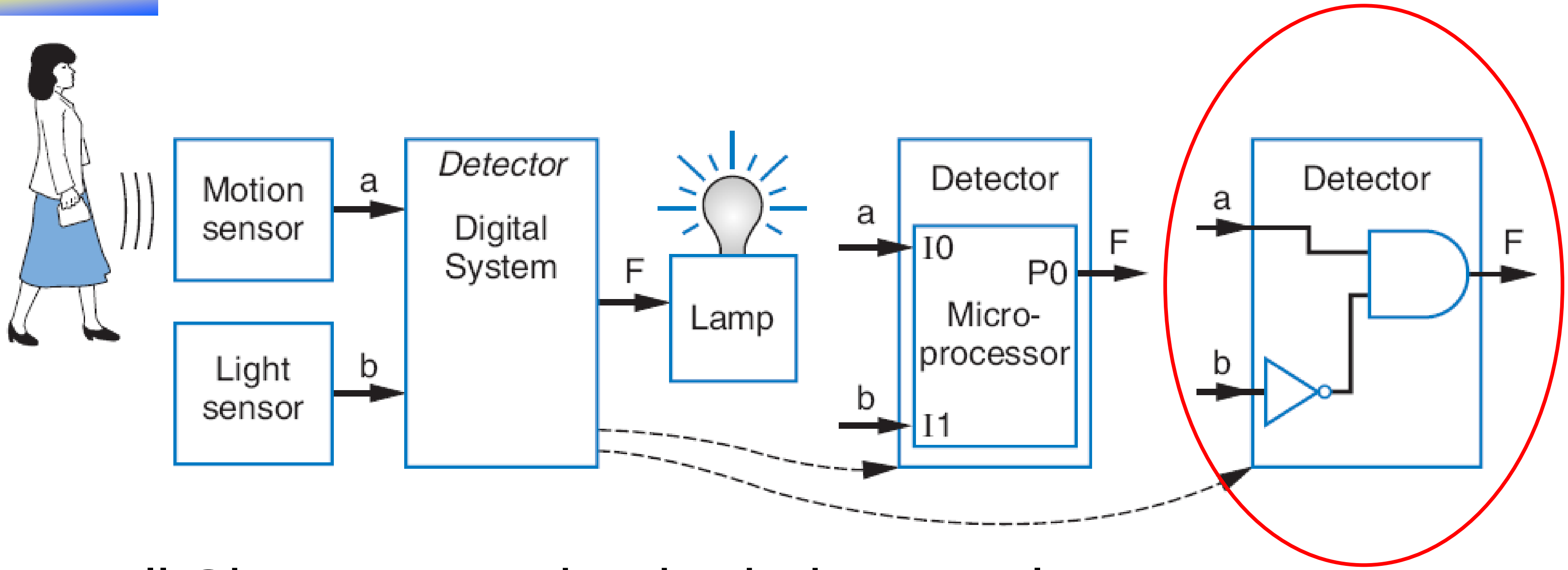
NOT/OR/AND Logic Gate Timing Diagrams







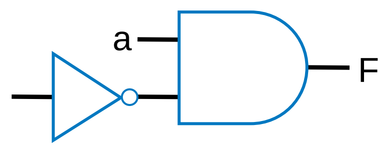
Building Circuits Using Gates



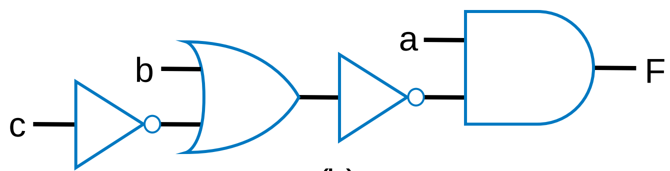
- Recall Chapter 1 motion-in-dark example
  - Turn on lamp (F=1) when motion sensed (a=1) and no light (b=0)
  - F = a AND NOT(b)
  - Build using logic gates, AND and NOT, as shown
  - We just built our first digital circuit!

Example: Converting a Boolean Equation to a Circuit of Logic Gates

- Q: Convert the following equation to logic gates:
  F = a AND NOT(b OR NOT(c))



(a)

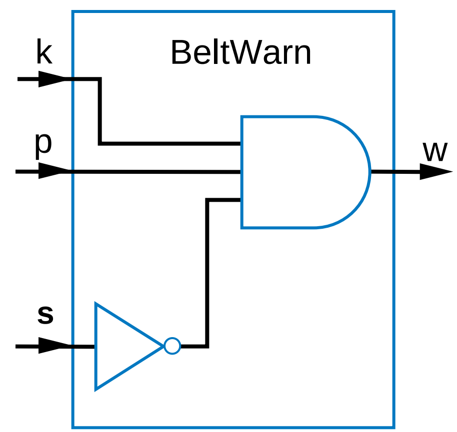


(b)

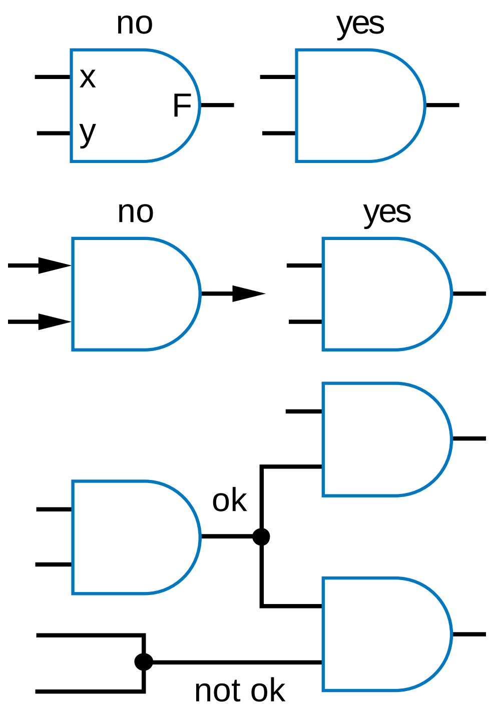
Example: Seat Belt Warning Light System

- Design circuit for warning light
- Sensors
  - s=1: seat belt fastened
  - k=1: key inserted
  - p=1: person in seat
- Capture Boolean equation
  - person in seat, and seat belt not fastened, and key inserted
- Convert equation to circuit
- Notice
  - Boolean algebra enables easy capture as equation and conversion to circuit
    - How design with switches?
    - Of course, logic gates are built from switches, but we think at level of logic gates, not switches

w = p AND NOT(s) AND k



Some Circuit Drawing Conventions



Boolean Algebra

- By defining logic gates based on Boolean algebra, we can use algebraic methods to manipulate circuits
  - So let's learn some Boolean algebraic methods
- Start with notation: Writing a AND b, a OR b, and NOT(a) is cumbersome
  - Use symbols: a * b, a + b, and a' (in fact, a * b can be just ab).
    - Original: w = (p AND NOT(s) AND k) OR t
    - New: w = ps'k + t
      - Spoken as "w equals p and s prime and k, or t"
      - Or even just "w equals p s prime k, or t"
      - s' known as "complement of s"
    - While symbols come from regular algebra, *don't* say "times" or "plus"

Boolean algebra precedence, highest precedence first.

Symbol	Name	Description
( )	Parentheses	Evaluate expressions nested in parentheses first
'	NOT	Evaluate from left to right
*	AND	Evaluate from left to right
+	OR	Evaluate from left to right

Boolean Algebra Operator Precendence

- Evaluate the following Boolean equations, assuming a=1, b=1, c=0, d=1.
  - Q1. F = a * b + c.
    - Answer: * has precedence over +, so we evaluate the equation as F = (1 *1) + 0 = (1) + 0 = 1 + 0 = 1.
  - Q2. F = ab + c.
    - Answer: the problem is identical to the previous problem, using the shorthand notation for *.
  - Q3. F = ab'.
    - Answer: we first evaluate b' because NOT has precedence over AND, resulting in F = 1 * (1') = 1 * (0) = 1 * 0 = 0.
  - Q4. F = (ac)'.
    - Answer: we first evaluate what is inside the parentheses, then we NOT the result, yielding (1*0)' = (0)' = 0' = 1.
  - Q5. F = (a + b') * c + d'.
    - Answer: Inside left parentheses: (1 + (1')) = (1 + (0)) = (1 + 0) = 1. Next, * has precedence over +, yielding (1 * 0) + 1' = (0) + 1'. The NOT has precedence over the OR, giving (0) + (1') = (0) + (0) = 0 + 0 = 0.

Boolean Algebra Terminology

- Example equation: **$F(a,b,c) = a'bc + abc' + ab + c$**
- ***Variable***
  - Represents a value (0 or 1)
  - Three variables: a, b, and c
- ***Literal***
  - Appearance of a variable, in true or complemented form
  - Nine literals: a’, b, c, a, b, c’, a, b, and c
- ***Product term***
  - Product of literals
  - Four product terms: a’bc, abc’, ab, c
- ***Sum-of-products***
  - Equation written as OR of product terms only
  - Above equation is in sum-of-products form. “F = (a+b)c + d” is not.

Boolean Algebra Properties

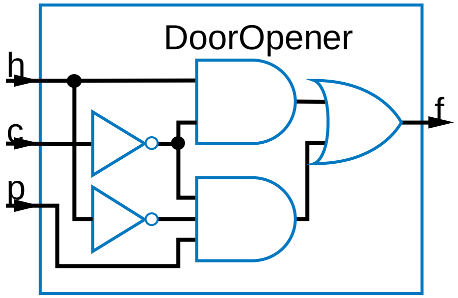
- Commutative
  - a + b = b + a
  - a * b = b * a
- Distributive
  - a * (b + c) = a * b + a * c
  - a + (b * c) = (a + b) * (a + c)
    - (this one is tricky!)
- Associative
  - (a + b) + c = a + (b + c)
  - (a * b) * c = a * (b * c)\'
- Identity
  - 0 + a = a + 0 = a
  - 1 * a = a * 1 = a
- Complement
  - a + a’ = 1
  - a * a’ = 0
- To prove, just evaluate all possibilities

Example uses of the properties

- Show abc’ equivalent to c’ba.
  - Use commutative property:
    - a*b*c’ = a*c’*b = c’*a*b = c’*b*a = c’ba.
- Show abc + abc’ = ab.
  - Use first distributive property
    - abc + abc’ = ab(c+c’).
  - Complement property
    - Replace c+c’ by 1: ab(c+c’) = ab(1).
  - Identity property
    - ab(1) = ab*1 = ab.
- Show x + x’z equivalent to x + z.
  - Second distributive property
    - Replace x+x’z by (x+x’)*(x+z).
  - Complement property
    - Replace (x+x’) by 1,
  - Identity property
    - replace 1*(x+z) by x+z.

Example that Applies Boolean Algebra Properties

- Want automatic door opener circuit (e.g., for grocery store)
  - Output: f=1 opens door
  - Inputs:
    - p=1: person detected
    - h=1: switch forcing hold open
    - c=1: key forcing closed
  - Want open door when
    - h=1 and c=0, or
    - h=0 and p=1 and c=0
  - Equation: f = hc’ + h’pc’



- Found inexpensive chip that computes:
    - f = c’hp + c’hp’ + c’h’p
  - Can we use it?
    - Is it the same as f = c’(p+h)?
- Use Boolean algebra:

f = c’hp + c’hp’ + c’h’p

f = c’h(p + p’) + c’h’p (by the distributive property)

f = c’h(1) + c’h’p (by the complement property)

f = c’h + c’h’p (by the identity property)

f = hc’ + h’pc’ (by the commutative property)

f =c’(h+h’p)

f = c’(h+h’)(h+p)

f=c′(p+h)

Same!

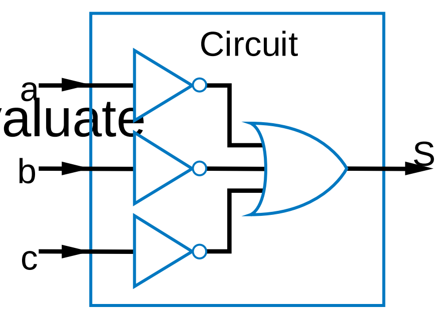
Boolean Algebra: Additional Properties

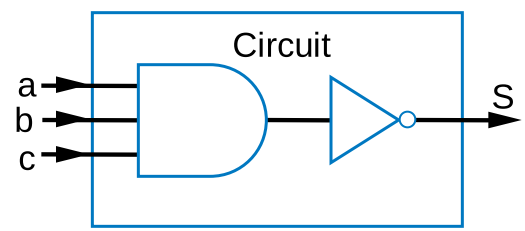
- Null elements
  - $a + 1 = 1$
  - $a * 0 = 0$
- Idempotent Law (等冪)
  - $a + a = a$
  - $a * a = a$
- Involution Law (乘方)
  - $(a')' = a$
- DeMorgan's Law
  - $(a + b)' = a'b'$
  - $(ab)' = a' + b'$
  - Very useful!
- To prove, just evaluate all possibilities

Aircraft lavatory sign example

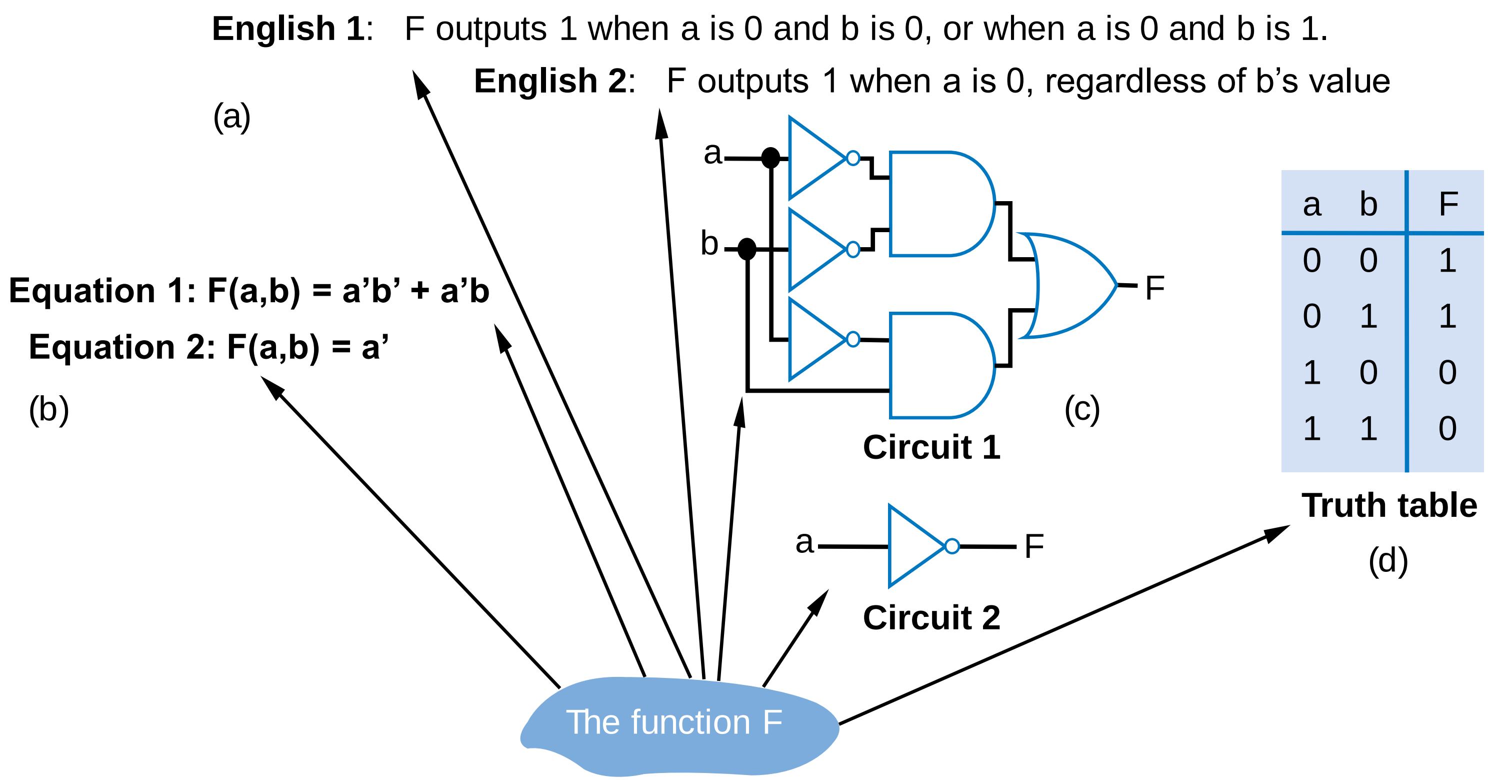
- Behavior
  - Three lavatories, each with sensor (a, b, c), equals 1 if door locked
  - Light "Available" sign (S) if any lavatory available
- Equation and circuit
  - $S = a' + b' + c'$
- Transform
  - $(abc)' = a'+b'+c'$ (by DeMorgan's Law)
  - $S = (abc)'$
- New equation and circuit

- Alternative: Instead of lighting "Available," light "Occupied"
  - Opposite of "Available" function $S = a' + b' + c'$
  - So $S' = (a' + b' + c')'$
    - $S' = (a')' * (b')' * (c')'$ (by DeMorgan's Law)
    - $S' = a * b * c$ (by Involution Law)
  - Makes intuitive sense
    - Occupied if all doors are locked





Representations of Boolean Functions



a	b	F
0	0	1
0	1	1
1	0	0
1	1	0

- A function can be represented in different ways
  - Above shows seven representations of the same functions F(a,b), using four different methods: English, Equation, Circuit, and Truth Table

Truth Table Representation of Boolean Functions

- Define value of F for each possible combination of input values
  - 2-input function: 4 rows
  - 3-input function: 8 rows
  - 4-input function: 16 rows
- Q: Use truth table to define function F(a,b,c) that is 1 when abc is 5 or greater in binary

a	b	F
0	0	
0	1	
1	0	
1	1	

(a)

a	b	c	F
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

(b)

a	b	c	d	F
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

(c)

a

a	b	c	F
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
0	1	1	**0**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

Converting among Representations

- Can convert from any representation to any other
- Common conversions
  - Equation to circuit (we did this earlier)
  - Truth table to equation (which we can convert to circuit)
    - Easy -- just OR each input term that should output 1
  - Equation to truth table
    - Easy -- just evaluate equation for each input combination (row)
    - Creating intermediate columns helps

Inputs		Outputs	Term
a	b	F	F = sum of
0	0	1	a'b'
0	1	1	a'b
1	0	0	
1	1	0	

F = a'b' + a'b

Q: Convert to equation

a	b	c	F	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	0	
1	0	0	0	
1	0	1	1	ab'c
1	1	0	1	abc'
1	1	1	1	abc

a

F = ab'c + abc' + abc

Q: Convert to truth table: F = a'b' + a'b

a

Inputs				Output
a	b	a' b'	a' b	F
0	0	1	0	**1**
0	1	0	1	**1**
1	0	0	0	**0**
1	1	0	0	**0**

Standard Representation: Truth Table

- How can we determine if two functions are the same?
  - Recall automatic door example
    - Same as f = hc' + h'pc'?
    - Used algebraic methods
    - But if we failed, does that prove *not* equal? No.
- Solution: Convert to truth tables
  - Only ONE truth table representation of a given function
    - ***Standard*** representation -- for given function, only one version in standard form exists

f = c'hp + c'hp' + c'h'

f = c'h(p + p') + c'h'p

f = c'h(1) + c'h'p

f = c'h + c'h'p

(what if we stopped here?)

f = hc' + h'pc'

Q: Determine if F=ab+a' is same function as F=a'b'+a'b+ab, by converting each to truth table first

F = ab + 'a

a	b	F
0	0	1
0	1	1
1	0	0
1	1	1

F = a'b' + a'b + ab

a	b	F
0	0	1
0	1	1
1	0	0
1	1	1

Same

a

Canonical Form -- Sum of Minterms

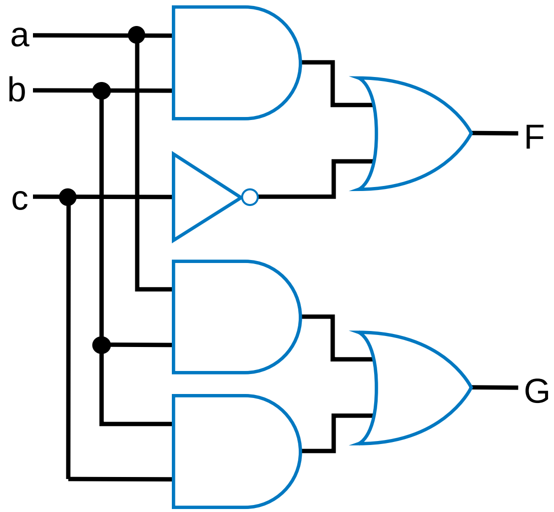
- Truth tables too big for numerous inputs
- Use standard form of equation instead
  - Known as ***canonical form***
  - Regular algebra: group terms of polynomial by power
    - $ax^2 + bx + c$ $(3x^2 + 4x + 2x^2 + 3 + 1 \text{ --> } 5x^2 + 4x + 4)$
  - Boolean algebra: create sum of minterms
    - ***Minterm***: product term with every function literal appearing exactly once, in true or complemented form
    - Just multiply-out equation until sum of product terms
    - Then expand each term until all terms are minterms

Q: Determine if F(a,b)=ab+a’ is same function as F(a,b)=a’b’+a’b+ab, by converting first equation to canonical form (second already in canonical form)

F = ab+a’ (already sum of products)
F = ab + a’(b+b’) (expanding term)
F = ab + a’b + a’b’ (SAME -- same three terms as other equation)

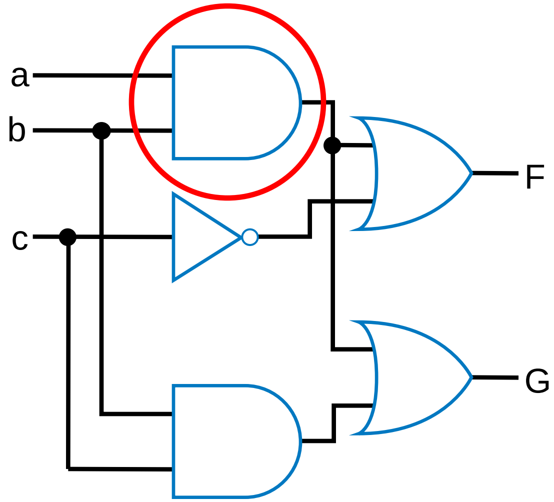
Multiple-Output Circuits

- Many circuits have more than one output
- Can give each a separate circuit, or can share gates
- Ex: F = ab + c', G = ab + bc



(a)

Option 1: Separate circuits



(b)

Option 2: Shared gates

Multiple-Output Example: BCD (binary-coded decimal) to 7-Segment Converter

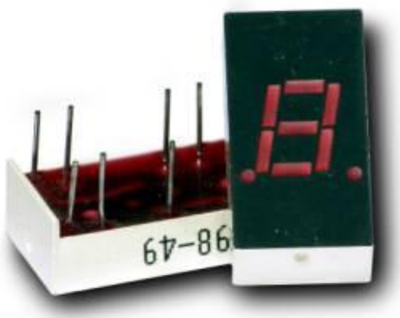
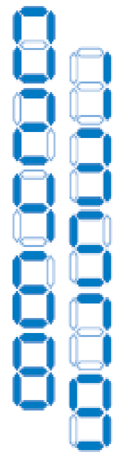
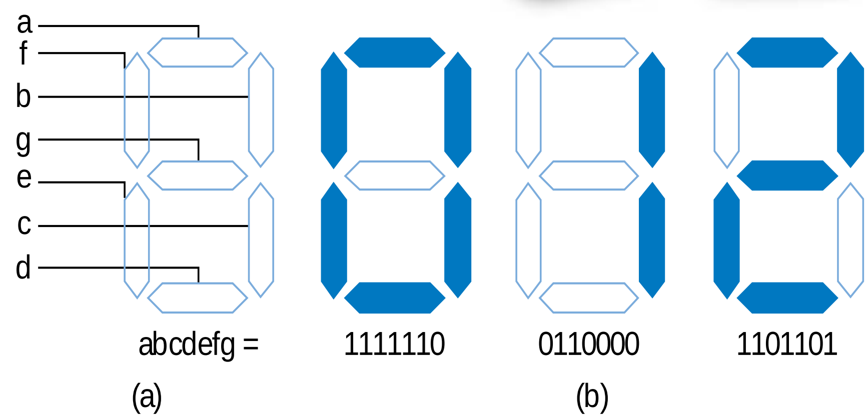


TABLE 2-4 4-bit binary number to seven-segment display truth table

w	x	y	z	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0





a = w'x'y'z' + w'x'yz' + w'x'yz + w'xy'z + w'xyz' + w'xyz + wx'y'z' + wx'y'z

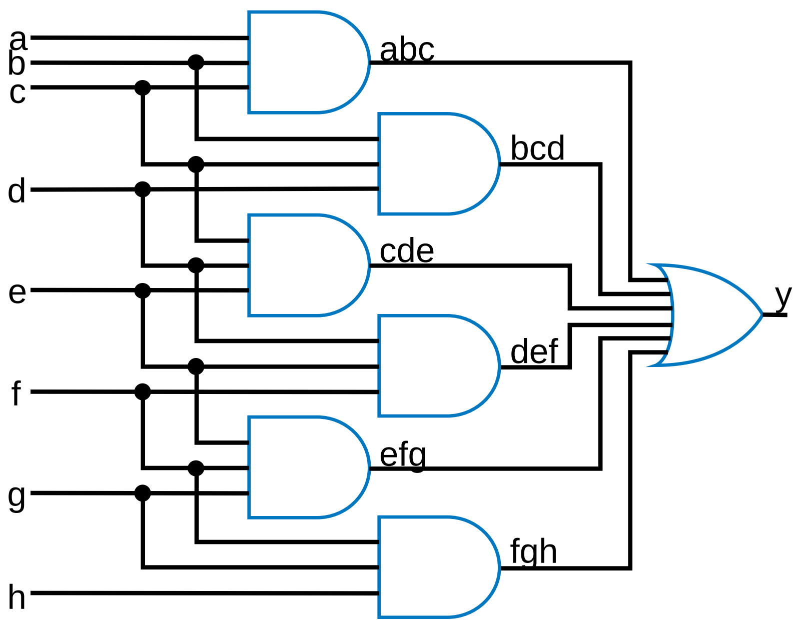
b = w'x'y'z' + w'x'y'z + w'x'yz' + w'x'yz + w'xy'z' + w'xyz + wx'y'z' + wx'y'z

Combinational Logic Design Process

	Step	Description
Step 1	**Capture** the function	Create a truth table or equations, ***whichever is most natural for the given problem***, to describe the desired behavior of the combinational logic.
Step 2	**Convert** to equations	This step is only necessary if you captured the function using a truth table instead of equations. Create an equation for each output by ORing all the minterms for that output. Simplify the equations if desired.
Step 3	**Implement** as a gate-based circuit	For each output, create a circuit corresponding to the output's equation. (Sharing gates among multiple outputs is OK optionally.)

Example: Three 1s Detector

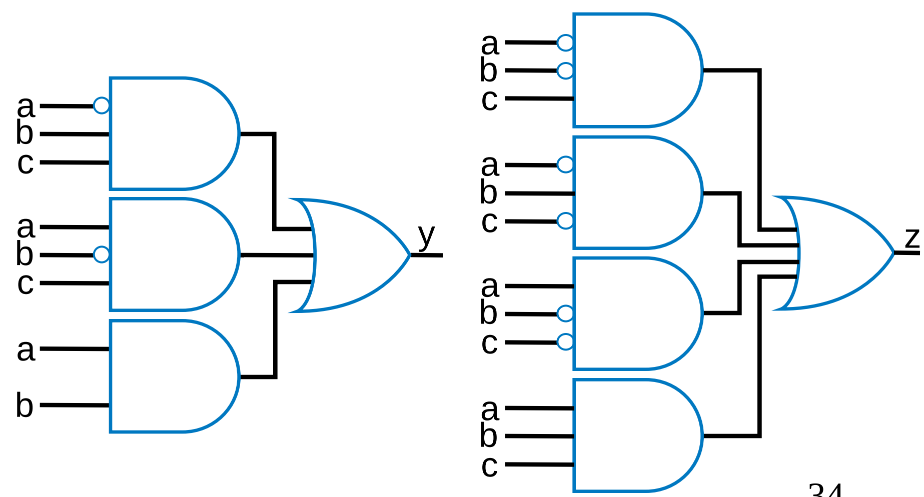
- Problem: Detect three consecutive 1s in 8-bit input: abcdefgh
    - 000**111**01 → 1 10101011 → 0 **111**10000 → 1
  - **Step 1: Capture** the function
    - Truth table or equation?
      - Truth table too big: 2^8=256 rows
      - Equation: create terms for each possible case of three consecutive 1s
    - **y = abc + bcd + cde + def + efg + fgh**
  - **Step 2: Convert** to equation -- already done
  - **Step 3: Implement** as a gate-based circuit



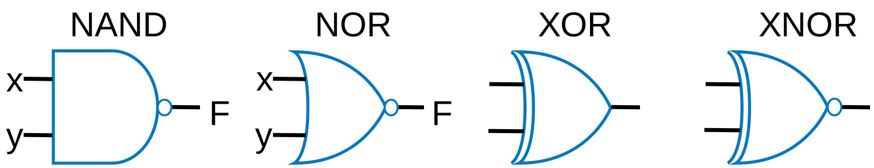
Example: Number of 1s Count

- Problem: Output in binary on two outputs yz the number of 1s on three inputs
    - 010 → 01   101 → 10   000 → 00
  - **Step 1: Capture** the function
    - Truth table or equation?
      - Truth table is straightforward
  - **Step 2: Convert** to equation
    - y = a'bc + ab'c + abc' + abc
    - z = a'b'c + a'bc' + ab'c' + abc
  - **Step 3: Implement** as a gate-based circuit

Inputs			(# of 1s)	Outputs	
a	b	c		y	z
0	0	0	(0)	0	0
0	0	1	(1)	0	1
0	1	0	(1)	0	1
0	1	1	(2)	1	0
1	0	0	(1)	0	1
1	0	1	(2)	1	0
1	1	0	(2)	1	0
1	1	1	(3)	1	1



More Gates

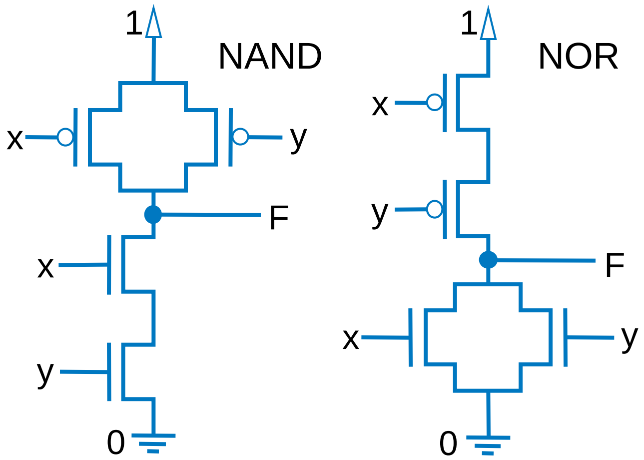


x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

x	y	F
0	0	1
0	1	0
1	0	0
1	1	0

x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

x	y	F
0	0	1
0	1	0
1	0	0
1	1	1

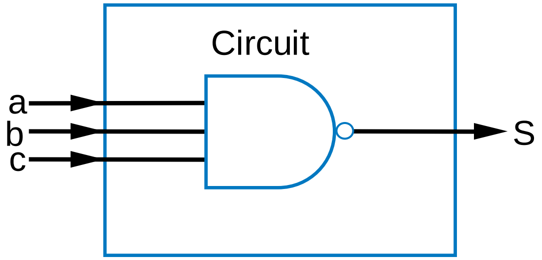


- NAND: Opposite of AND (“NOT AND”)
- NOR: Opposite of OR (“NOT OR”)
- XOR: Exactly 1 input is 1, for 2-input XOR. (For more inputs -- odd number of 1s)
- XNOR: Opposite of XOR (“NOT XOR”)

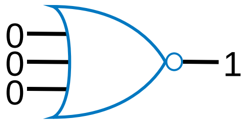
- NAND same as AND with power & ground switched
  - Why? nMOS conducts 0s well, but not 1s (reasons beyond our scope) -- so NAND more efficient
- Likewise, NOR same as OR with power/ground switched
- AND in CMOS: NAND with NOT
- OR in CMOS: NOR with NOT
- So NAND/NOR more common

More Gates: Example Uses

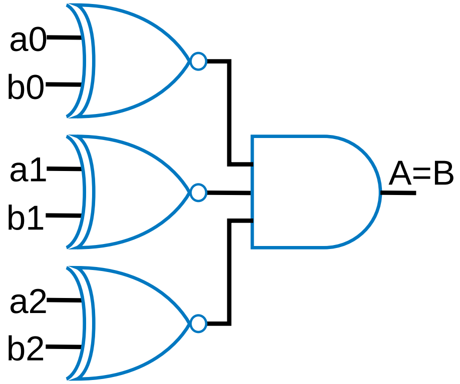
- Aircraft lavatory sign example
  - S = (abc)'



- Detecting all 0s
  - Use NOR



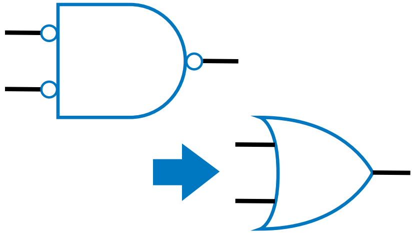
- Detecting equality
  - Use XNOR



- Detecting odd # of 1s
  - Use XOR
  - Useful for generating "parity" bit common for detecting errors

Completeness of NAND

- Any Boolean function can be implemented *using just NAND gates.* Why?
  - Need AND, OR, and NOT
  - NOT: 1-input NAND (or 2-input NAND with inputs tied together)
  - AND: NAND followed by NOT
  - OR: NAND preceded by NOTs



- Likewise for NOR

Number of Possible Boolean Functions

- How many possible functions of 2 variables?
  - 2^2 rows in truth table, 2 choices for each
  - $2^{(2^2)} = 2^4 = 16$ possible functions
- N variables
  - 2^N rows
  - $2^{(2^N)}$ possible functions

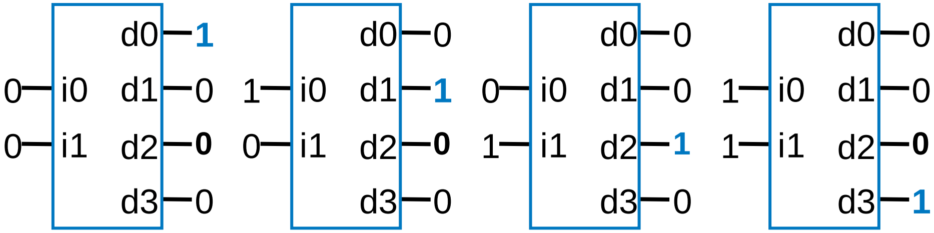
a	b	F	
0	0	0 or 1	2 choices
0	1	0 or 1	2 choices
1	0	0 or 1	2 choices
1	1	0 or 1	2 choices

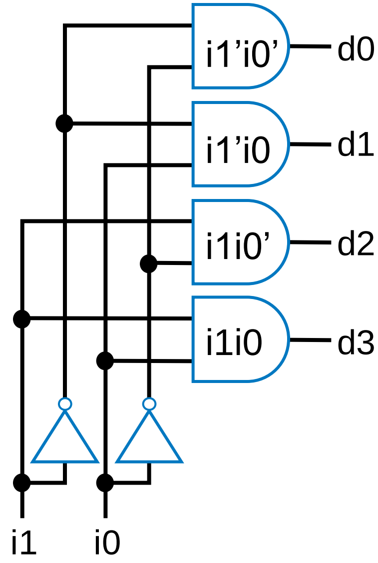
$2^4 = 16$
possible functions

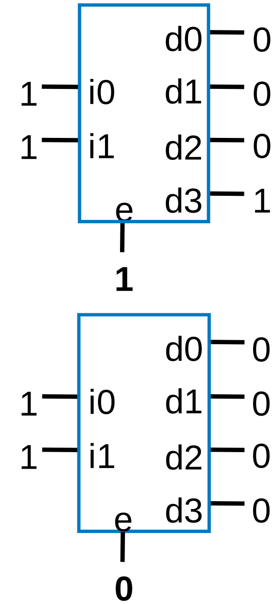
a	b	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13	f14	f15
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
0	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**
1	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**
1	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**
		0	a AND b		a		b	a XOR b	a OR b	a NOR b	a XNOR b	b’		a’		a NAND b	1

Decoders and Muxes

- **Decoder**: Popular combinational logic building block, in addition to logic gates
  - Converts input binary number to one high output
- 2-input decoder: four possible input binary numbers
  - So has four outputs, one for each possible input binary number
- Internal design
  - AND gate for each output to detect input combination
- Decoder with enable e
  - Outputs all 0 if e=0
  - Regular behavior if e=1
- n-input decoder: 2^n outputs

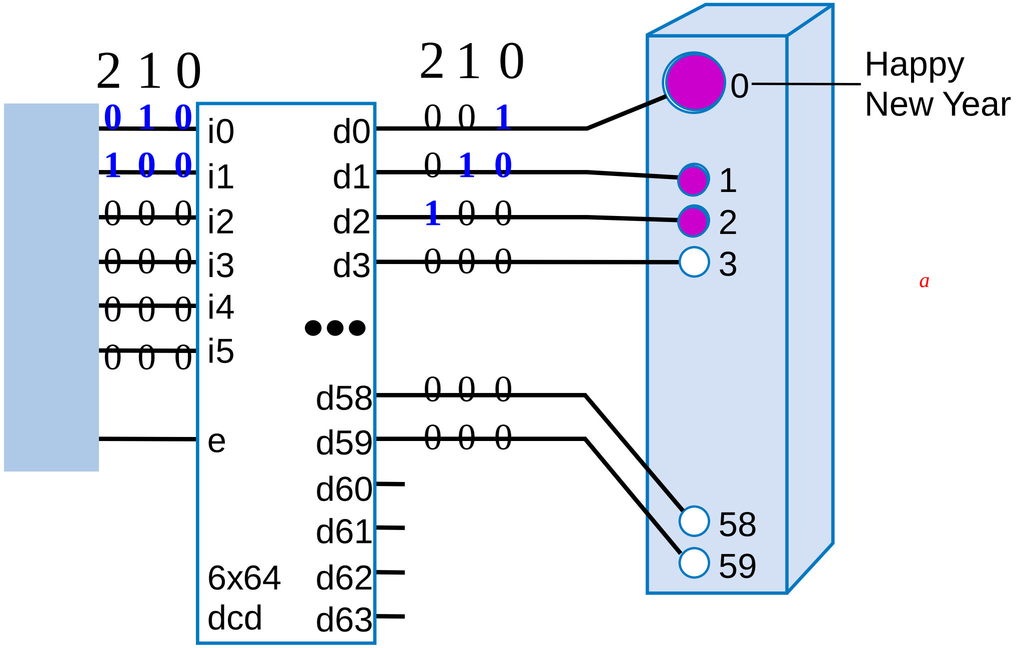






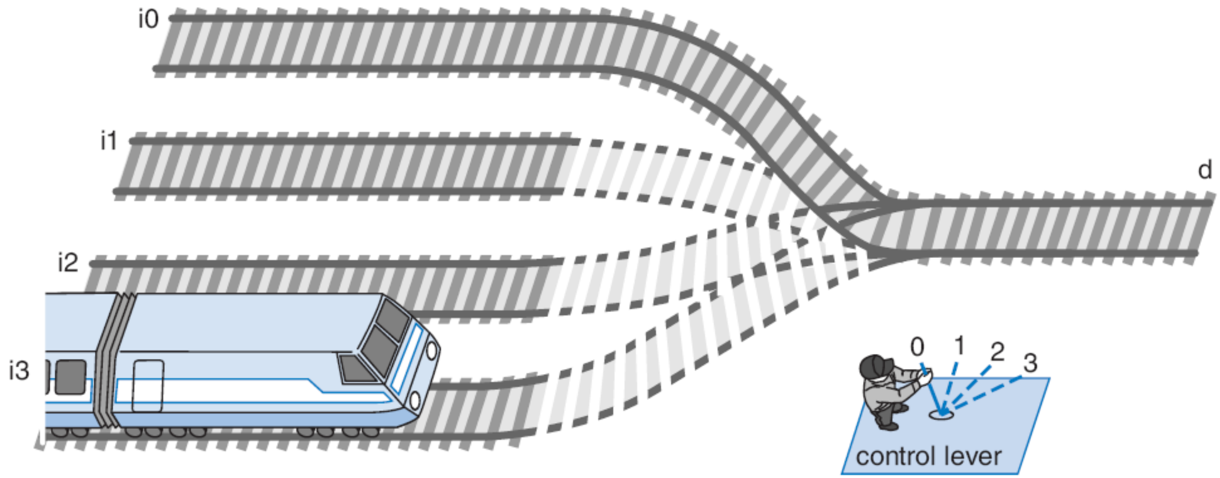
Decoder Example

- New Year's Eve Countdown Display
  - Microprocessor counts from 59 down to 0 in binary on 6-bit output
  - Want illuminate one of 60 lights for each binary number
  - Use 6x64 decoder
    - 4 outputs unused

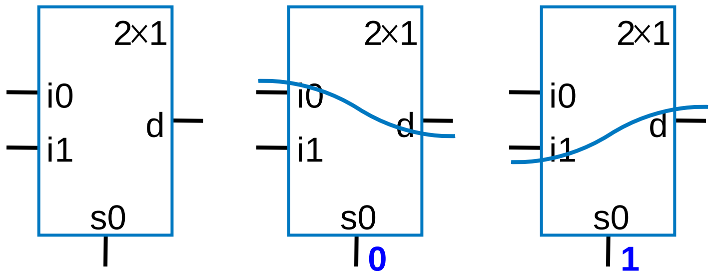


Multiplexor (Mux)

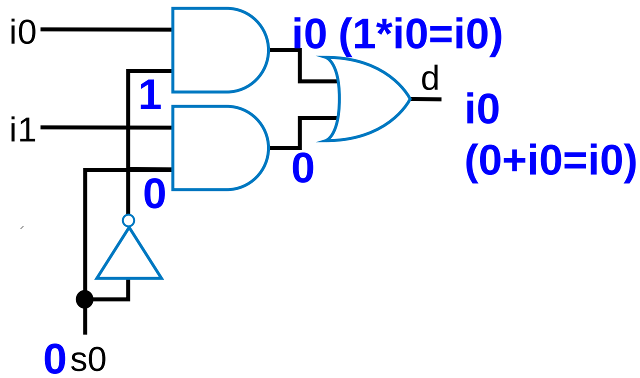
- Mux: Another popular combinational building block
  - Routes one of its N data inputs to its one output, based on binary value of select inputs
    - 4 input mux → needs 2 select inputs to indicate which input to route through
    - 8 input mux → 3 select inputs
    - N inputs → $\log_2(N)$ selects
  - Like a railyard switch

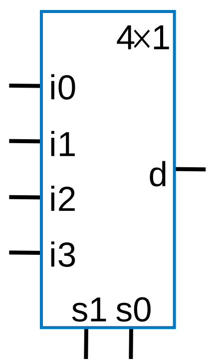


Mux Internal Design

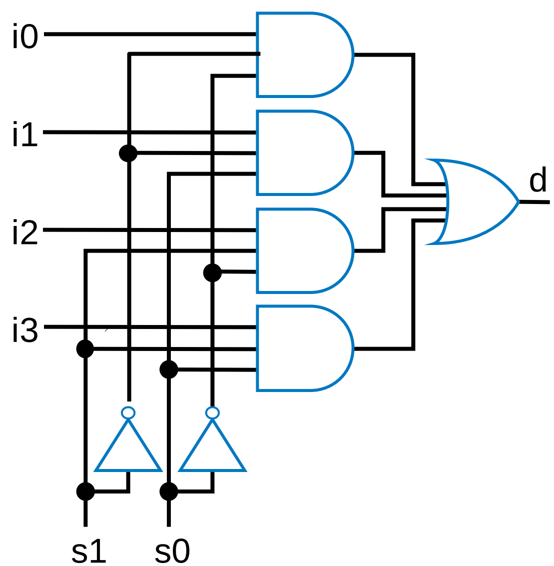


2x1 mux



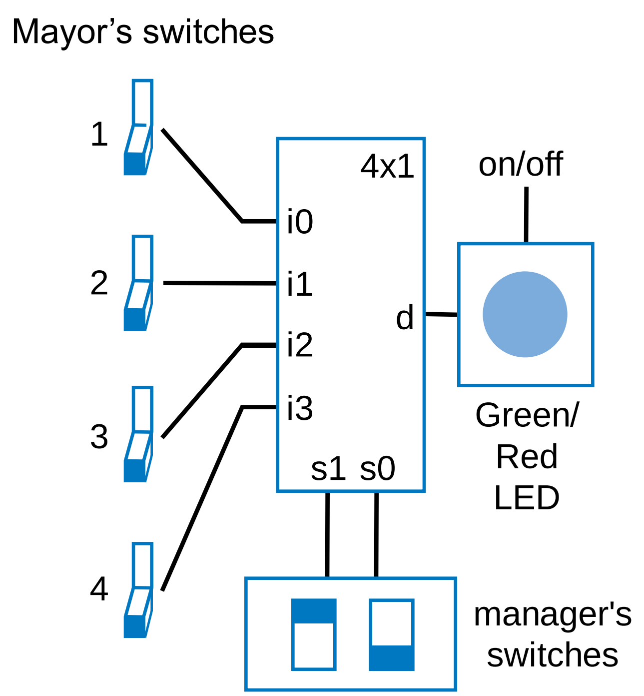


4x1 mux

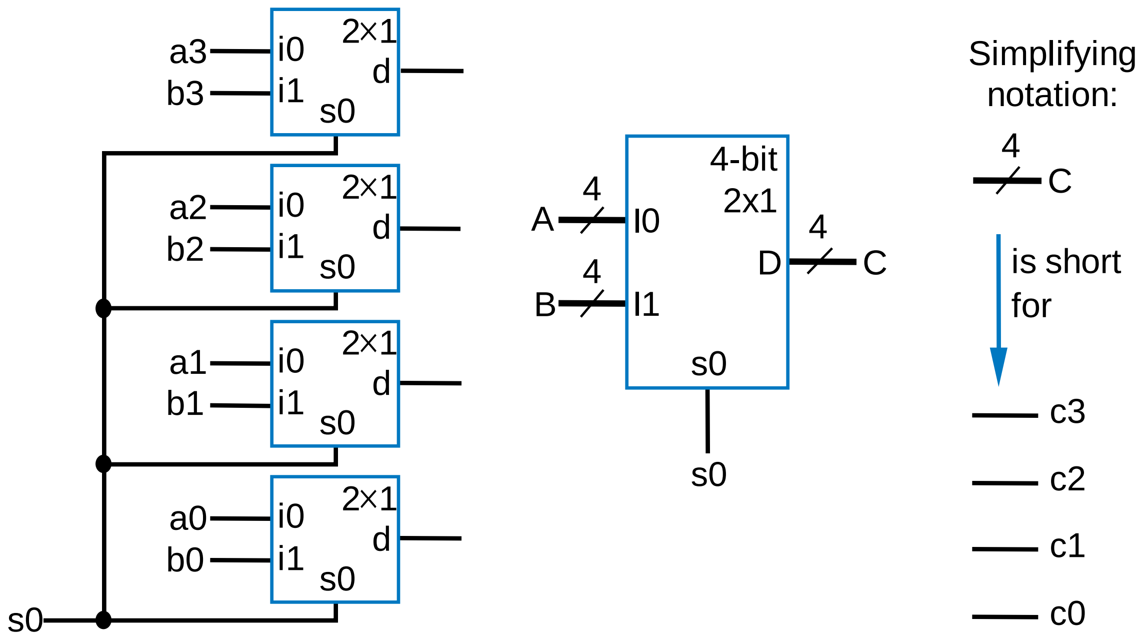


Mux Example

- City mayor can set four switches up or down, representing his/her vote on each of four proposals, numbered 0, 1, 2, 3
- City manager can display any such vote on large green/red LED (light) by setting two switches to represent binary 0, 1, 2, or 3
- Use 4x1 mux

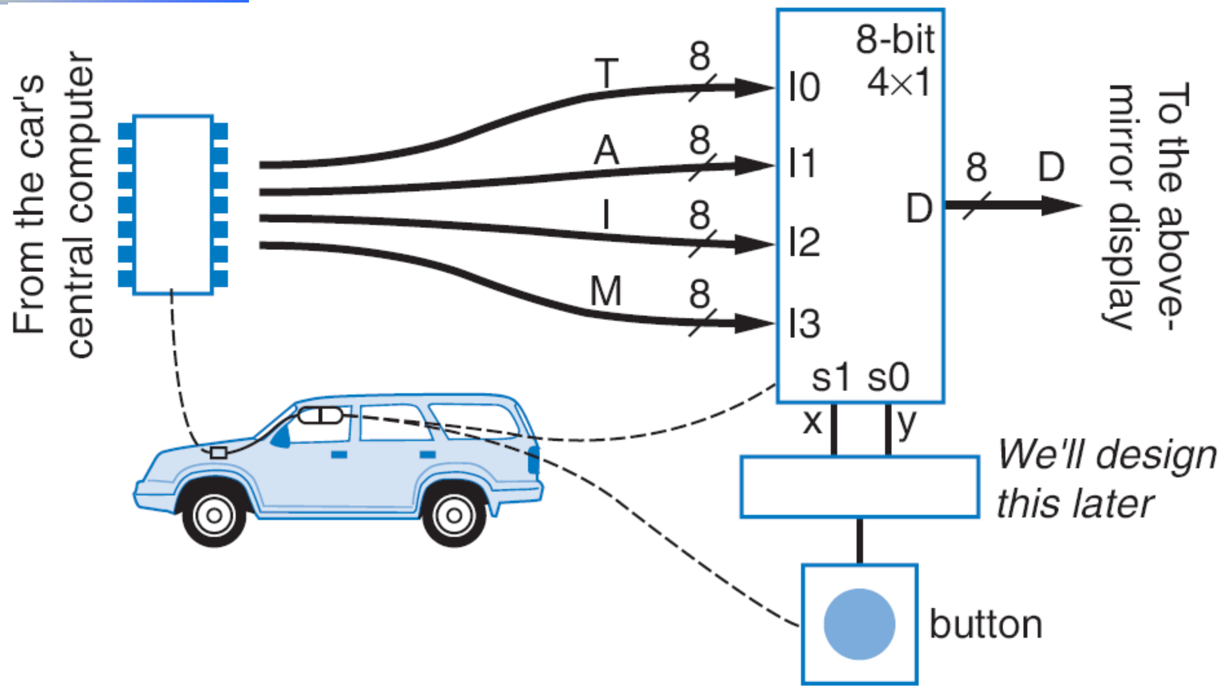


Muxes Commonly Together -- N-bit Mux



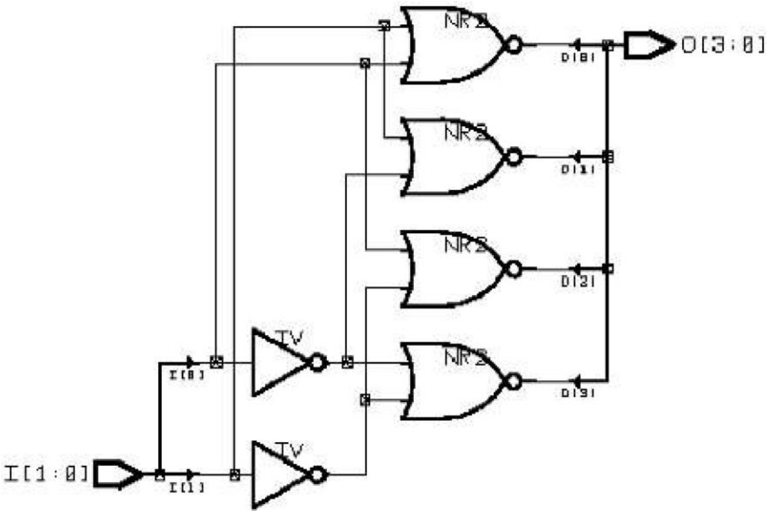
- Ex: Two 4-bit inputs, A (a3 a2 a1 a0), and B (b3 b2 b1 b0)
  - 4-bit 2x1 mux (just four 2x1 muxes sharing a select line) can select between A or B

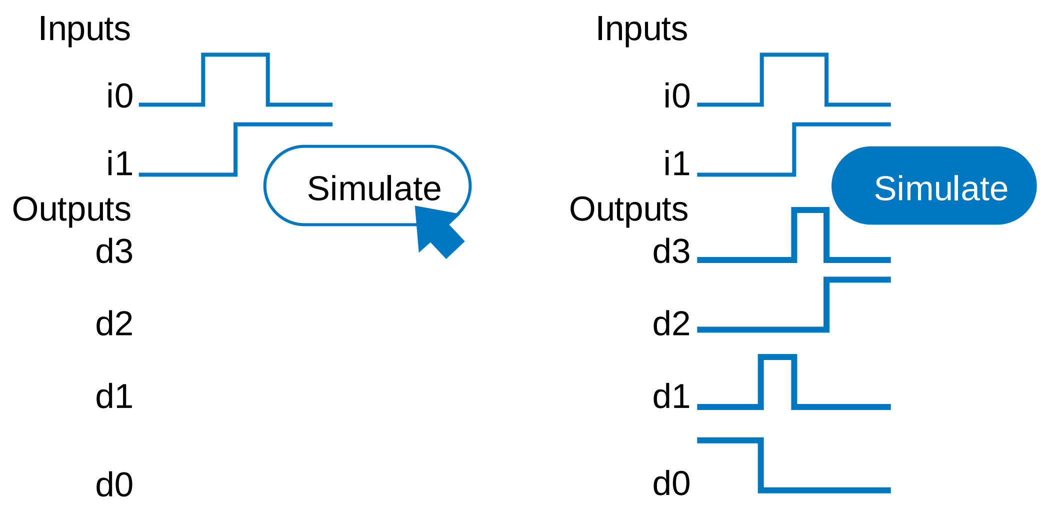
N-bit Mux Example



- Four possible display items
  - Temperature (T), Average miles-per-gallon (A), Instantaneous mpg (I), and Miles remaining (M) -- each is 8-bits wide
  - Choose which to display using two inputs x and y
  - Use 8-bit 4x1 mux

Additional Considerations
Schematic Capture and Simulation

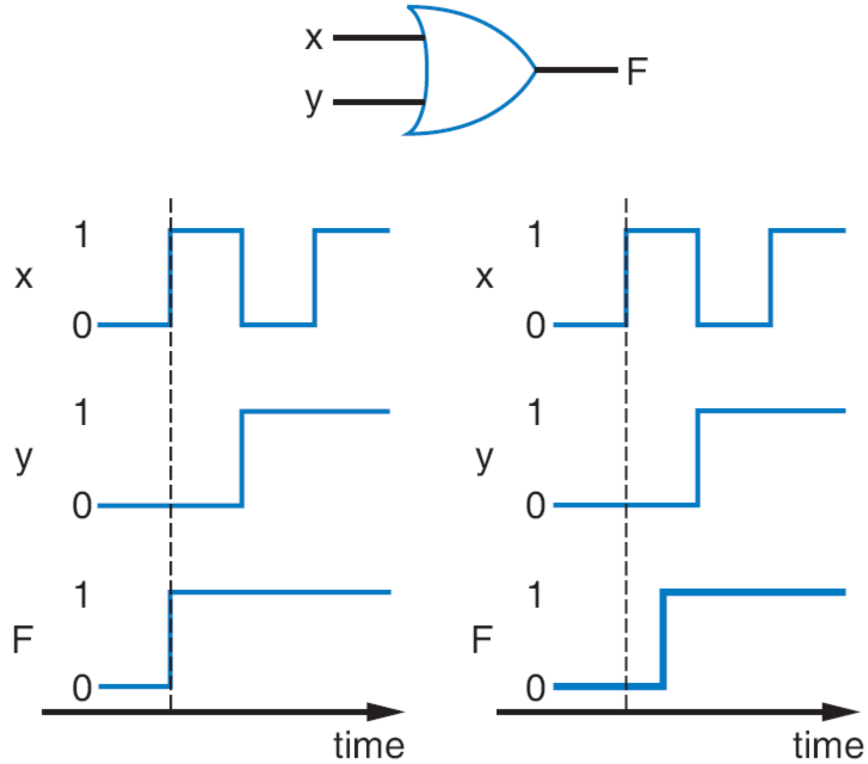




- **Schematic capture**
  - Computer tool for user to capture logic circuit graphically
- **Simulator**
  - Computer tool to show what circuit outputs would be for given inputs
    - Outputs commonly displayed as **waveform**

Additional Considerations

Non-Ideal Gate Behavior -- Delay



- Real gates have some delay
  - Outputs don't change immediately after inputs change

Chapter Summary

- Combinational circuits
  - Circuit whose outputs are function of present inputs
    - No “state”
- Switches: Basic component in digital circuits
- Boolean logic gates: AND, OR, NOT -- Better building block than switches
  - Enables use of Boolean algebra to design circuits
- Boolean algebra: uses true/false variables/operators
- Representations of Boolean functions: Can translate among
- Combinational design process: Translate from equation (or table) to circuit through well-defined steps
- More gates: NAND, NOR, XOR, XNOR also useful
- Muxes and decoders: Additional useful combinational building blocks