

Digital Design

Chapter 2: Combinational Logic Design

Slides to accompany the textbook *Digital Design, with RTL Design, VHDL, and Verilog*, 2nd Edition,
by Frank Vahid, John Wiley and Sons Publishers, 2010.
<http://www.ddvahid.com>

Copyright © 2010 Frank Vahid

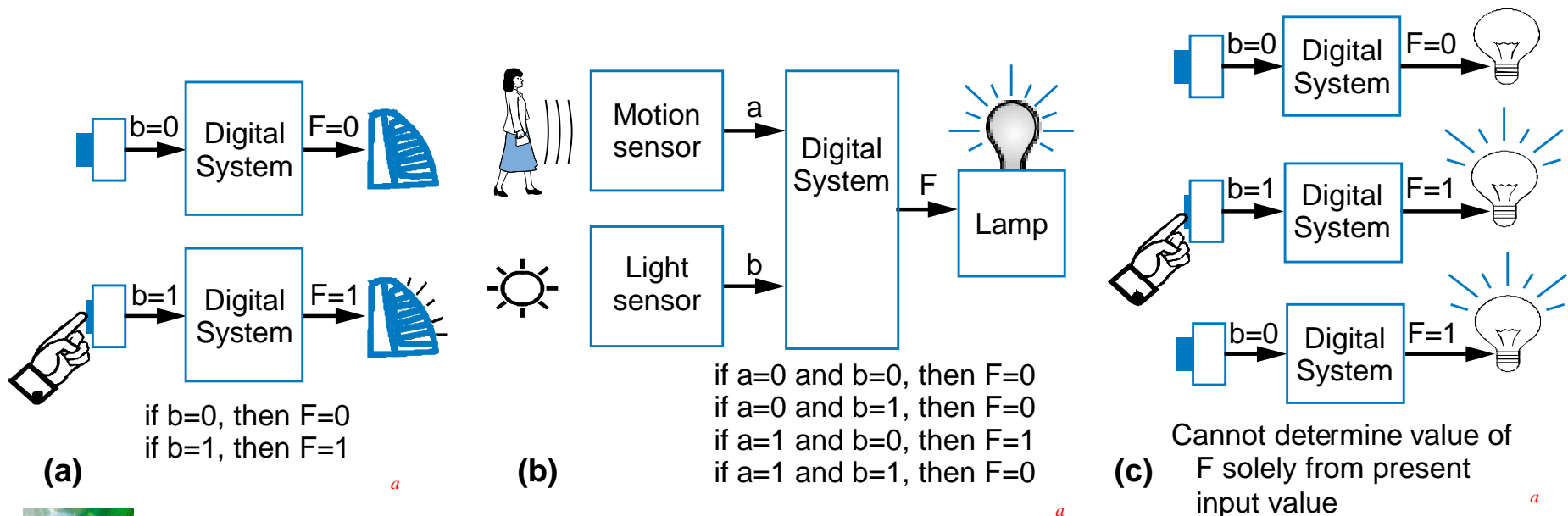
Instructors of courses requiring Vahid's *Digital Design* textbook (published by John Wiley and Sons) have permission to modify and use these slides for customary course-related activities, subject to keeping this copyright notice in place and unmodified. These slides may be posted as unanimated pdf versions on publicly-accessible course websites.. PowerPoint source (or pdf with animations) may **not** be posted to publicly-accessible websites, but may be posted for students on internal protected sites or distributed directly to students by other electronic means. Instructors may make printouts of the slides available to students for a reasonable photocopying charge, without incurring royalties. Any other use requires explicit permission. Instructors may obtain PowerPoint source or obtain special use permissions from Wiley – see <http://www.ddvahid.com> for information.

Introduction

- Let's learn to design digital circuits, starting with a simple form of circuit:

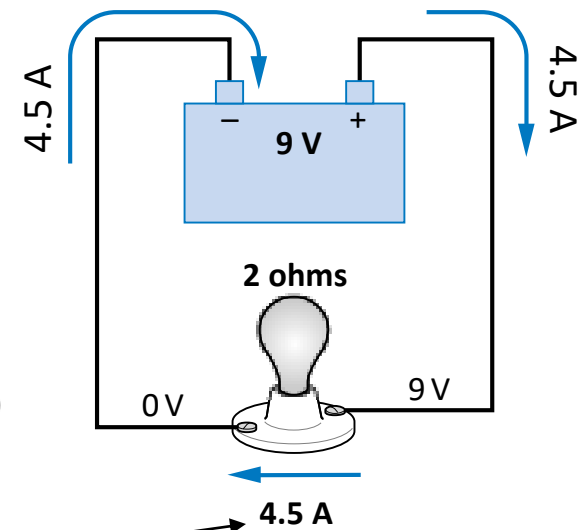
- **Combinational circuit**

- Outputs depend solely on the present combination of the circuit inputs' values
- Vs. sequential circuit: Has “memory” that impacts outputs too



Switches

- Electronic switches are the basis of binary digital circuits
 - Electrical terminology
 - Voltage:** Difference in electric potential between two points (volts, V)
 - Analogous to water pressure
 - Resistance:** Tendency of wire to resist current flow (ohms, Ω)
 - Analogous to water pipe diameter
 - Current:** Flow of charged particles (amps, A)
 - Analogous to water flow
 - $V = I * R$ (Ohm's Law)
 - $9\text{ V} = I * 2\text{ ohms}$
 - $I = 4.5\text{ A}$

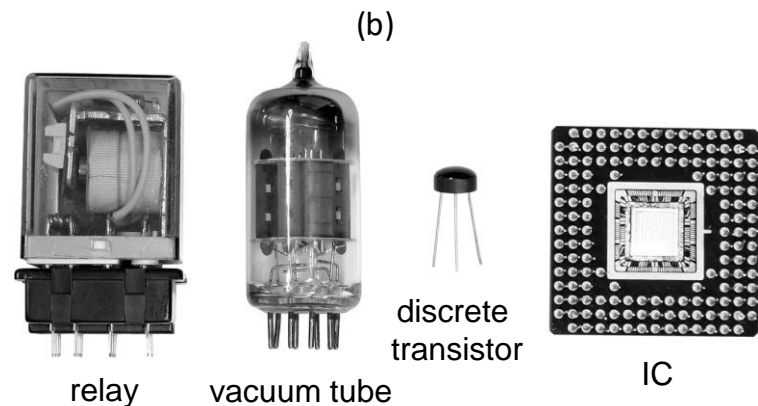
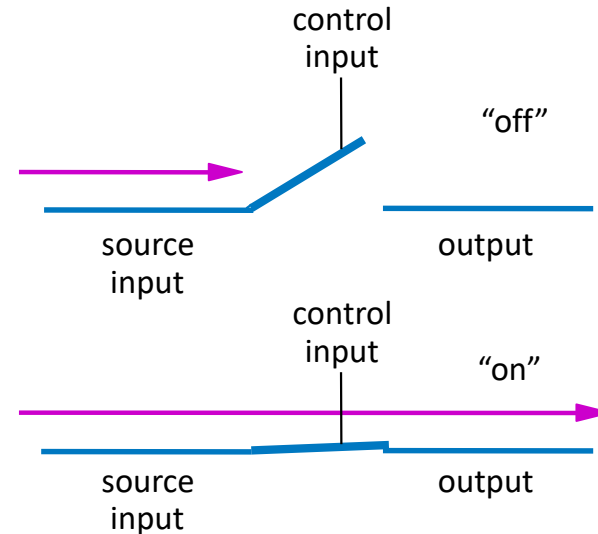


If a 9V potential difference is applied across a 2 ohm resistor, then 4.5 A of current will flow.



Switches

- A switch has three parts
 - Source input, and output
 - Current tries 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...

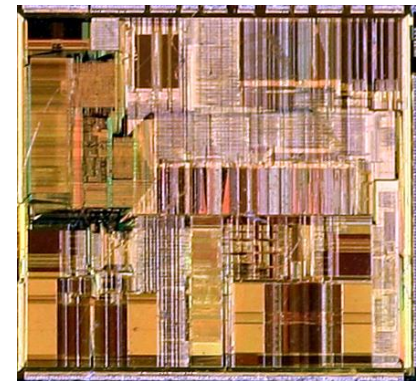


quarter
(to see the relative size)



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 area shrinks by half every 18 months
 - Consider how much shrinking occurs in just 10 years (try drawing it)
 - 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

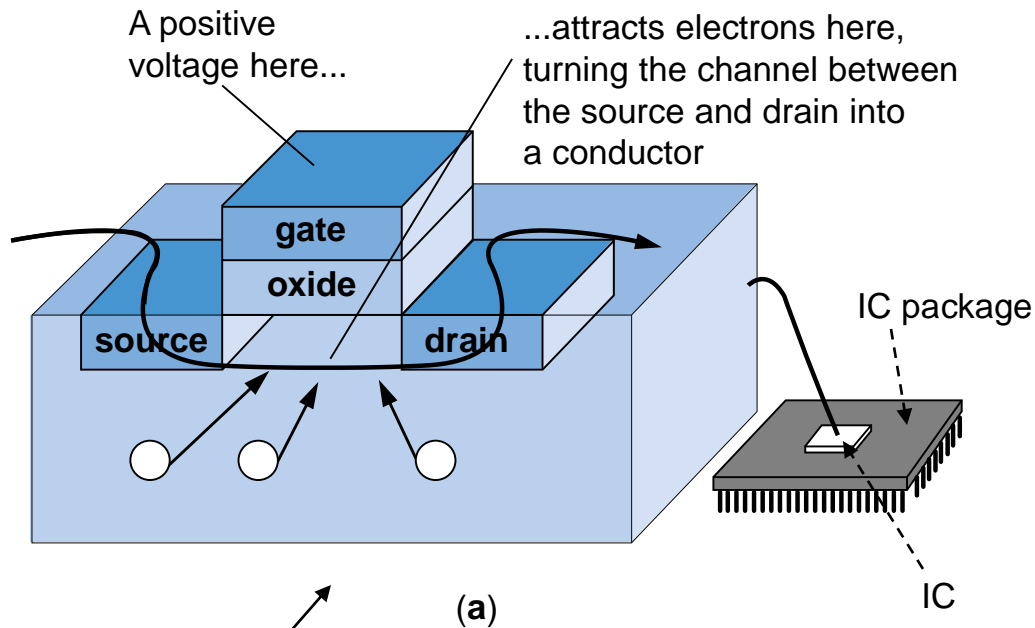


An Intel Pentium processor IC having millions of transistors



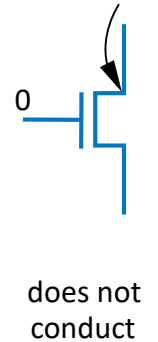
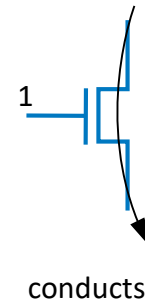
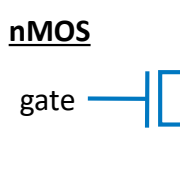
The CMOS Transistor

- CMOS transistor
 - Basic switch in modern ICs

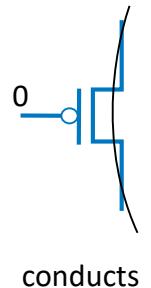
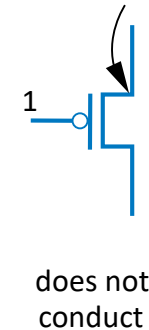
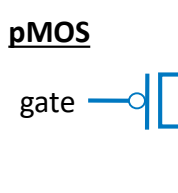


Silicon -- not quite a conductor or insulator:
Semiconductor

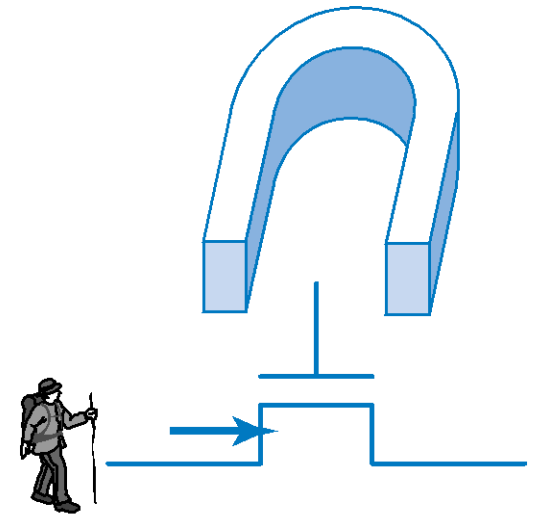
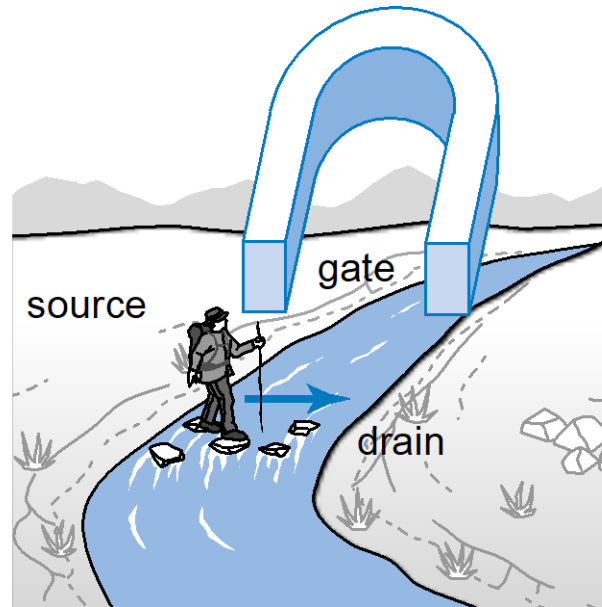
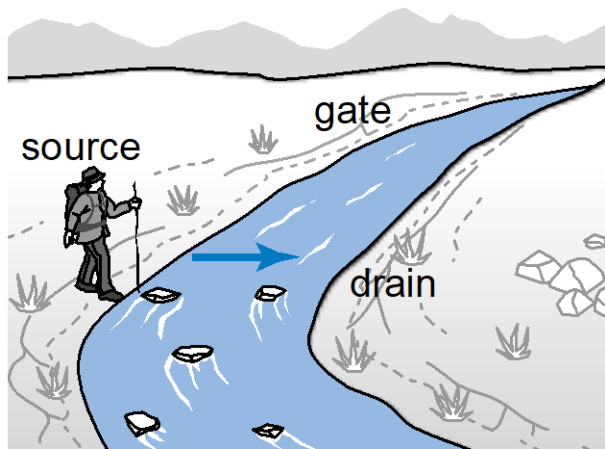
nMOS



pMOS



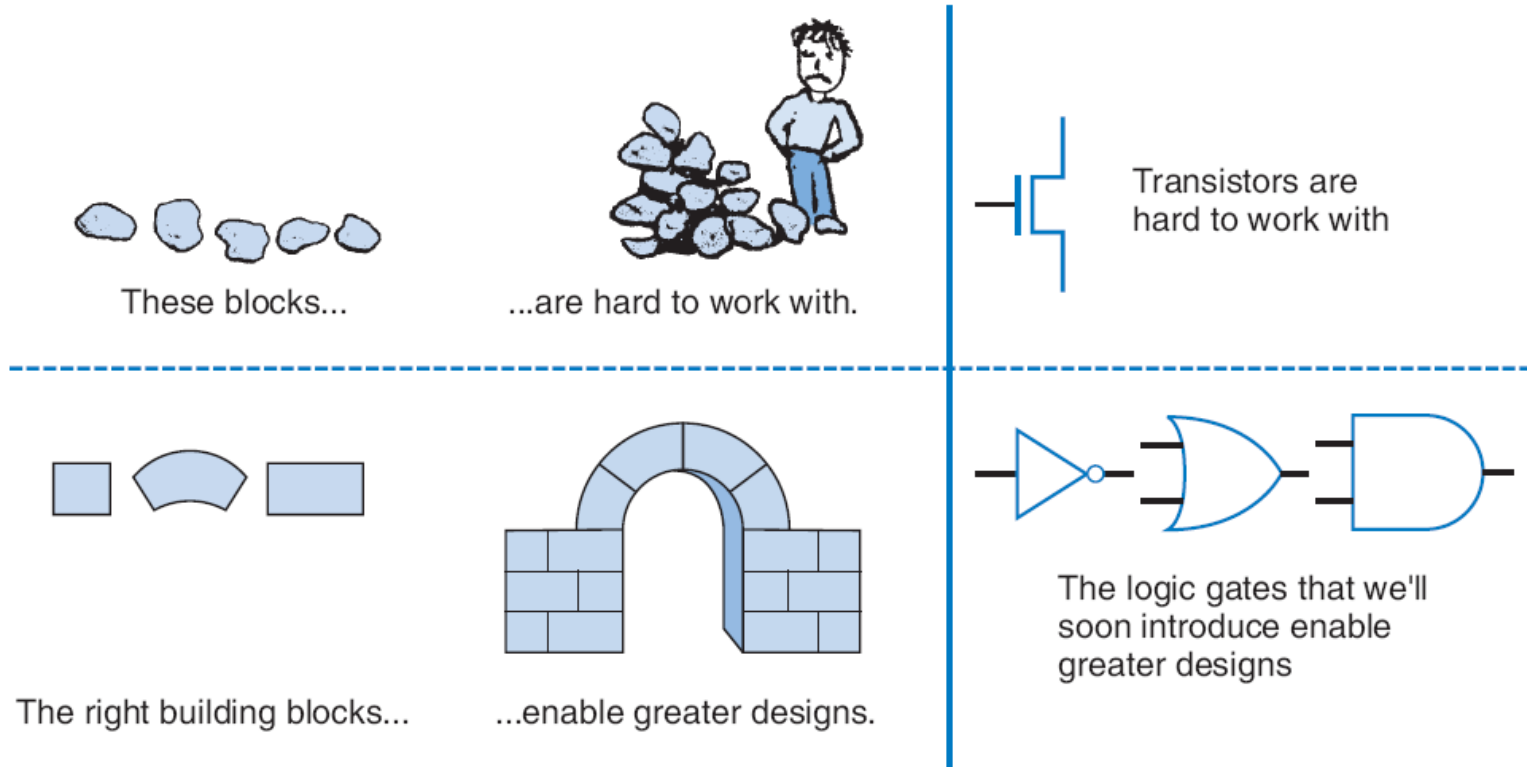
CMOS Transistor Analogy



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
 - Variables represent real numbers (x, y)
 - Operators operate on variables, return real numbers ($2.5x + y - 3$)
- **Boolean Algebra**
 - Variables represent 0 or 1 only
 - Operators return 0 or 1 only
 - Basic operators
 - AND: $a \text{ AND } b$ returns 1 only when both $a=1$ and $b=1$
 - OR: $a \text{ OR } b$ returns 1 if either (or both) $a=1$ or $b=1$
 - NOT: $\text{NOT } a$ 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

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 **$F = (m \text{ OR } j) \text{ AND NOT}(s)$**
 - Nice features
 - Formally evaluate
 - $m=1, j=0, s=1 \rightarrow F = (1 \text{ OR } 0) \text{ AND NOT}(1) = 1 \text{ AND } 0 = \underline{0}$
 - Formally transform
 - $F = (m \text{ and NOT}(s)) \text{ OR } (j \text{ and NOT}(s))$
 - Looks different, but same function
 - We'll show transformation techniques soon
 - Formally prove
 - Prove that if Sally goes to lunch ($s=1$), then I don't go ($F=0$)
 - $F = (m \text{ OR } j) \text{ AND NOT}(1) = (m \text{ OR } j) \text{ AND } 0 = 0$

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 **$F = (a \text{ AND } b) \text{ OR } (c \text{ AND } d)$** for the given values of variables a, b, c, and d:
 - Q1: $a=1, b=1, c=1, d=0$.
 - Answer: $F = (1 \text{ AND } 1) \text{ OR } (1 \text{ AND } 0) = 1 \text{ OR } 0 = 1$.
 - Q2: $a=0, b=1, c=0, d=1$.
 - Answer: $F = (0 \text{ AND } 1) \text{ OR } (0 \text{ AND } 1) = 0 \text{ OR } 0 = 0$.
 - Q3: $a=1, b=1, c=1, d=1$.
 - Answer: $F = (1 \text{ AND } 1) \text{ OR } (1 \text{ AND } 1) = 1 \text{ OR } 1 = 1$.

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: $F = a \text{ AND } b$
 - Q2. either of a or b is 1.
 - Answer: $F = a \text{ OR } b$
 - Q3. a is 1 and b is 0.
 - Answer: $F = a \text{ AND NOT}(b)$
 - Q4. a is not 0.
 - Answer:
 - (a) Option 1: $F = \text{NOT}(\text{NOT}(a))$
 - (b) Option 2: $F = a$



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: $F = h \text{ AND } e$.
- 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: $F = a \text{ AND } (s \text{ OR } d)$.
 - (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: $F = a \text{ AND } (s \text{ OR } \text{NOT}(d))$.



Relating Boolean Algebra to Digital Design

Boolean algebra
(mid-1800s)

Boole's intent: formalize human thought

Switches
(1930s)

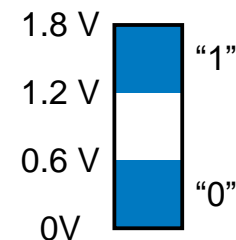
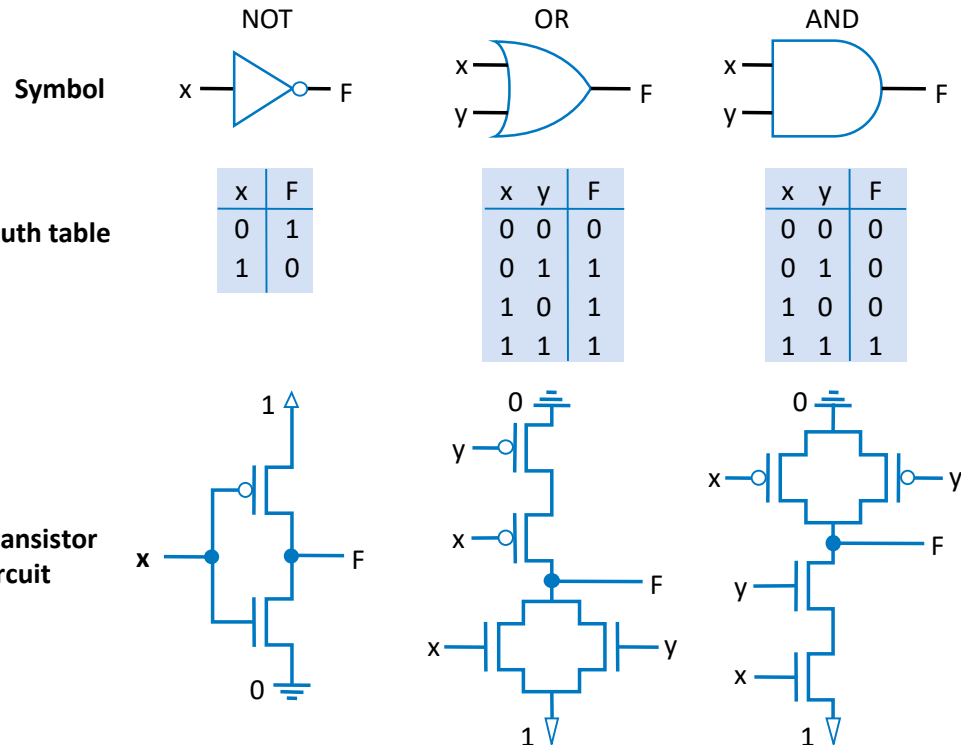
For telephone switching and other electronic uses

Shannon (1938)

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

Digital design

- Implement Boolean operators using transistors
 - Call those implementations **logic gates**.
 - Lets us build circuits by doing math** - powerful concept

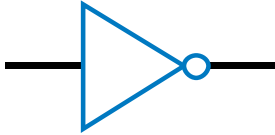


1 and 0 each actually corresponds to a voltage range

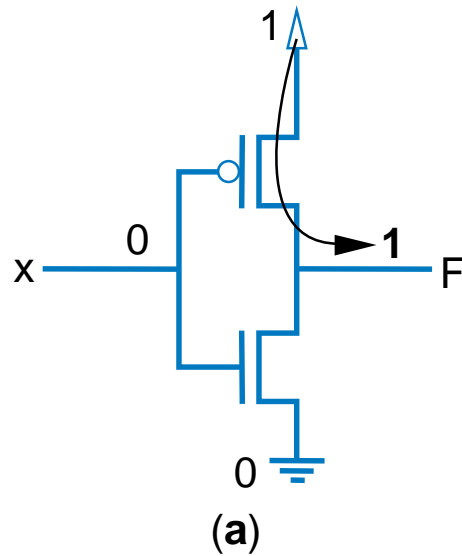
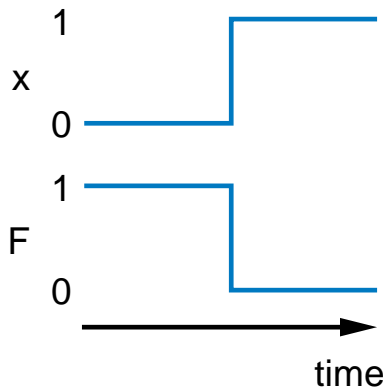
Next slides show how these circuits work.
Note: The above OR/AND implementations are inefficient; we'll show why, and show better ones, later.



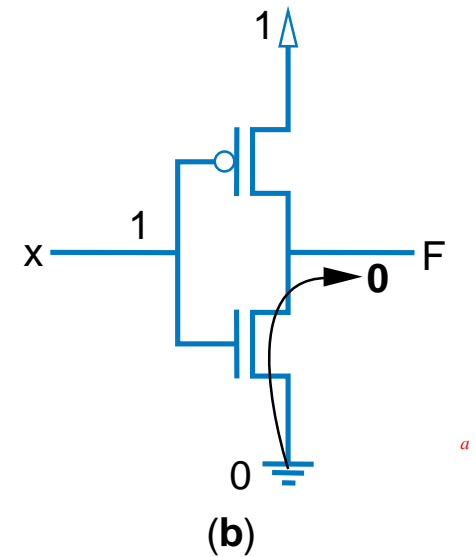
NOT gate



x	F
0	1
1	0



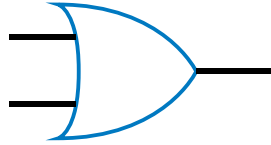
When the input is 0



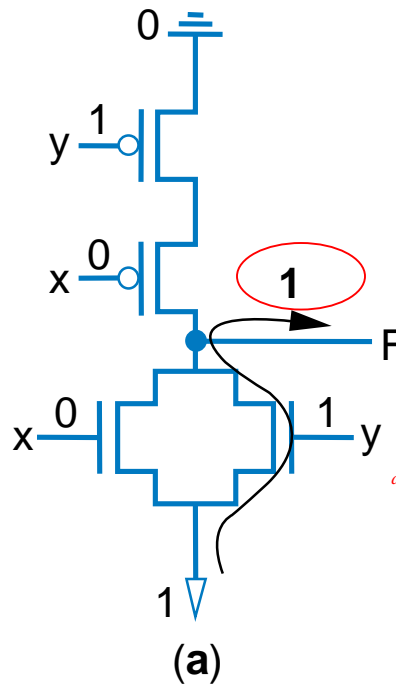
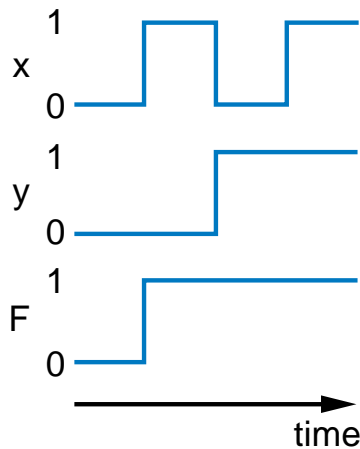
When the input is 1



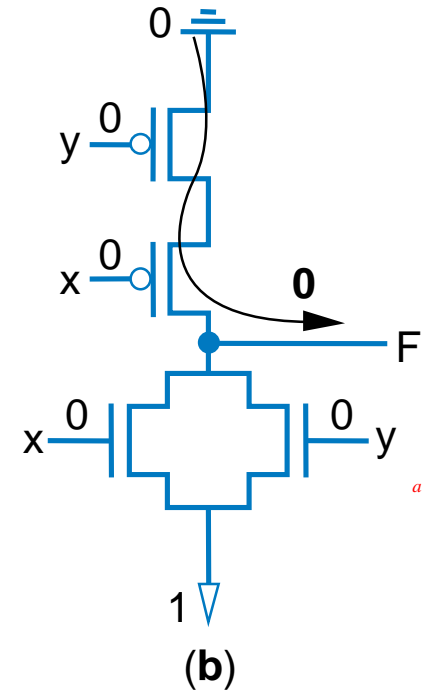
OR gate



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



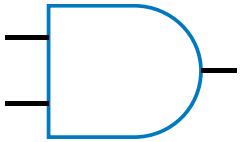
When an input is 1



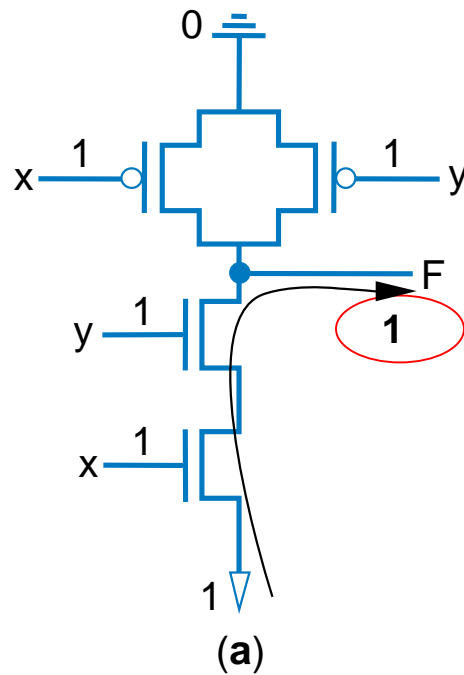
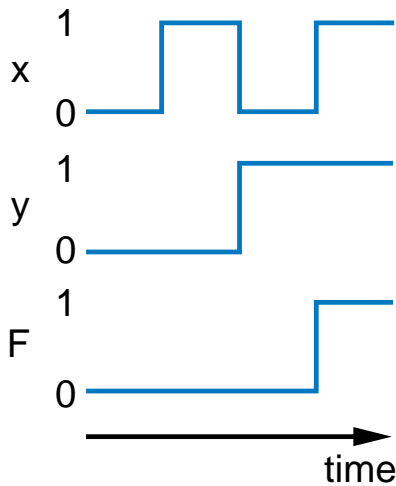
When both inputs are 0



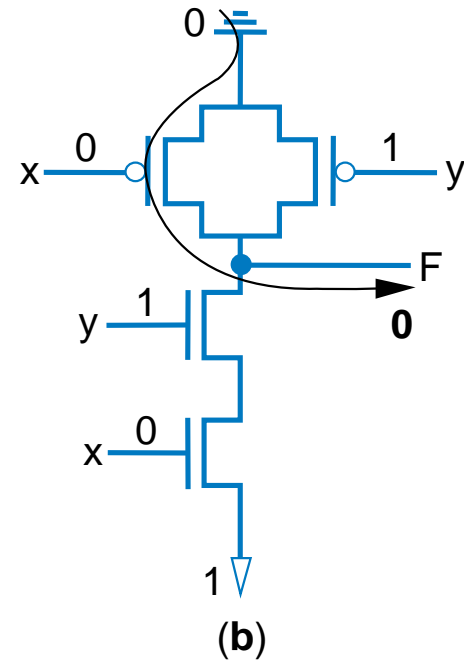
AND gate



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



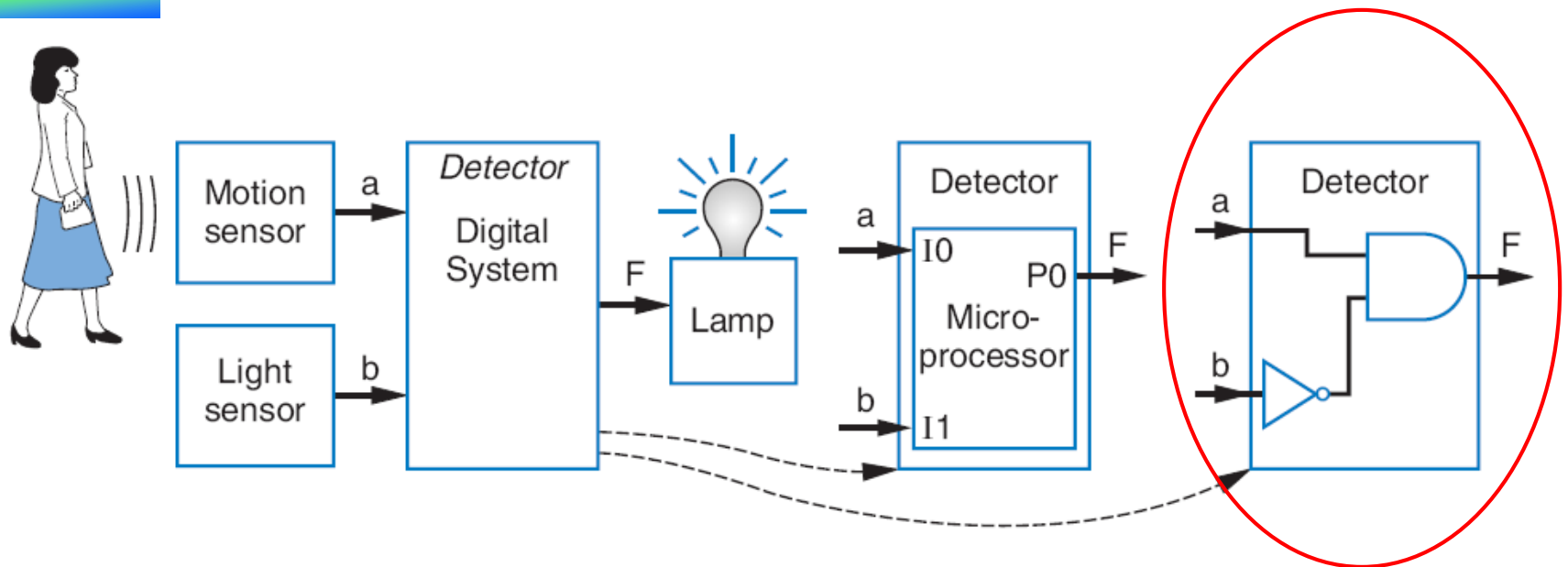
When both inputs are 1



When an input is 0



Building Circuits Using Gates



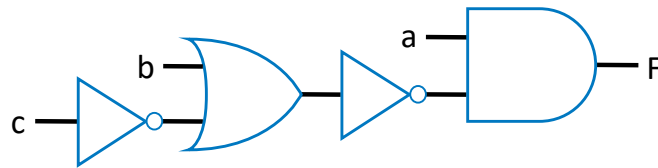
- Recall Chapter 1 motion-in-dark example
 - Turn on lamp ($F=1$) when motion sensed ($a=1$) and no light ($b=0$)
 - $F = a \text{ AND NOT}(b)$
 - Build using logic gates, AND and NOT, as shown
 - We just built our first digital circuit!



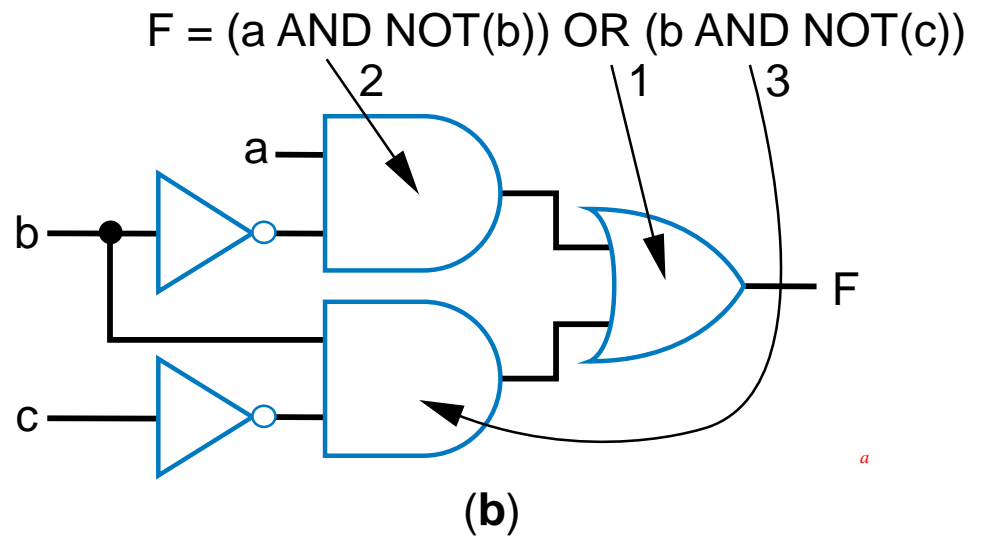
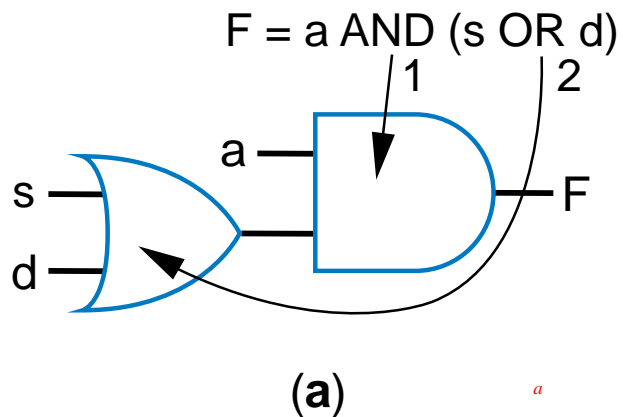
Example: Converting a Boolean Equation to a Circuit of Logic Gates

Start from the output, work back towards the inputs

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



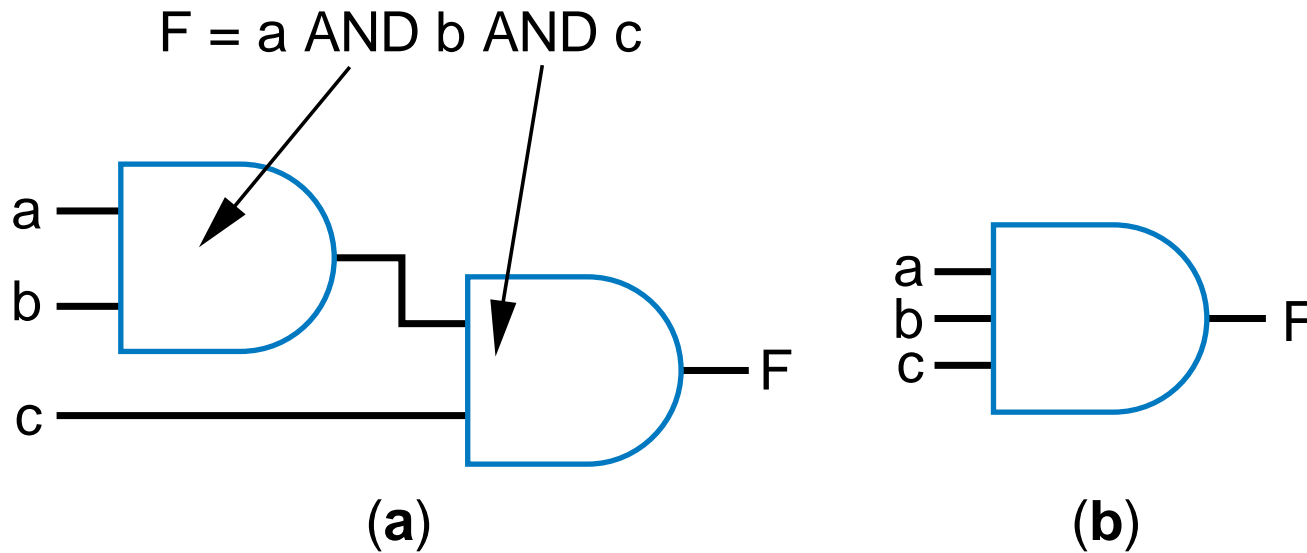
More examples



Start from the output, work back towards the inputs



Using gates with more than 2 inputs



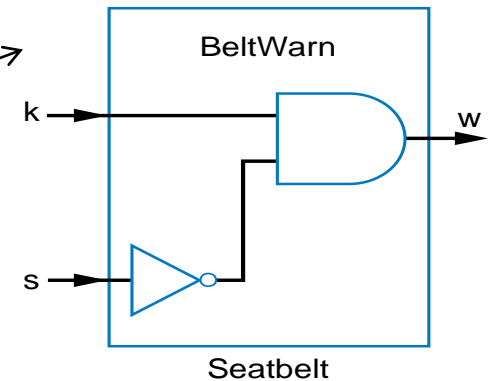
Can think of as $AND(a,b,c)$



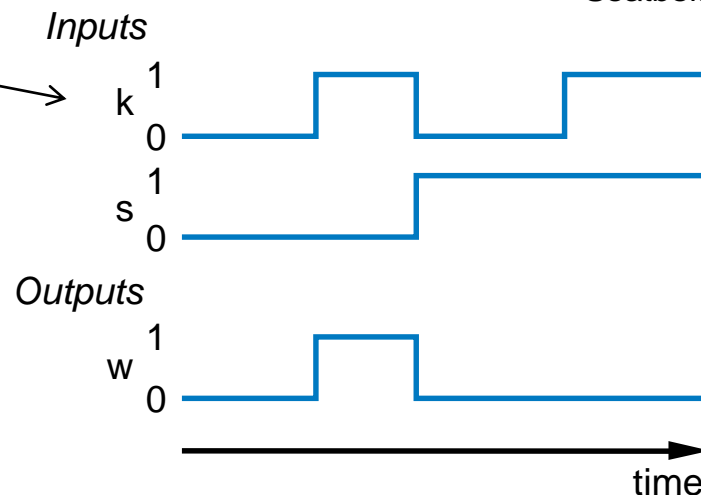
Example: Seat Belt Warning Light System

- Design circuit for warning light
- Sensors
 - $s=1$: seat belt fastened
 - $k=1$: key inserted
- Capture Boolean equation
 - seat belt not fastened, and key inserted
- Convert equation to circuit

$$w = \text{NOT}(s) \text{ AND } k$$



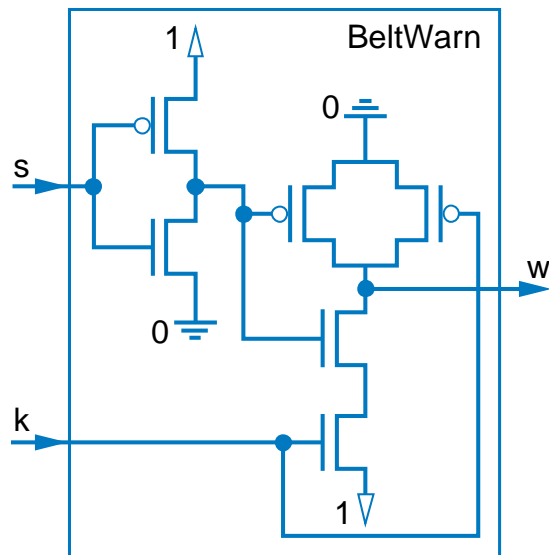
- *Timing diagram* illustrates circuit behavior
 - We set inputs to any values
 - Output set according to circuit



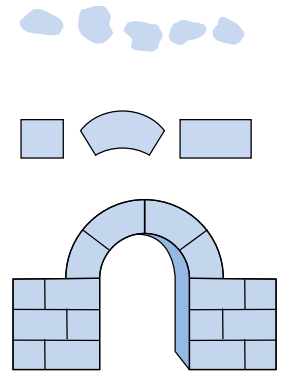
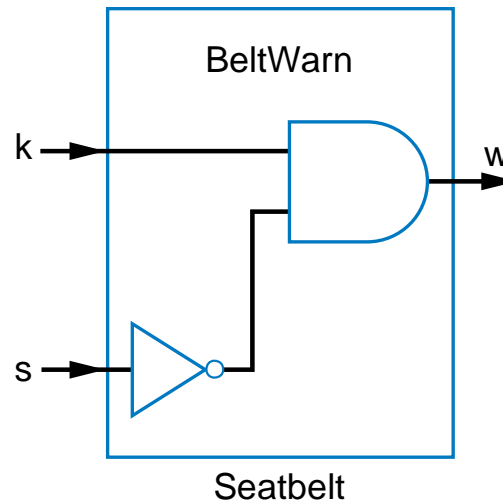
Gates vs. switches

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

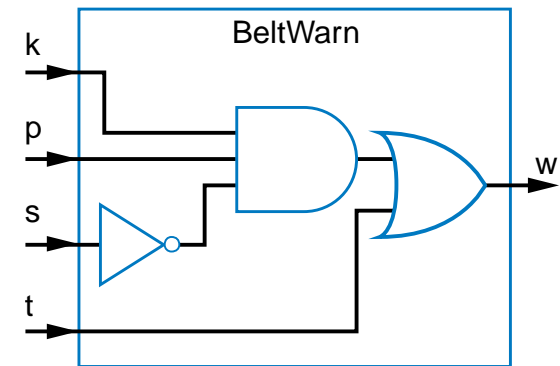
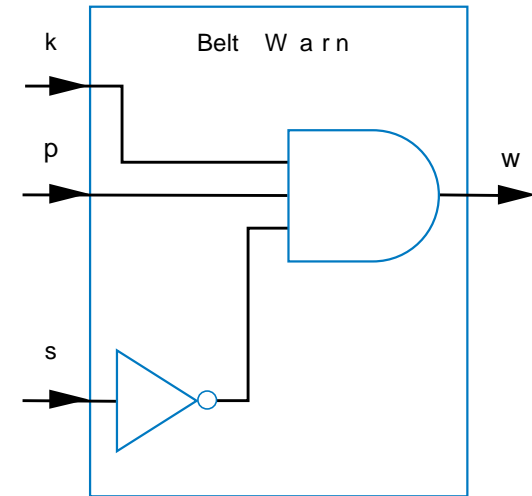


$$w = \text{NOT}(s) \text{ AND } k$$

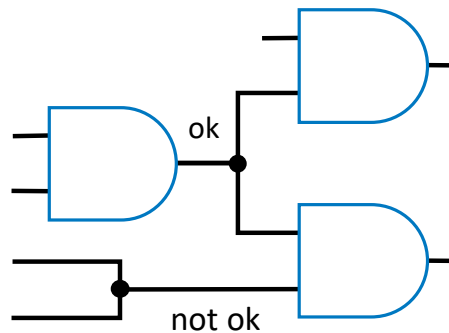
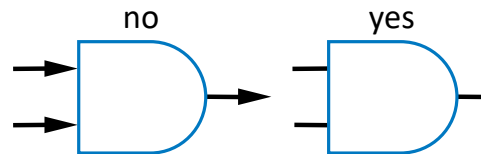
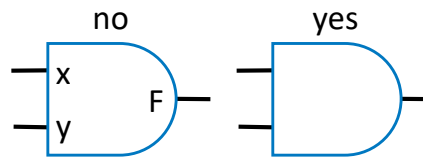


More examples: Seat belt warning light extensions

- Only illuminate warning light if person is in the seat ($p=1$), and seat belt not fastened and key inserted
- $w = p \text{ AND NOT}(s) \text{ AND } k$
- Given $t=1$ for 5 seconds after key inserted. Turn on warning light when $t=1$ (to check that warning lights are working)
- $w = (p \text{ AND NOT}(s) \text{ AND } k) \text{ OR } t$



Some Gate-Based Circuit Drawing Conventions



a

a



Boolean Algebra

- By defining logic gates based on Boolean algebra, we can *use algebraic methods to manipulate circuits*
- Notation: Writing a AND b , a OR b , NOT(a) is cumbersome
 - Use symbols: $a * b$ (or just ab), $a + b$, and a'
 - Original: $w = (p \text{ AND NOT}(s) \text{ AND } k) \text{ OR } t$
 - New: $w = ps'k + t$
 - Spoken as “ w equals p and s prime and k , or t ”
 - Or 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”
 - “product” and “sum” are OK and commonly used

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 Precedence

- 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 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 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
 - $a * (b + c) = a * b + a * c$
 - Can write as: $a(b+c) = ab + ac$
 - $a + (b * c) = (a + b) * (a + c)$
 - (This second one is tricky!)
 - Can write as: $a+(bc) = (a+b)(a+c)$
- 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

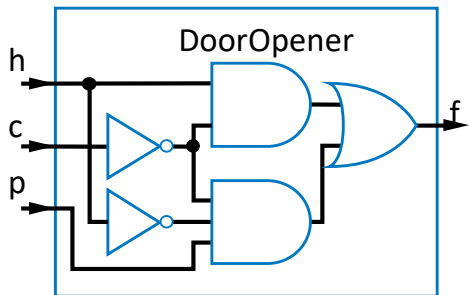
Example uses of the properties

- Show abc' equivalent to $c'ba$.
 - Use commutative property:
 - $a*b*c' = a*c'*b = c'*a*b = c'*b*a$
- 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: $f = hc' + h'pc'$



- Can the circuit be simplified?

$$f = hc' + h'pc'$$

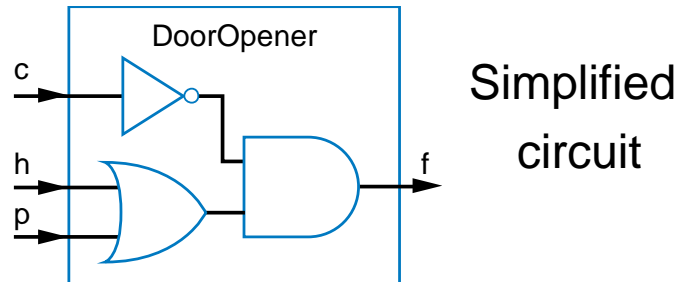
$$f = c'h + c'h'p \quad (\text{by the commutative property})$$

$$f = c'(h + h'p) \quad (\text{by the first distrib. property})$$

$$f = c'((h+h')*(h+p)) \quad (\text{2nd distrib. prop.; tricky one})$$

$$f = c'((1)*(h + p)) \quad (\text{by the complement property})$$

$$f = c'(h+p) \quad (\text{by the identity property})$$



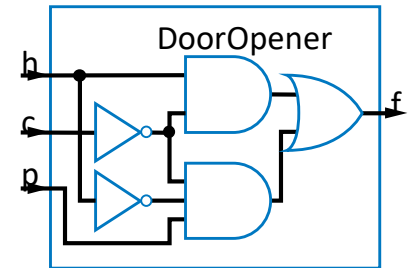
*Simplification of circuits is covered
in Sec. 2.11 / Sec 6.2.*



Example that Applies Boolean Algebra Properties



- Found inexpensive chip that computes:
 - $f = c'hp + c'hp' + c'h'p$
 - Can we use it for the door opener?
 - Is it the same as $f = hc' + h'pc'$?
- Apply Boolean algebra:



- Commutative
 - $a + b = b + a$
 - $a * b = b * a$
- Distributive
 - $a * (b + c) = a * b + a * c$
 - $a + (b * c) = (a + b) * (a + c)$
- 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
 - $a + a' = 1$
 - $a * a' = 0$

$$f = c'hp + c'hp' + c'h'p$$

$$f = c'h(p + p') + c'h'p \text{ (by the distributive property)}$$

$$f = c'h(1) + c'h'p \text{ (by the complement property)}$$

$$f = c'h + c'h'p \text{ (by the identity property)}$$

$$f = hc' + h'pc' \text{ (by the commutative property)}$$

Same! Yes, we can use it.



Boolean Algebra: Additional Properties

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



Example Applying DeMorgan's Law

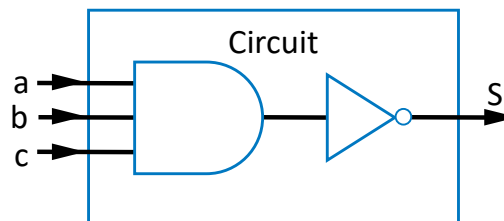
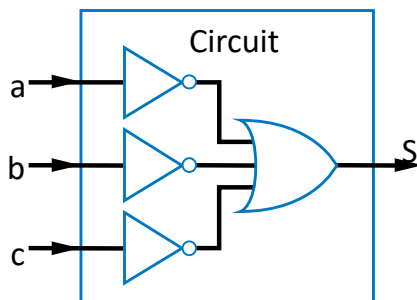
$$(a + b)' = a'b'$$
$$(ab)' = a' + b'$$

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 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



Example Applying Properties

- Commutative
 - $a + b = b + a$
 - $a * b = b * a$
- Distributive
 - $a * (b + c) = a * b + a * c$
 - $a + (b * c) = (a + b) * (a + c)$
- 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
 - $a + a' = 1$
 - $a * a' = 0$
- Null elements
 - $a + 1 = 1$
 - $a * 0 = 0$
- Idempotent Law
 - $a + a = a$
 - $a * a = a$
- Involution Law
 - $(a')' = a$
- DeMorgan's Law
 - $(a + b)' = a'b'$
 - $(ab)' = a' + b'$
- For door opener $f = c'(h+p)$, *prove* door stays closed ($f=0$) when $c=1$
 - $f = c'(h+p)$
 - *Let $c = 1$* (door forced closed)
 - $f = 1'(h+p)$
 - $f = 0(h+p)$
 - $f = 0h + 0p$ (by the distributive property)
 - $f = 0 + 0$ (by the null elements property)
 - $f = 0$

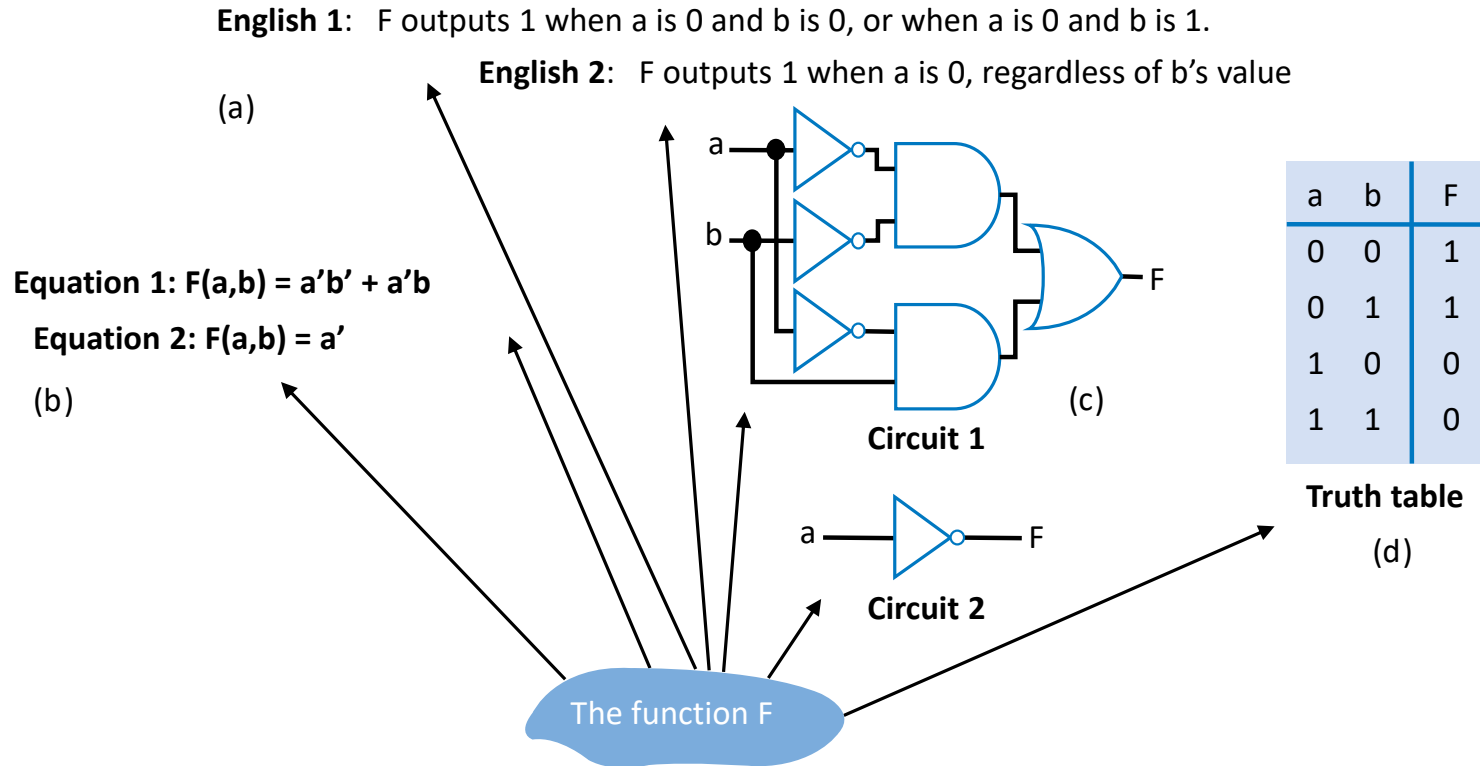


Complement of a Function

- Commonly want to find complement (inverse) of function F
 - 0 when F is 1; 1 when F is 0
- Use DeMorgan's Law repeatedly
 - Note: DeMorgan's Law defined for more than two variables, e.g.:
 - $(a + b + c)' = (abc)'$
 - $(abc)' = (a' + b' + c')$
- Complement of $f = w'xy + wx'y'z'$
 - $f' = (w'xy + wx'y'z')'$
 - $f' = (w'xy)'(wx'y'z')'$ (by DeMorgan's Law)
 - $f' = (w+x+y')(w'+x+y+z)$ (by DeMorgan's Law)
- Can then expand into sum-of-products form



Representations of Boolean Functions



- 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	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

a

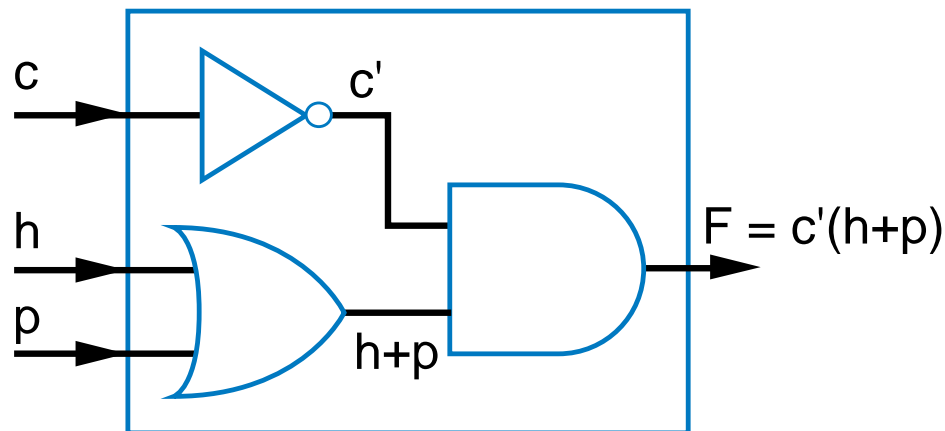
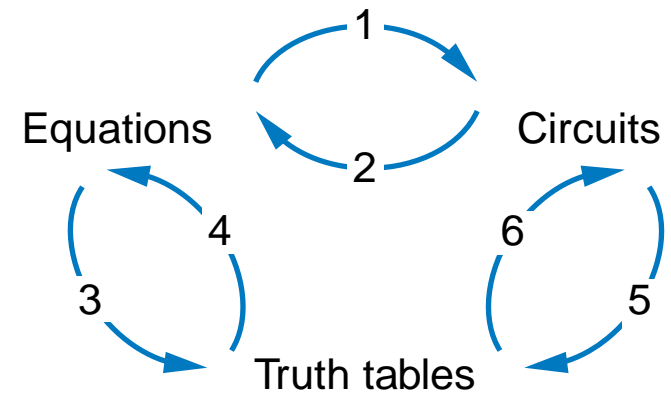
a	b	c	d	F
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

(c)

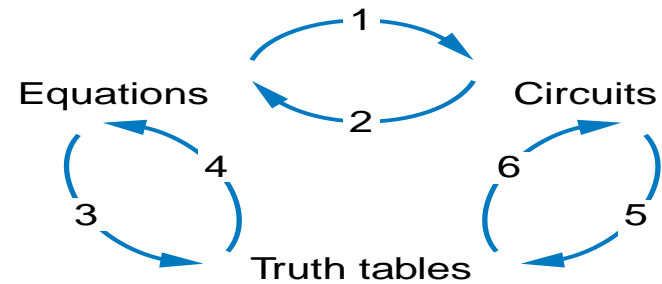


Converting among Representations

- Can convert from any representation to another
- Common conversions
 - Equation to circuit (we did this earlier)
 - Circuit to equation
 - Start at inputs, write expression of each gate output



Converting among Representations



- More common conversions
 - Truth table to equation (which we can then 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

$$F = ab'c + abc' + abc$$

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

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



Example: Converting from Truth Table to Equation

- Parity bit: Extra bit added to data, intended to enable detection of error (a bit changed unintentionally)
 - e.g., errors can occur on wires due to electrical interference
- Even parity: Set parity bit so total number of 1s (data + parity) is even
 - e.g., if data is 001, parity bit is 1
→ 0011 has even number of 1s
- Want equation, but easiest to start from truth table for this example

a	b	c	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

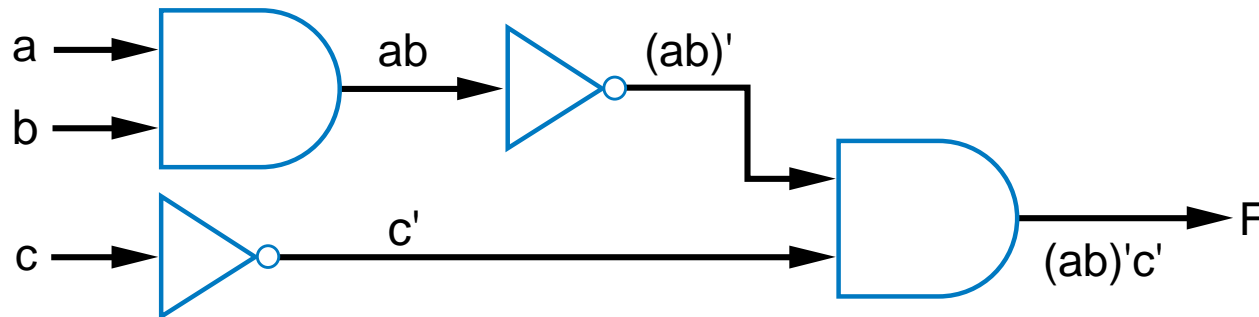
Convert to eqn.

$$P = a'b'c + a'bc' + ab'c' + abc$$



Example: Converting from Circuit to Truth Table

- First convert to circuit to equation, then equation to table



Inputs						Outputs
a	b	c	ab	$(ab)'$	c'	F
0	0	0	0	1	1	1
0	0	1	0	1	0	0
0	1	0	0	1	1	1
0	1	1	0	1	0	0
1	0	0	0	1	1	1
1	0	1	0	1	0	0
1	1	0	1	0	1	0
1	1	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

$$f = c'h p + c'h p' + c'h'$$

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

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

$$f = c'h + c'h'p$$

(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'			F = a'b' + a'b + ab		
a	b	F	a	b	F
0	0	1	0	0	1
0	1	1	0	1	1
1	0	0	1	0	0
1	1	1	1	1	1

Same



Truth Table Canonical Form

- Q: Determine via truth tables whether $ab + a'$ and $(a+b)'$ are equivalent

$F = ab + a'$			$F = (a+b)'$		
a	b	F	a	b	F
0	0	1	0	0	1
0	1	1	0	1	0
1	0	0	1	0	0
1	1	1	1	1	0

Not equivalent

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
 - $ax^2 + bx + c$ ($3x^2 + 4x + 2x^2 + 3 + 1 \rightarrow 5x^2 + 4x + 4$)
 - 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 equivalent to $F(a,b)=a'b'+a'b+ab$, by converting first equation to canonical form (second already is)

$F = ab+a'$ (already sum of products)

$F = ab + a'(b+b')$ (expanding term)

$F = ab + a'b + a'b'$ (Equivalent – same three terms as other equation)



Canonical Form – Sum of Minterms

- Q: Determine whether the functions $G(a,b,c,d,e) = abcd + a'bcde$ and $H(a,b,c,d,e) = abcde + abcde' + a'bcde + a'bcde(a' + c)$ are equivalent.

$$G = abcd + a'bcde$$

$$G = abcd(e+e') + a'bcde$$

$$G = abcde + abcde' + a'bcde$$

$$G = a'bcde + abcde' + abcde \quad (\text{sum of minterms form})$$

Equivalent

$$H = abcde + abcde' + a'bcde + a'bcde(a' + c)$$

$$H = abcde + abcde' + a'bcde + a'bcdea' + a'bcdec$$

$$H = abcde + abcde' + a'bcde + a'bcde + a'bcde$$

$$H = abcde + abcde' + a'bcde$$

$$H = a'bcde + abcde' + abcde$$



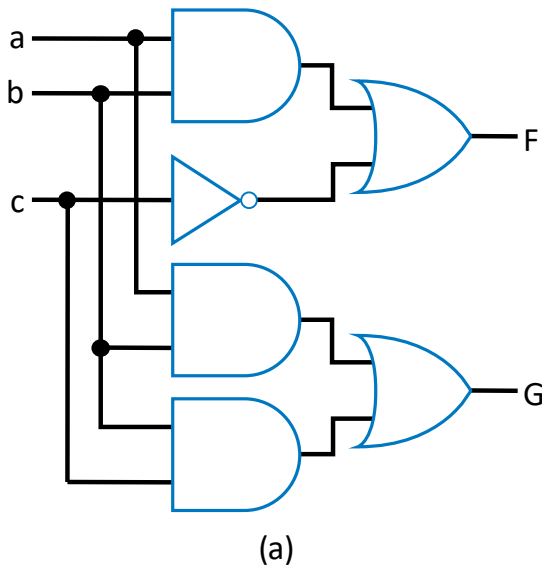
Compact Sum of Minterms Representation

- List each minterm as a number
- Number determined from the binary representation of its variables' values
 - $a'b c d e$ corresponds to 01111, or 15
 - $a b c d e'$ corresponds to 11110, or 30
 - $a b c d e$ corresponds to 11111, or 31
- Thus, $H = a'b c d e + a b c d e' + a b c d e$ can be written as:
 - $H = \sum m(15, 30, 31)$
 - "H is the sum of minterms 15, 30, and 31"

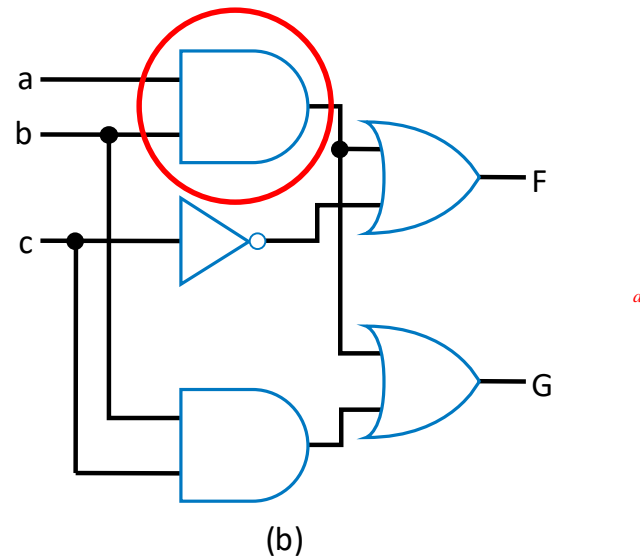


Multiple-Output Circuits

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



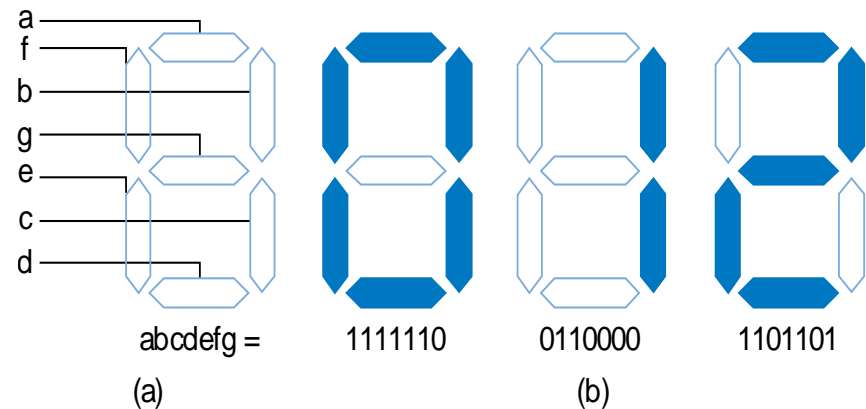
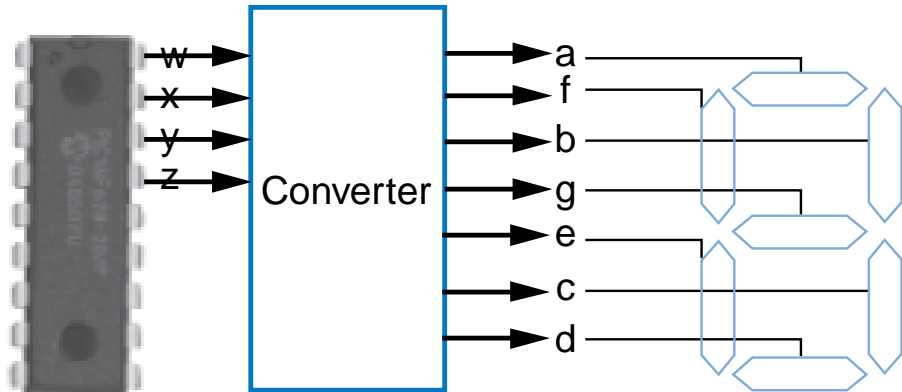
Option 1: Separate circuits



Option 2: Shared gates



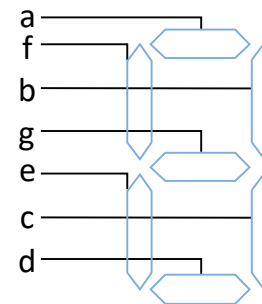
Multiple-Output Example: BCD to 7-Segment Converter



Multiple-Output Example: BCD 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



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

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

...



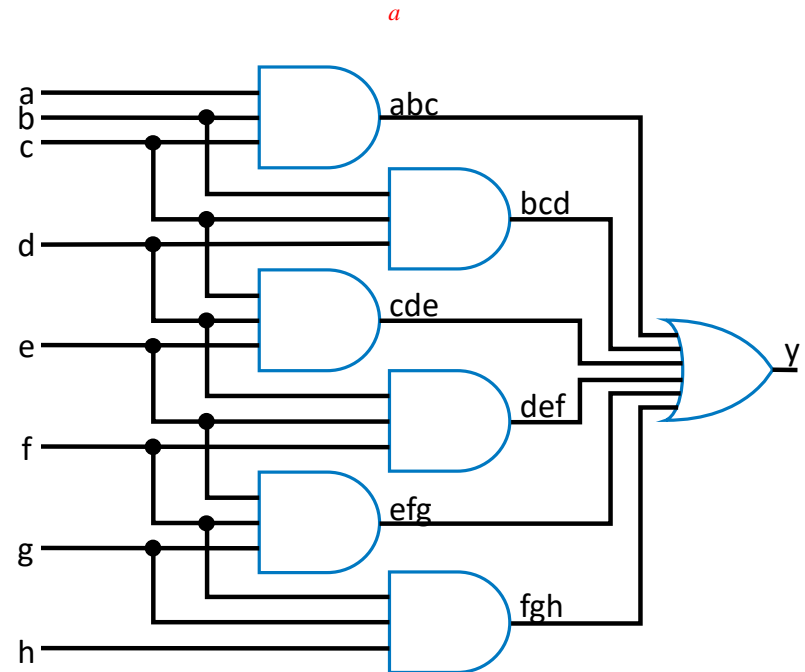
Combinational Logic Design Process

Step	Description
Step 1: Capture behavior Capture the function	Create a truth table or equations, <i>whichever is most natural for the given problem</i> , to describe the desired behavior of each output of the combinational logic.
Step 2: Convert to circuit 2A: Create equations 2B: Implement as a gate-based circuit	<p>This substep 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.</p> <p>For each output, create a circuit corresponding to the output's equation. (Sharing gates among multiple outputs is OK optionally.)</p>



Example: Three 1s Pattern Detector

- Problem: Detect three consecutive 1s in 8-bit input: abcdefgh
 - 00011101 → 1
 - 10101011 → 0
 - 11110000 → 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 2a: Create** equation -- already done
- **Step 2b: Implement** as a gate-based circuit



Example: Number of 1s Counter

- Problem: Output in binary on two outputs yz the # of 1s on three inputs

- $010 \rightarrow 01$
- $101 \rightarrow 10$
- $000 \rightarrow 00$

– **Step 1: Capture** the function

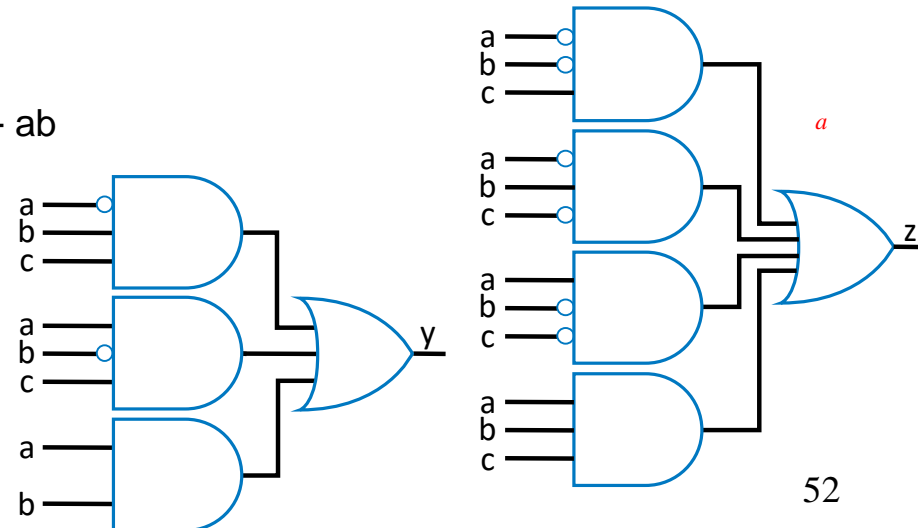
- Truth table or equation?
 - Truth table is straightforward

– **Step 2a: Create** equations

- $y = a'bc + ab'c + abc' + abc$
- $z = a'b'c + a'bc' + ab'c' + abc$
- Optional: Let's simplify y:
 - $y = a'bc + ab'c + ab(c' + c) = a'bc + ab'c + ab$

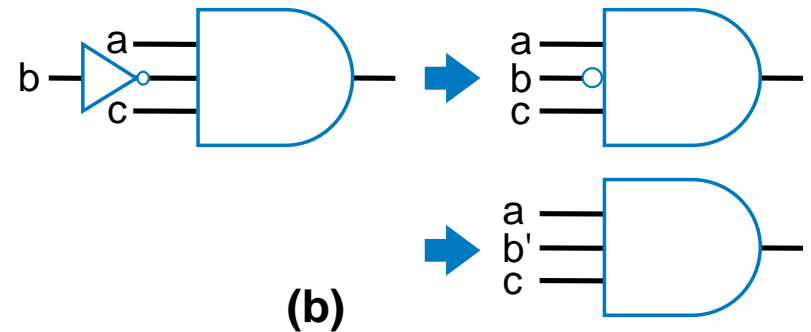
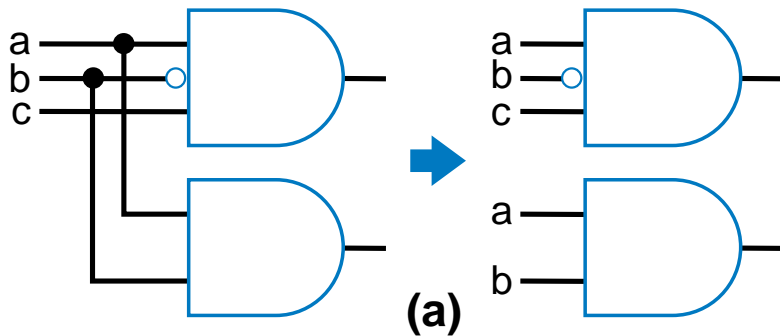
– **Step 2b: Implement** as a gate-based circuit

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



Simplifying Notations

- Used in previous circuit



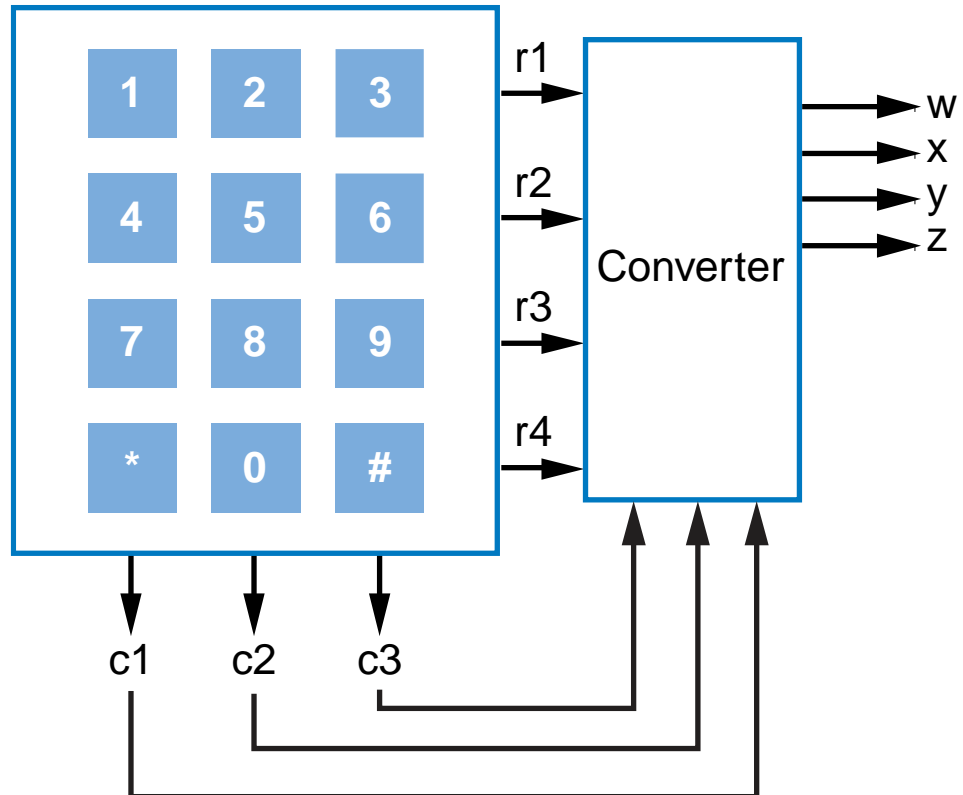
List inputs multiple times
→ Less wiring in drawing

Draw inversion bubble
rather than inverter. Or list
input as complemented.



Example: Keypad Converter

- Keypad has 7 outputs
 - One per row
 - One per column
- Key press sets one row and one column output to 1
 - Press "5" \rightarrow $r2=1$, $c2=1$
- Goal: Convert keypad outputs into 4-bit binary number
 - 0-9 \rightarrow 0000 to 1001
 - * \rightarrow 1010, # \rightarrow 1011
 - nothing pressed: 1111



Example: Keypad Converter

- Step 1: Capture behavior
 - Truth table too big (2^7 rows); equations not clear either
 - Informal table can help

TABLE 2.7 Informal table for the 12-button keypad to 4-bit code converter.

Button	Signals	4-bit code outputs			
		w	x	y	z
1	r1 c1	0	0	0	1
2	r1 c2	0	0	1	0
3	r1 c3	0	0	1	1
4	r2 c1	0	1	0	0
5	r2 c2	0	1	0	1
6	r2 c3	0	1	1	0
7	r3 c1	0	1	1	1

Button	Signals	4-bit code outputs			
		w	x	y	z
8	r3 c2	1	0	0	0
9	r3 c3	1	0	0	1
*	r4 c1	1	0	1	0
0	r4 c2	0	0	0	0
#	r4 c3	1	0	1	1
(none)		1	1	1	1

Step 2b: Implement^a
as circuit (note
sharable gates) ...

$$w = r3c2 + r3c3 + r4c1 + r4c3 + r1'r2'r3'r4'c1'c2'c3'$$

$$x = r2c1 + r2c2 + r2c3 + r3c1 + r1'r2'r3'r4'c1'c2'c3'$$

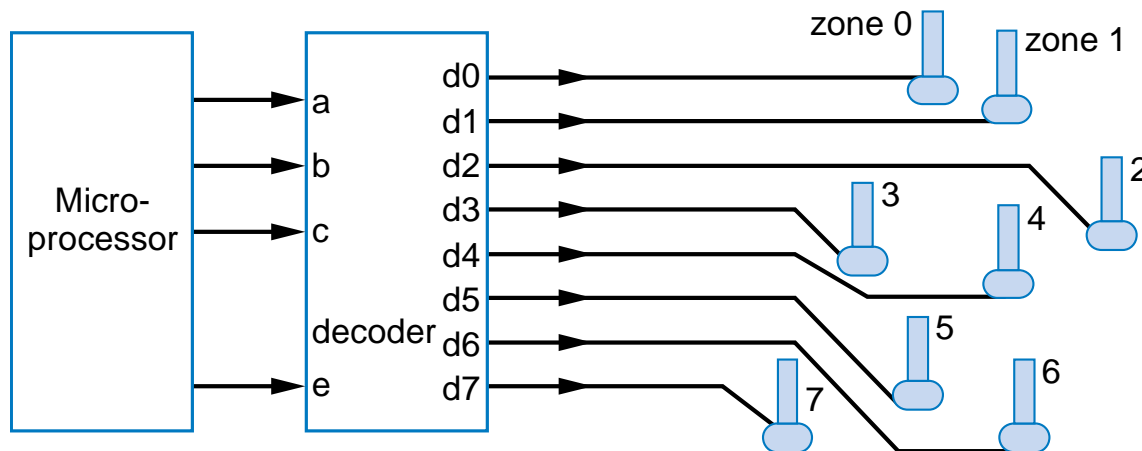
$$y = r1c2 + r1c3 + r2c3 + r3c1 + r4c1 + r4c3 + r1'r2'r3'r4'c1'c2'c3'$$

$$z = r1c1 + r1c3 + r2c2 + r3c1 + r3c3 + r4c3 + r1'r2'r3'r4'c1'c2'c3'$$



Example: Sprinkler Controller

- Microprocessor outputs which zone to water (e.g., cba=110 means zone 6) and enables watering (e=1)
- Decoder should set appropriate valve to 1



Step 1: Capture behavior

$$d0 = a'b'c'e$$

$$d1 = a'b'ce$$

$$d2 = a'bc'e$$

$$d3 = a'bce$$

$$d4 = ab'c'e$$

$$d5 = ab'ce$$

$$d6 = abc'e$$

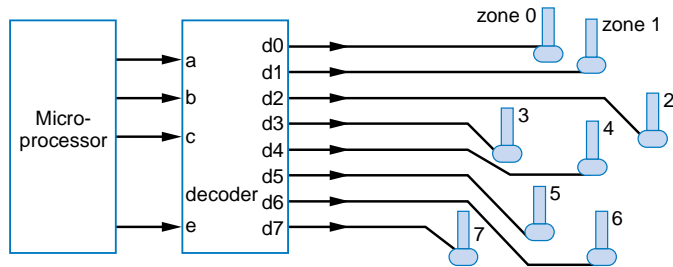
$$d7 = abce$$

*Equations seem like
a natural fit*



Example: Sprinkler Controller

- Step 2b: Implement as circuit



$$d0 = a'b'c'e$$

$$d1 = a'b'ce$$

$$d2 = a'bc'e$$

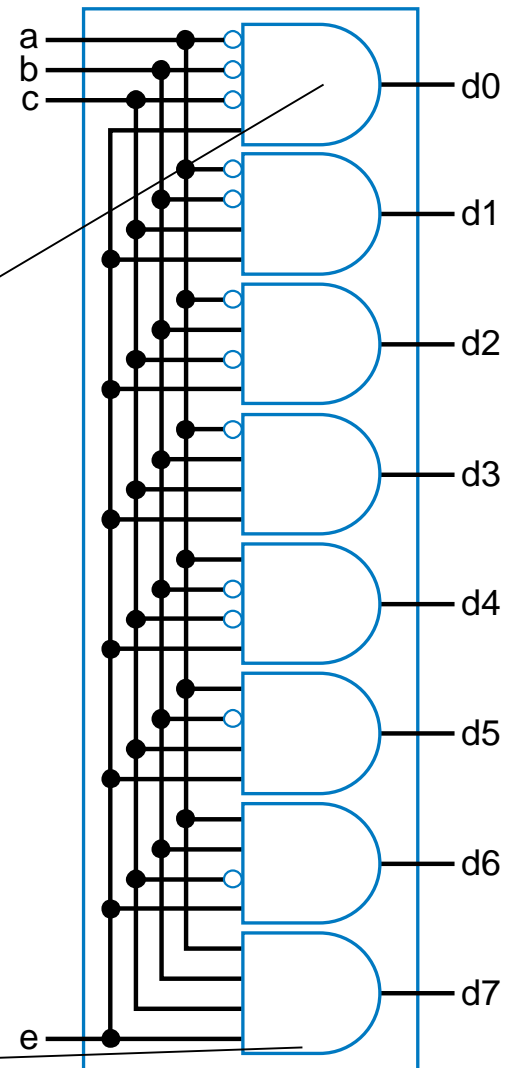
$$d3 = a'bce$$

$$d4 = ab'c'e$$

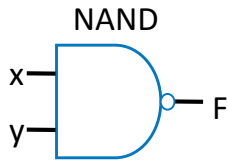
$$d5 = ab'ce$$

$$d6 = abc'e$$

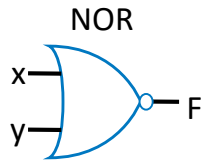
$$d7 = abce$$



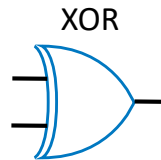
More Gates



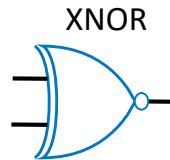
x	y	F
0	0	1
0	1	1
1	0	1
1	1	0



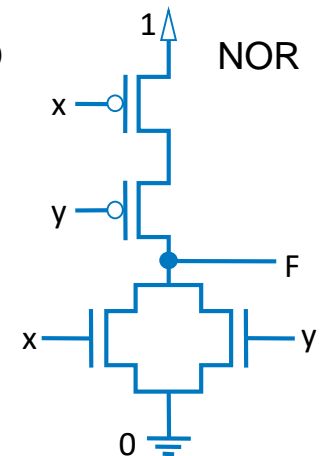
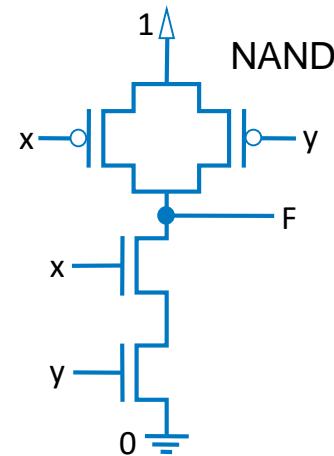
x	y	F
0	0	1
0	1	0
1	0	0
1	1	0



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



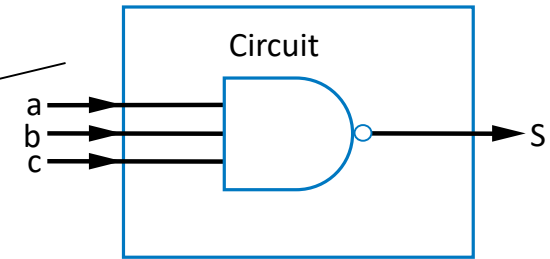
- NAND: Opposite of AND (“NOT AND”)
- NOR: Opposite of OR (“NOT OR”)
- XOR: Exactly 1 input is 1, for 2-input XOR. (For more inputs -- odd number of 1s)
- XNOR: Opposite of XOR (“NOT XOR”)
- NAND same as AND with power & ground switched
 - nMOS conducts 0s well, but not 1s (reasons beyond our scope) – so NAND is more efficient
- Likewise, NOR same as OR with power/ground switched
- NAND/NOR more common
- AND in CMOS: NAND with NOT
- OR in CMOS: NOR with NOT



More Gates: Example Uses

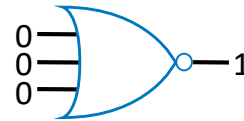
- Aircraft lavatory sign example

- $S = (abc)'$



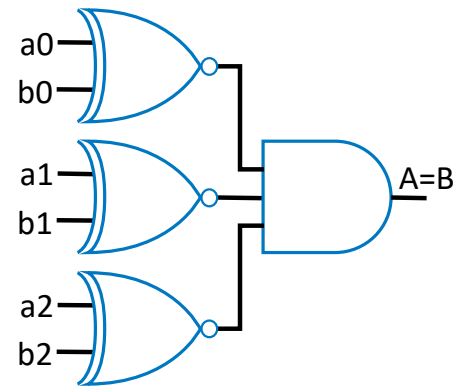
- Detecting all 0s

- Use NOR



- Detecting equality

- Use XNOR



- Detecting odd # of 1s

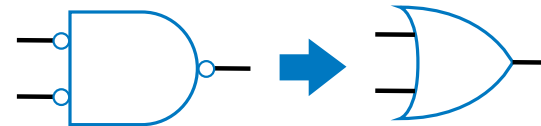
- Use XOR

- 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
 - Thus, NAND is a universal gate
 - Can implement any circuit using just NAND gates
- 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

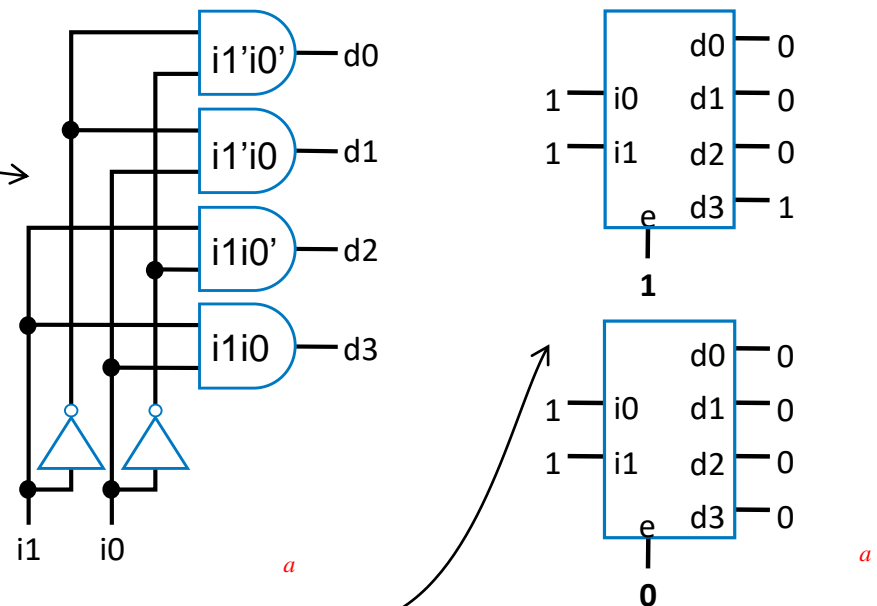
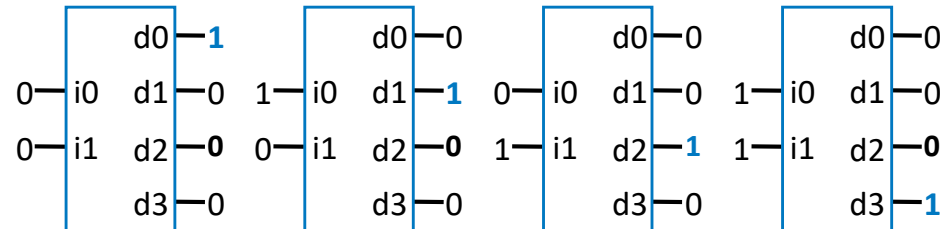
$2^4 = 16$
possible functions

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



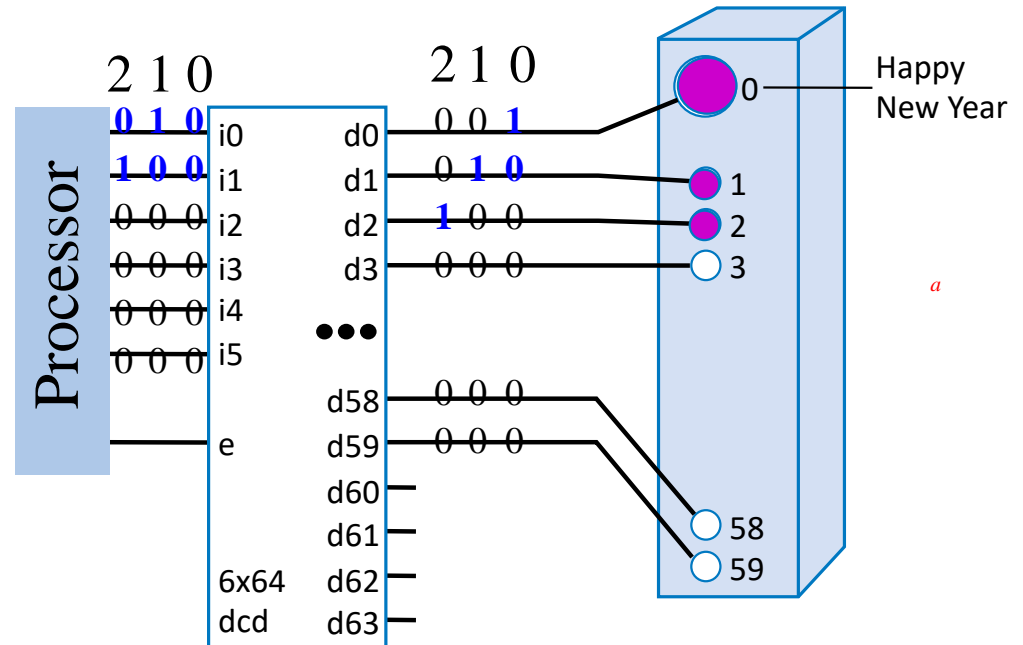
Decoders and Muxes

- **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



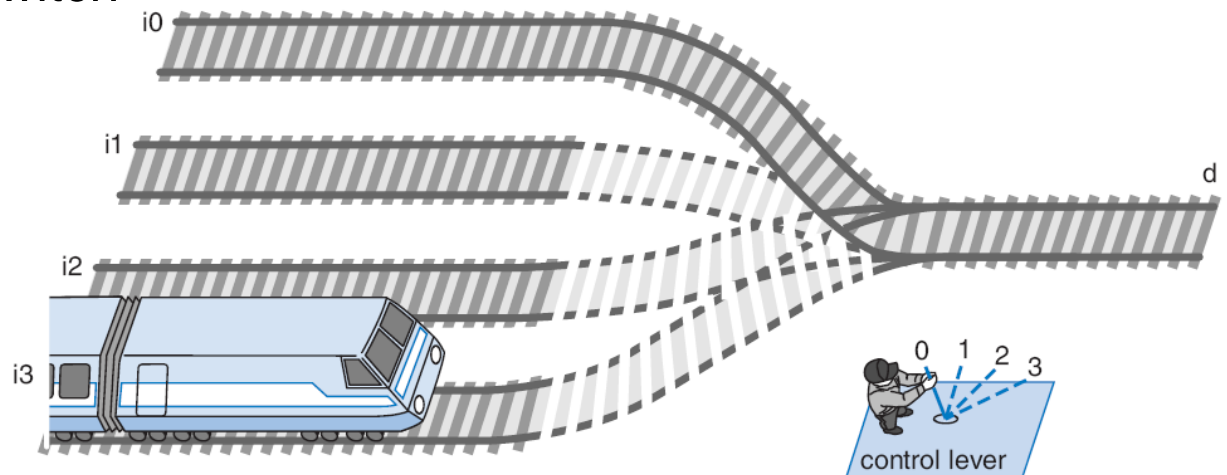
Decoder Example

- 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

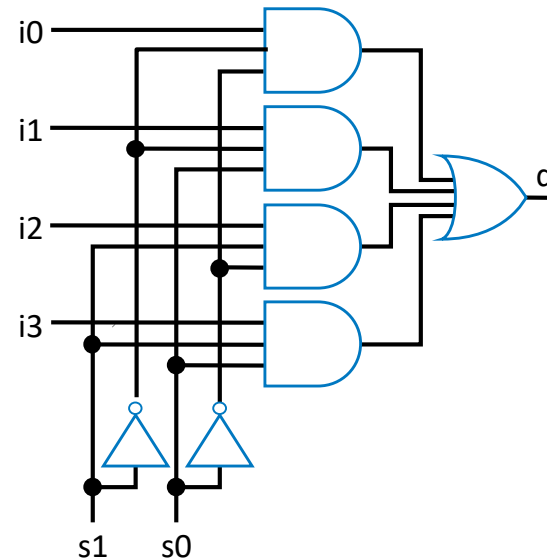
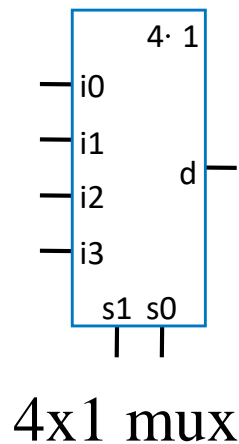
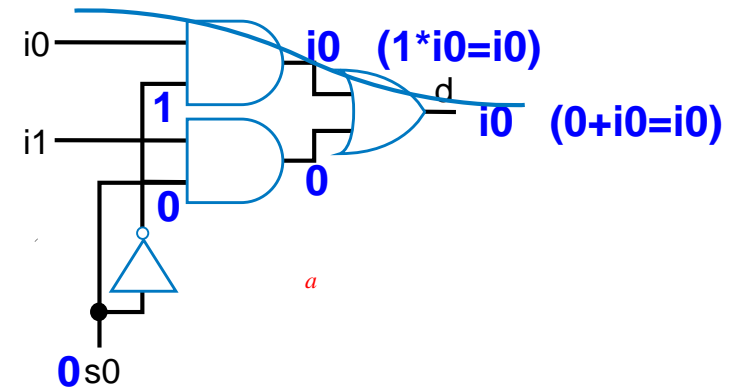
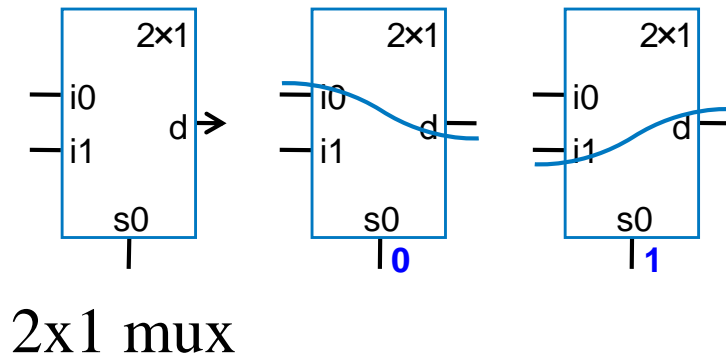


Multiplexor (Mux)

- 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 \rightarrow needs 2 select inputs to indicate which input to route through
 - 8 input mux \rightarrow 3 select inputs
 - N inputs $\rightarrow \log_2(N)$ selects
 - Like a rail yard switch

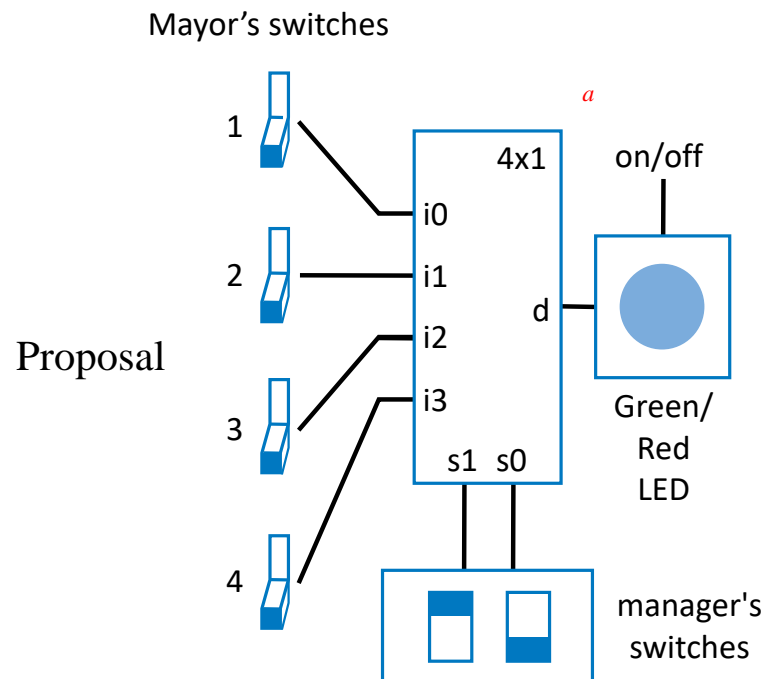


Mux Internal Design

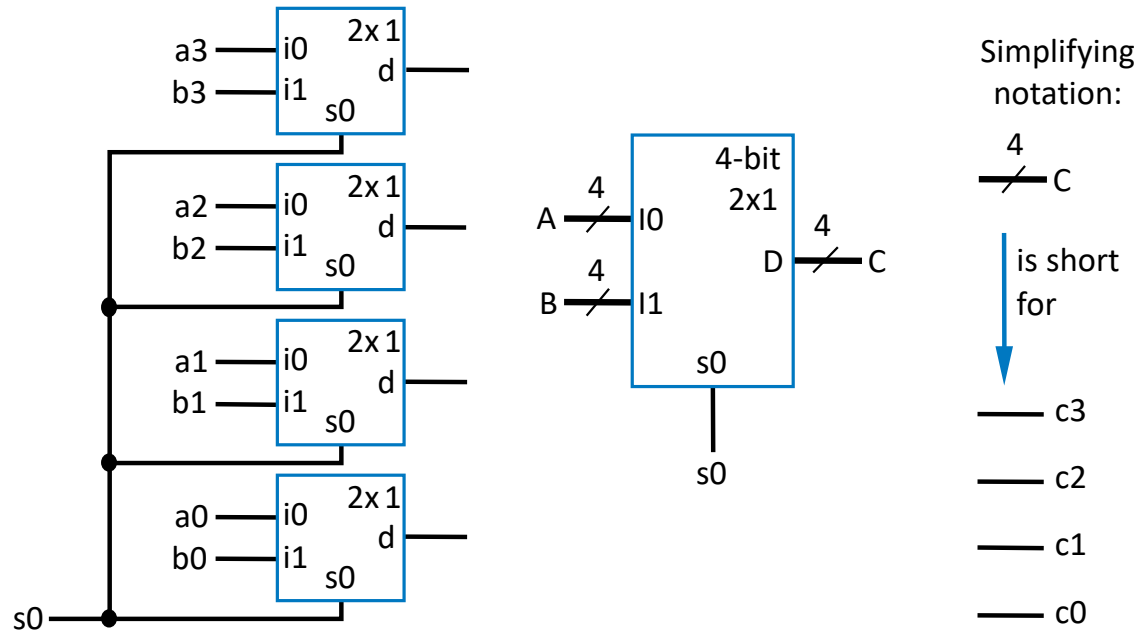


Mux Example

- 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



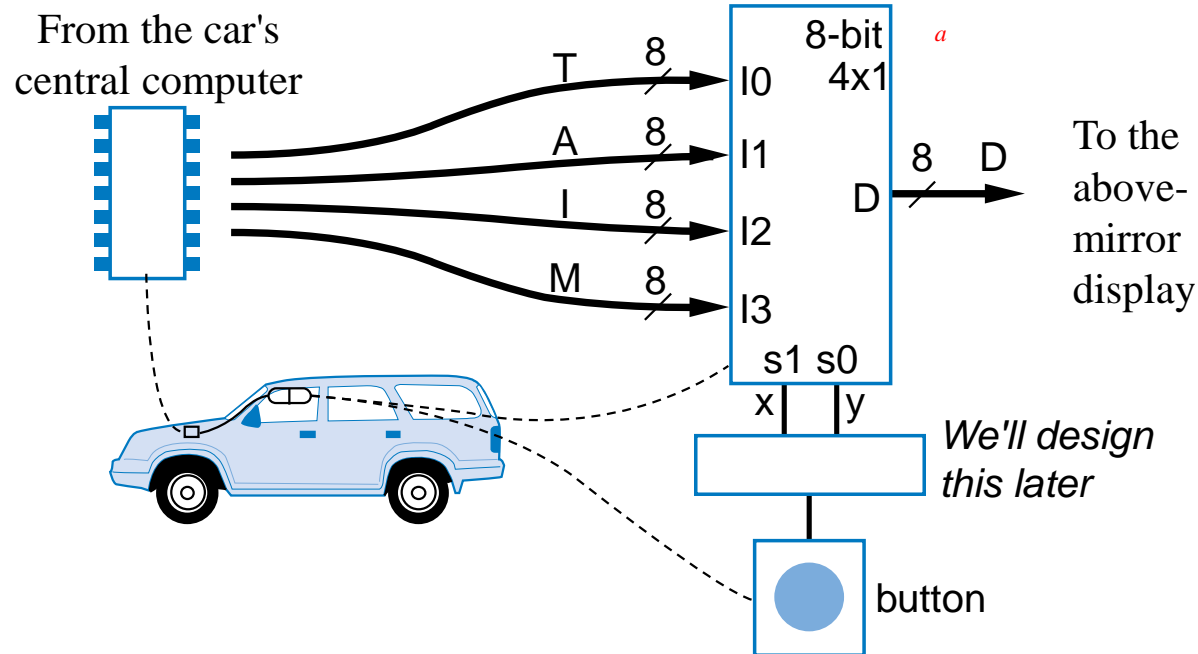
Muxes Commonly Together – N-bit Mux



- 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

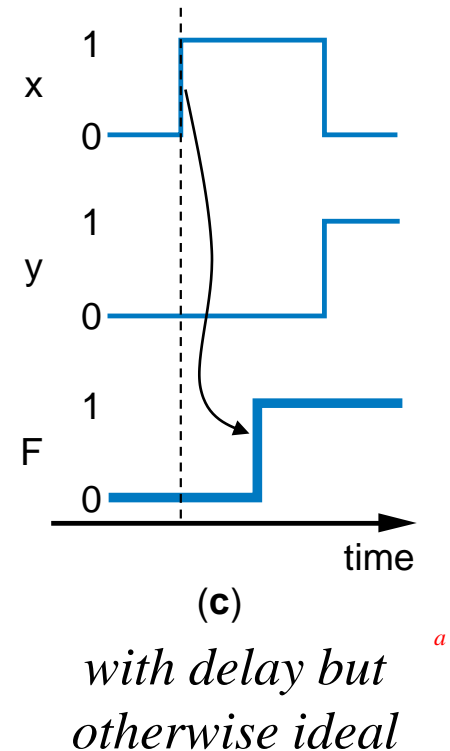
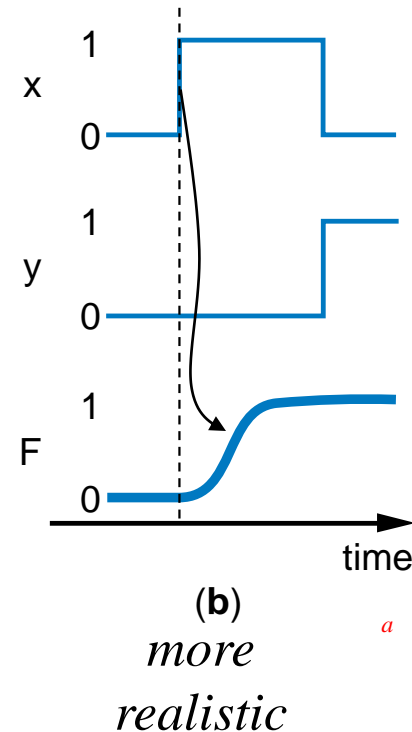
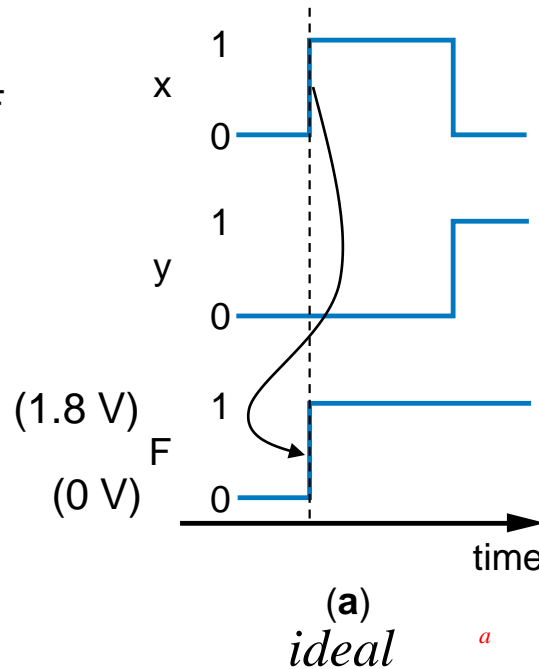
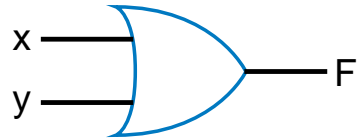


- Four possible display items
 - Temperature (T), Average miles-per-gallon (A), Instantaneous mpg (I), and Miles remaining (M) – each is 8-bits wide
 - Choose which to display on D using two inputs x and y
 - Pushing button sequences to the next item
 - Use 8-bit 4x1 mux



Additional Considerations

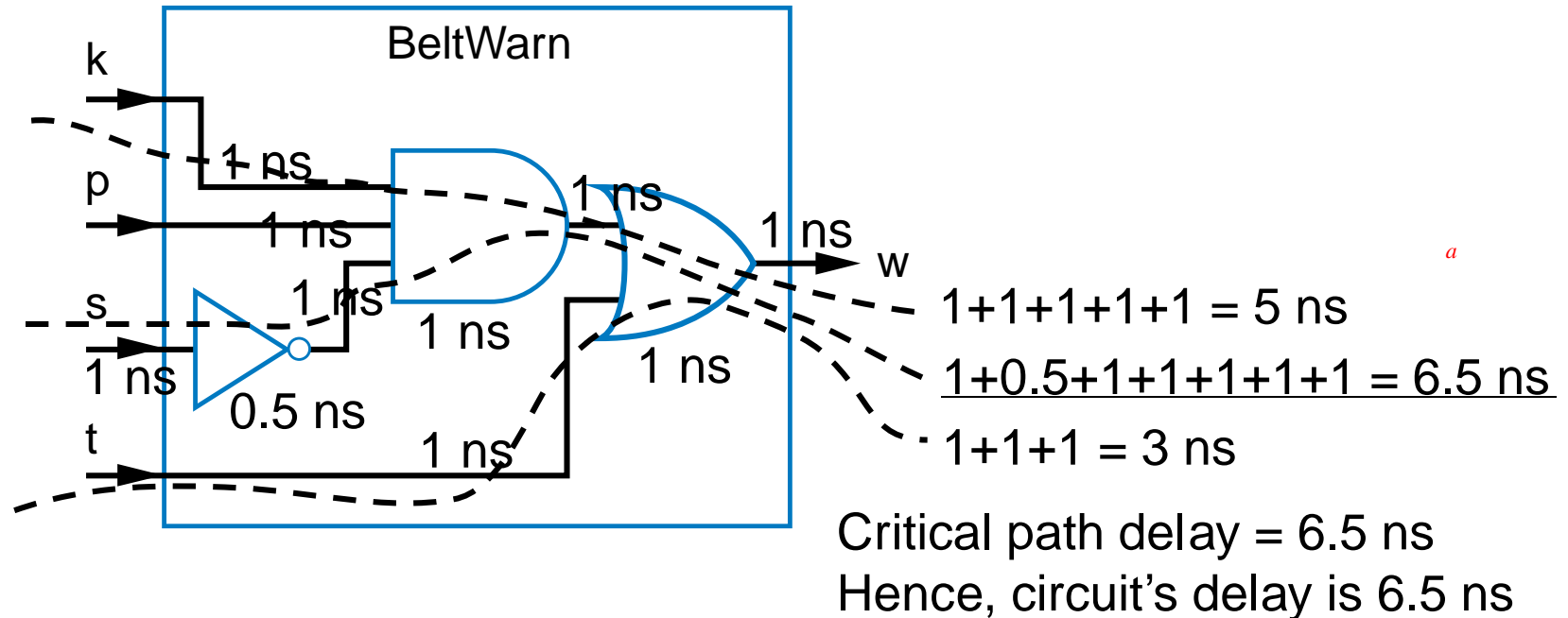
Non-Ideal Gate Behavior -- Delay



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



Circuit Delay and Critical Path

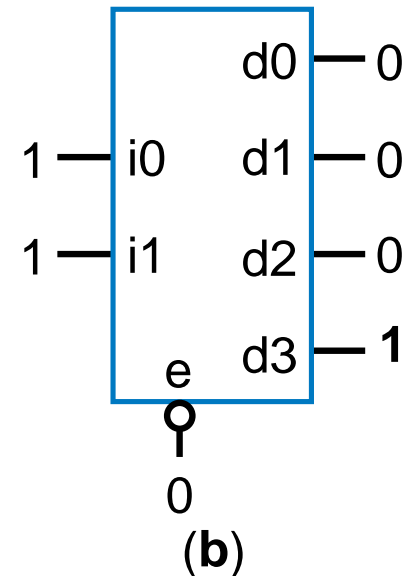
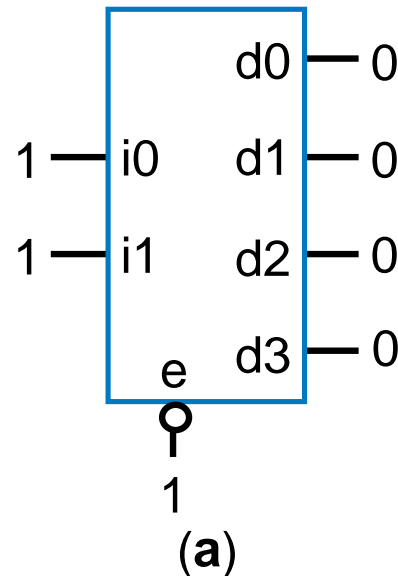


- Wires also have delay
- Assume gates and wires have delays as shown
- Path delay – time for input to affect output
- Critical path – path with longest path delay
- Circuit delay – delay of critical path

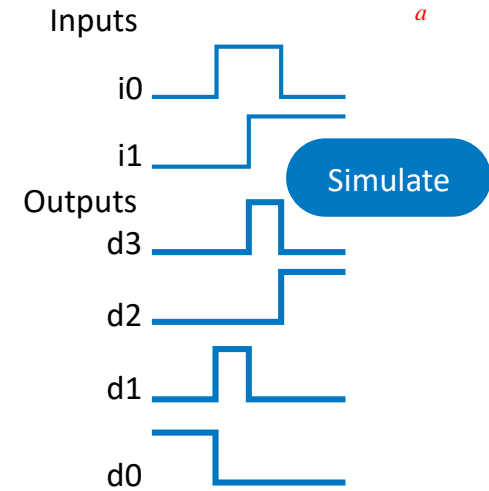
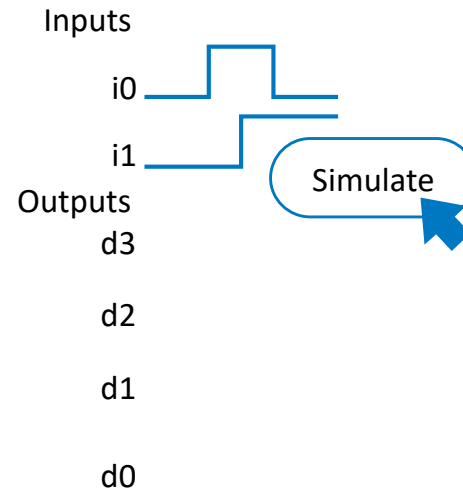
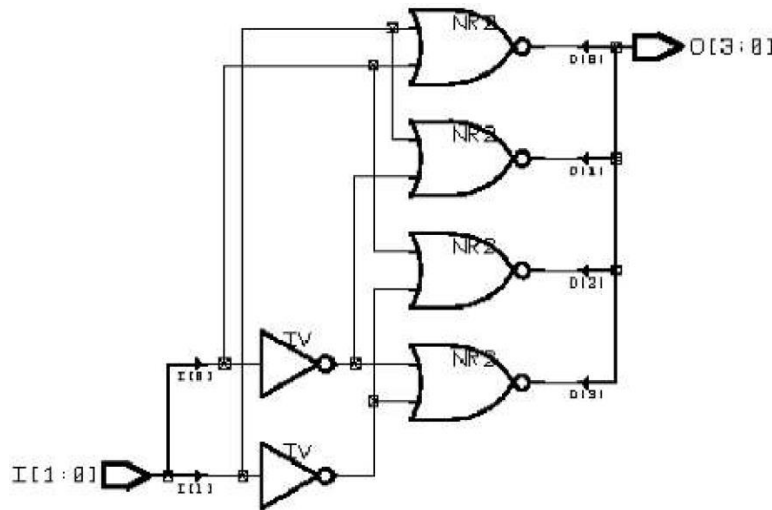


Active Low Inputs

- Data inputs: flow through component (e.g., mux data input)
- Control input: influence component behavior
 - Normally active high – 1 causes input to carry out its purpose
 - Active low – Instead, 0 causes input to carry out its purpose
 - Example: 2x4 decoder with active low enable
 - 1 disables decoder, 0 enables
 - Drawn using inversion bubble



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**



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

