# COMP3222/9222 Digital Circuits & Systems

20T3 L01 - Introduction

Course website: www.cse.unsw.edu.au/~cs3222

Oliver Diessel
O: K17-501B
E: o.diessel@unsw.edu.au
P: 9385 7384

# What you will learn in this class

- Introduction to the design of digital logic circuits
  - Boolean algebra, logic minimization, combinational logic components, sequential circuits, simple systems

- Principles of creating digital circuit designs
  - using VHDL hardware description language
  - simulation techniques to verify the correct working of designs
  - logic compilers to synthesize hardware
  - implementing and testing designs using programmable hardware

# What is logic design?

- What is design?
  - given a specification of a problem, come up with a way of solving it choosing appropriately from a collection of available tools, techniques and components
  - while meeting certain performance criteria for size, cost, power, beauty, elegance, etc.
- What is logic design?
  - **determining the collection of digital logic components and the interconnections between them to perform a specified data capturing, processing, storage, control and/or communication function**
  - which logic components to choose? – there are many implementation technologies, with differing costs/benefits
  - the design may need to be optimized and/or transformed to meet design constraints

# Applications of logic design

- Computer hardware & systems
  - design of processors, CPUs, systems, accelerators, custom chips, buses, interfaces, memories, peripherals
- Communications & networks
  - I/O devices, phones, switches, routers, base stations, satellites
- Embedded systems
  - consumer electronics, appliances, entertainment devices, IoT, medical devices, security, transport, robotics, mining, agriculture and manufacturing equipment
- Scientific instruments & equipment
  - simulation, testing, sensing, logging, reporting, analyzing

- Challenge:
  - Which human activities (a) don't yet, and (b) won't ever involve digital technology?

# Why study logic design?

- Obvious reasons
  - fundamental abstraction for implementing all digital devices
  - to study the digital design process
  - this course is part of the CompEng requirements

- Other important reasons
  - it's an important counterpart to software design
  - it's essential to furthering our understanding of the efficient implementation of computation
  - the inherent parallelism in hardware provides exposure to real parallelism on a large, fine-grained scale

low cost
high performance

# COMP3222/9222 details

- **Lectures, Weeks 1 – 5, 7-10**
  - Recordings available on website for asynchronous viewing
  - Synchronous meetings to review main points and discuss questions Wed 14–16 & Fri 14–16

- **Labs, Weeks 1 – 10**
  - Lab exercises and resources available on website
  - 2 hr online Q&A drop-in sessions: Wed 16–18, Thu 15–17, Fri 11–13
  - Submissions due roughly each week on Mondays at 23:59

- **Tutes, Weeks 2 – 10**
  - 1 hr online tutes: Thu 12–13, Thu 17–18

# COMP3222/9222 assessment

- **Assessment:**
  - 7 lab exs (**due the week after they are scheduled**): **40% total**
  - 4 fortnightly quizzes on theory in Weeks 3, 5, 7 & 9: **20% total**
  - 1 hr Final Theory Test: **15%**
  - 2 hr Final Practical Test: **25% (need to score >40% in Prac Exam to pass course)**

- **Supplementaries**
  - Only two eligibility criteria:
    1. Need to score >45% overall and >36% in Practical Test ⇒ final score = 50% if supp passed
    2. Valid medical excuse with certificate ⇒ final score calculated as if sitting normal exam
  - Note: there is no supp for missing the lab submission deadlines or quizzes

# COMP3222/9222 text

- Brown & Vranesic, Fundamentals of Digital Logic with VHDL Design, 3ed, McGraw-Hill
  - available from bookshop
  - available for short loan from the Library
  - tutorial problems drawn from text
  - extracts of Ch 1 & 2 provided on course website
  - text is designed to be used with the FPGA board you will be using
    - contains tutorials on loading & using the CAD tools
    - provides a detailed reference on VHDL
    - will be an invaluable aid in the follow-on computer architecture & design project courses

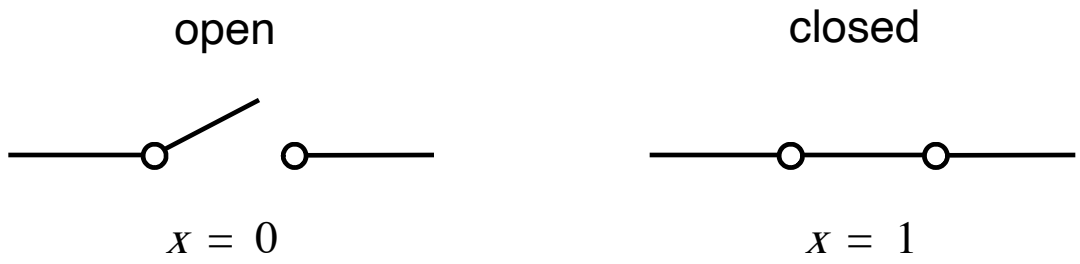
# Acknowledgement


- Most of the slides in this course are modified from the current text (Brown & Vranesic), or from the previous text: Katz, Contemporary Logic Design, 2ed, Pearson
- The material provided by these authors is gratefully acknowledged

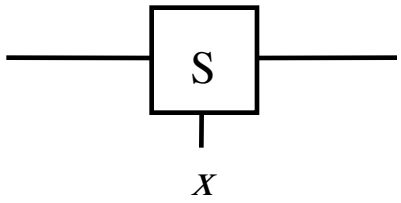
# Digital circuits

- The study of digital logic circuits is primarily motivated by their use in digital computers
- Logic circuits perform operations on digital signals and are usually implemented as *electronic circuits* in which the signal values are restricted to a few discrete values
  - In *binary* logic circuits, there are only two values, 0 and 1
  - *Decimal* logic circuits would have 10 values
- In contrast, in analog circuits, signals may take on continuous values between maximum and minimum levels
- We will only consider binary digital circuits
  - These are dominant because of their simplicity
  - The simplest binary element is a switch that has two states

# A binary switch



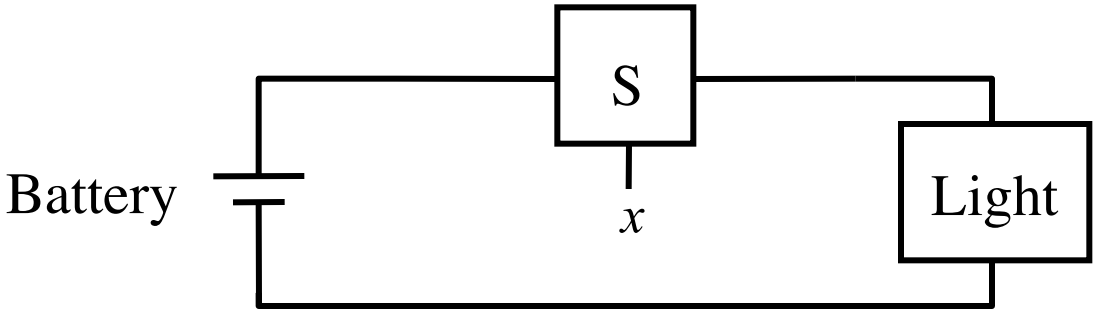

(a) Two states of a switch




(b) Symbol for a switch

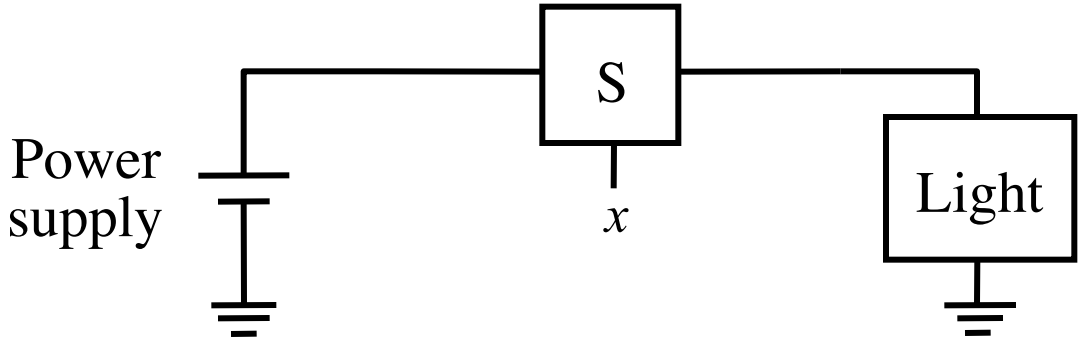
- We say the switch is controlled by input variable *x*, and that the switch is open if *x = 0* and closed if *x = 1*

# A light controlled by a switch



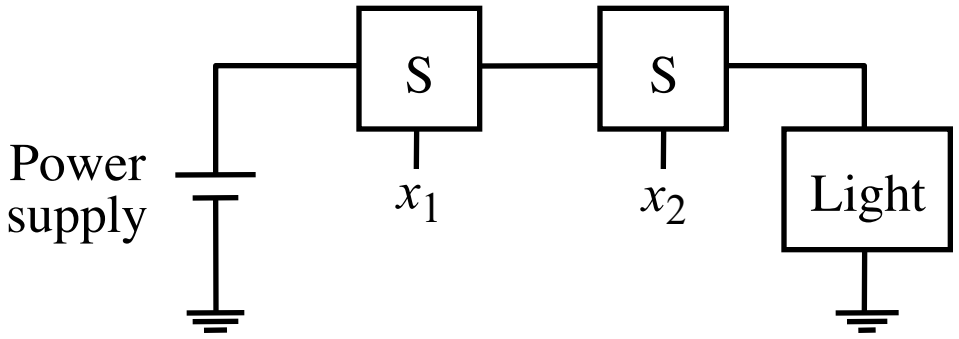

(a) Simple connection to a battery

- The state of the light L is dependent on the state of the switch
- The light is on, *L = 1,* if the switch is closed, *x = 1*, and vice versa
- Thus *L(x) = x*



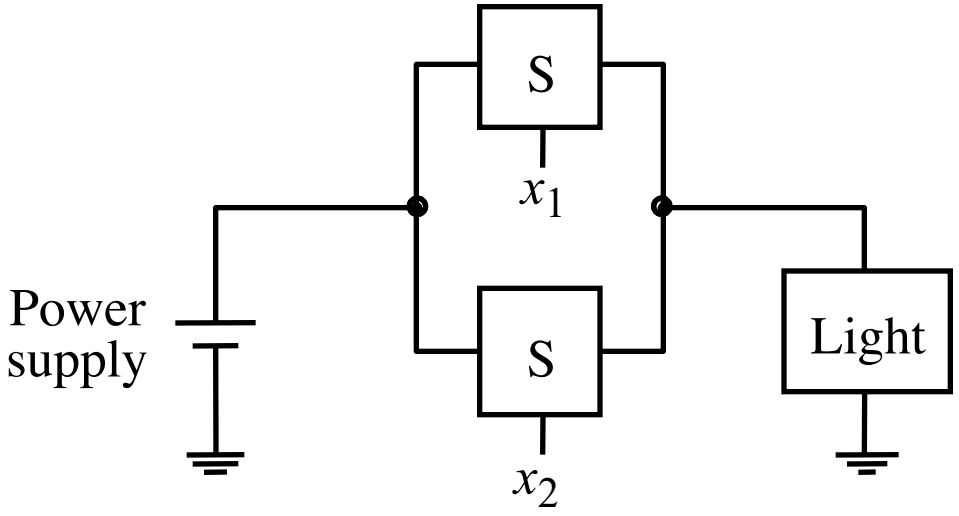

(b) An equivalent circuit using a ground connection as the return path

# Two basic functions




(a) The logical AND function (series connection)

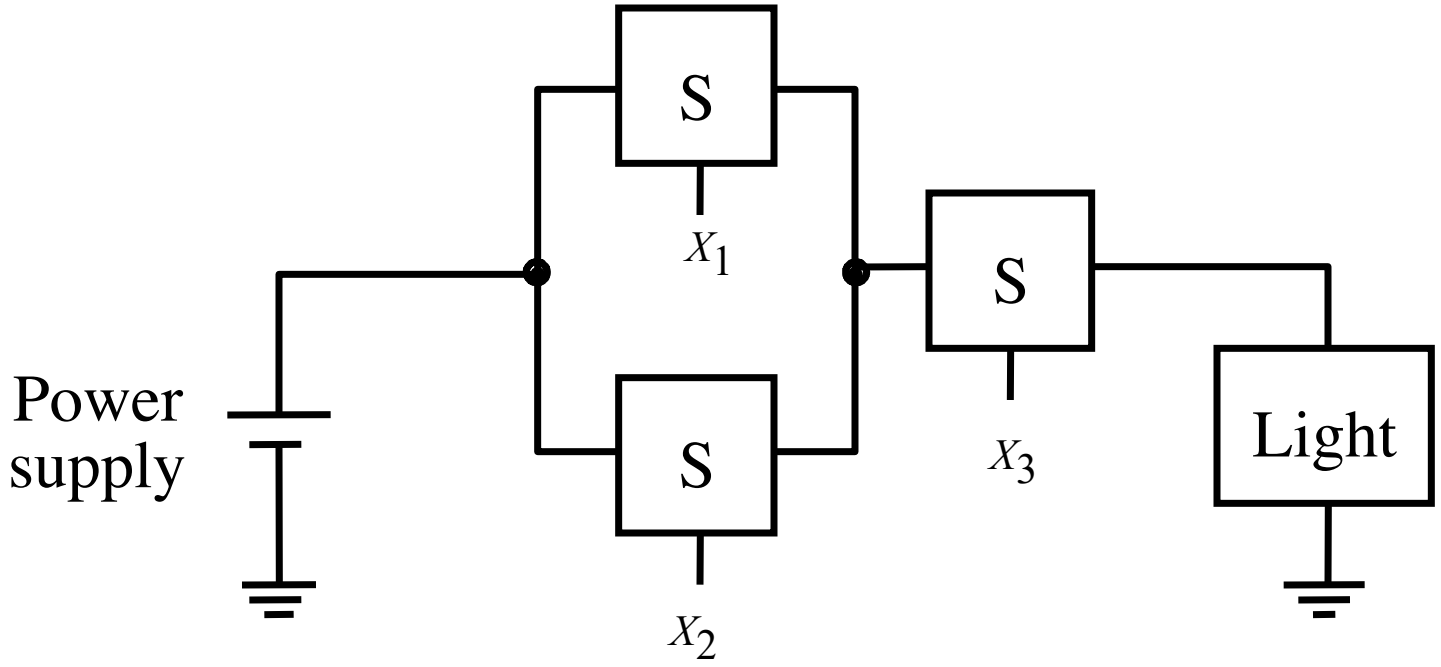
- In a series connection, **both** switches need to be closed for the light to be on
- Thus, $L(x_1, x_2) = x_1 \cdot x_2$




(b) The logical OR function (parallel connection)

- In a parallel connection, **either one** of the switches need to be closed for the light to be on
- Thus, $L(x_1, x_2) = x_1 + x_2$

# A series-parallel connection



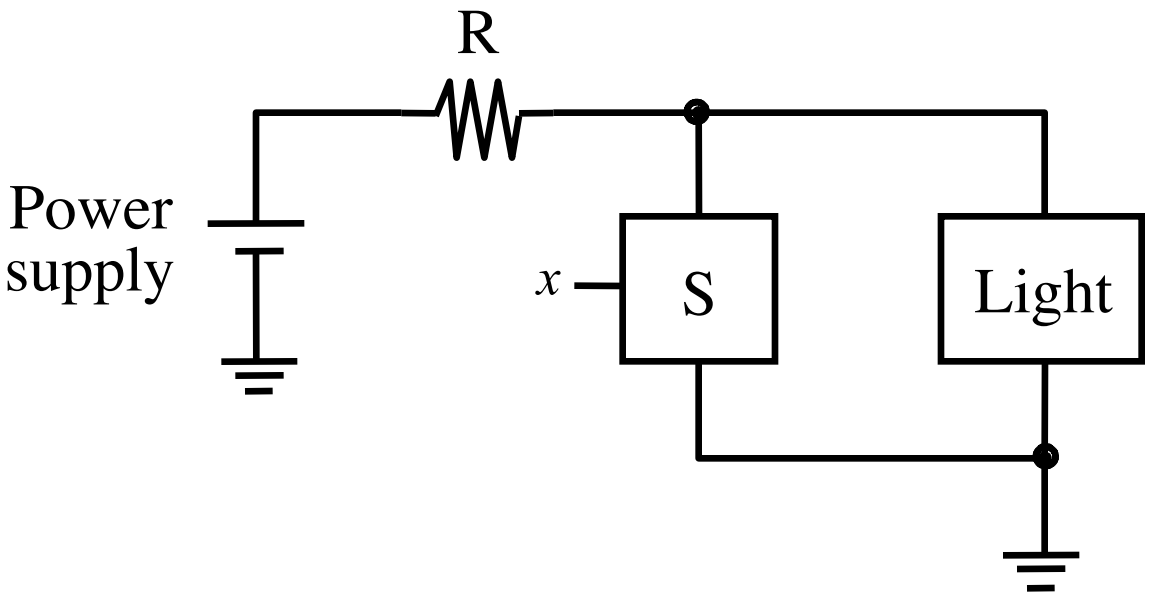

$L(x_1, x_2, x_3) = ?$

*Which function determines the output of the light?*

*How many possible states are there for this circuit?*

*How many of these have the light on?*

# Inversion




- Since the switch, when it is closed, short-circuits the potential difference across the light:

  $L(x) = \overline{x}$, where $L = 1$ if $x = 0$ and $L = 0$ if $x = 1$
- The complement op, NOT, is expressed in many ways:

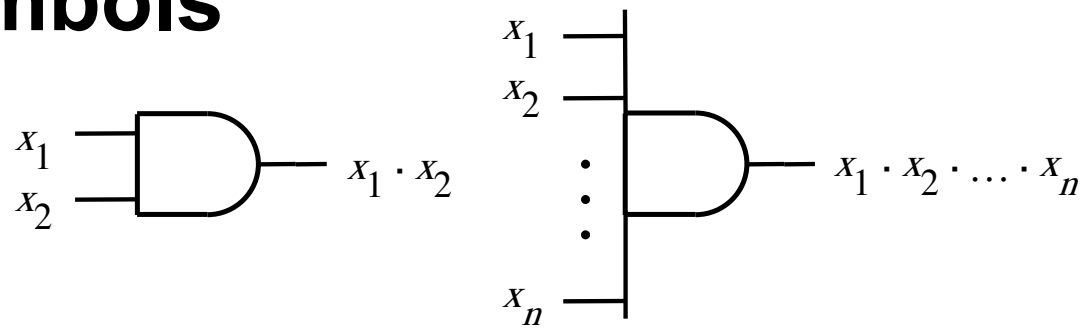
  $\overline{x} = x' = !x = \sim x =$ NOT $x$

# Truth tables for the AND, OR and NOT operations

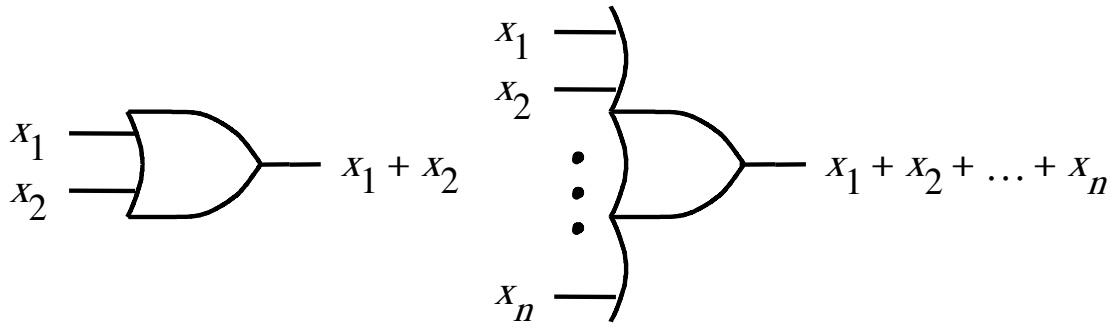
| $x_1$ | $x_2$ | $x_1 \cdot x_2$ | $x_1 + x_2$ | $\overline{x_1}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |
| | | AND | OR | NOT |

# Basic gate symbols

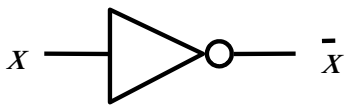
- Each logic operation can be implemented electronically with transistors, resulting in a circuit element called a *logic gate*
- Each logic gate has one or more inputs and one output that is a function of its inputs
- A logic circuit is often described by drawing a circuit diagram, or *schematic,* consisting of graphical symbols representing the logic gates




(a) AND gates

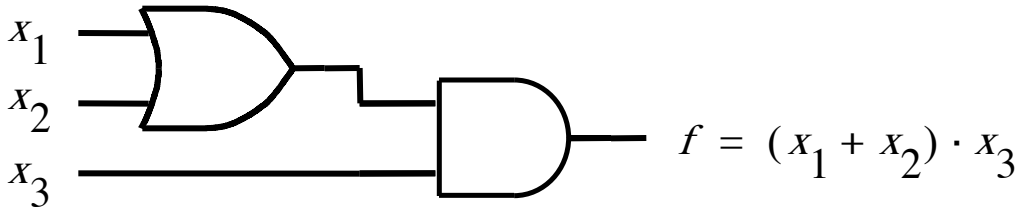



(b) OR gates



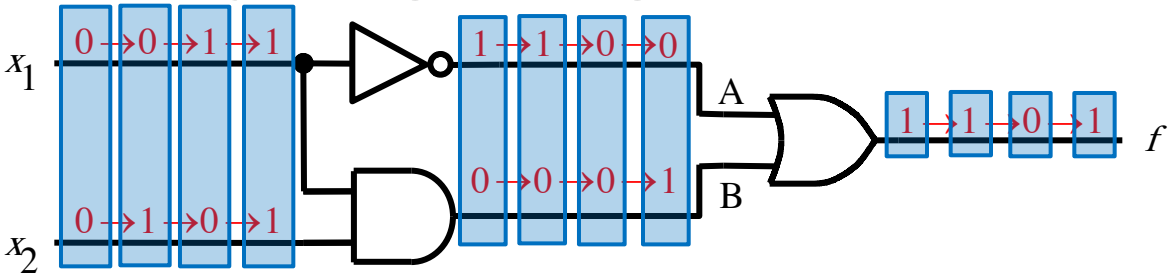

(c) NOT gate

# The function from L01/S14




- A larger circuit is implemented by a *network* of gates
- Such circuits are called *logic networks* or *logic circuits*
- The complexity of a given network (in terms of the gate count & number of gate inputs) has a direct impact on its cost
- In order to reduce manufactured cost, we seek ways to implement logic circuits as inexpensively as possible

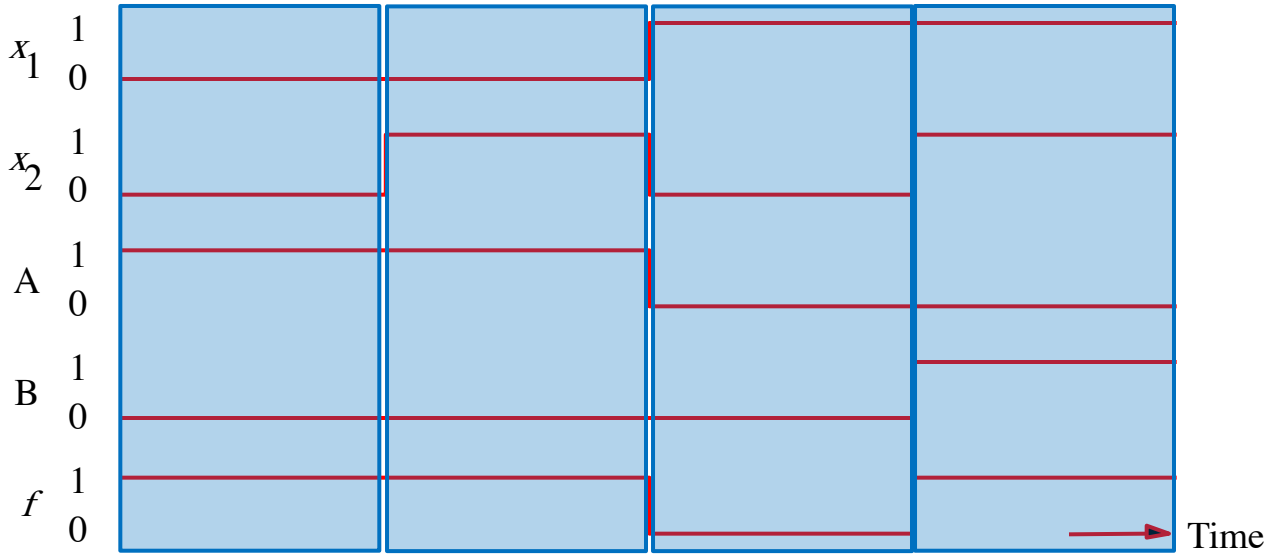
# Analyzing a logic network



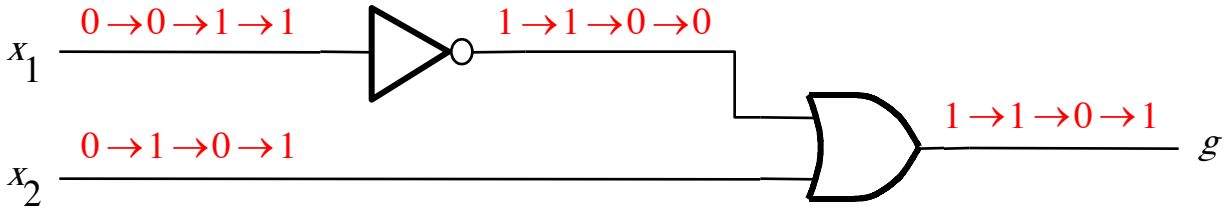

(a) Network that implements $f = A + B = \bar{x}_1 + x_1 \cdot x_2$

| $x_1$ | $x_2$ | $A$ | $B$ | $f(x_1, x_2)$ |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |

(b) Truth table




(c) Timing diagram




(d) Network that implements $g = \bar{x}_1 + x_2$

Note that $g$ implements the same function as $f$ but at a lower cost!

# Axioms and theorems of Boolean algebra

The theory underlying the simplification of logic expressions and their circuit realization is founded on the axioms and theorems of *Boolean algebra*

- identity
  1. $X + 0 = X$ — *Dual* → 1D. $X \cdot 1 = X$
- null
  2. $X + 1 = 1$ — 2D. $X \cdot 0 = 0$
- idempotency:
  3. $X + X = X$ — 3D. $X \cdot X = X$
- involution:
  4. $(X')' = X$
- complementarity:
  5. $X + X' = 1$ — 5D. $X \cdot X' = 0$
- commutativity:
  6. $X + Y = Y + X$ — 6D. $X \cdot Y = Y \cdot X$
- associativity:
  7. $(X + Y) + Z = X + (Y + Z)$ — 7D. $(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$

X is a Boolean variable with two possible values: *true = 1* or *false = 0*

# Axioms and theorems of Boolean algebra

- distributivity:
  - 8. $X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)$
  - 8D. $X + (Y \cdot Z) = (X + Y) \cdot (X + Z)$
- uniting:
  - 9. $X \cdot Y + X \cdot Y' = X$
  - 9D. $(X + Y) \cdot (X + Y') = X$
- absorption:
  - 10. $X + X \cdot Y = X$
  - 10D. $X \cdot (X + Y) = X$
  - 11. $(X + Y') \cdot Y = X \cdot Y$
  - 11D. $(X \cdot Y') + Y = X + Y$
- factoring:
  - 12. $(X + Y) \cdot (X' + Z) = X \cdot Z + X' \cdot Y$
  - 12D. $X \cdot Y + X' \cdot Z = (X + Z) \cdot (X' + Y)$
- consensus:
  - 13. $(X \cdot Y) + (X' \cdot Z) + (Y \cdot Z) = X \cdot Y + X' \cdot Z$
  - 13D. $(X + Y) \cdot (X' + Z) \cdot (Y + Z) = (X + Y) \cdot (X' + Z)$

# Axioms and theorems of Boolean algebra

- de Morgan's:

  14. $(X + Y + ...)' = X' \bullet Y' \bullet ...$ 14D. $(X \bullet Y \bullet ...)' = X' + Y' + ...$

- generalized de Morgan's:

  15. $\overline{f}(X_1,X_2,...,X_n,0,1,+,\bullet) = f(\overline{X}_1,\overline{X}_2,...,\overline{X}_n,1,0,\bullet,+)$

- establishes relationship between • and +

Example:

$f = x_2 + x_1\overline{x}_3 \Rightarrow \overline{f} = \overline{x_2 + x_1\overline{x}_3}$

$= \overline{x}_2 \bullet \overline{x_1\overline{x}_3}$

$= \overline{x}_2 \bullet (\overline{x}_1 + \overline{\overline{x}}_3)$

$= \overline{x}_2 \bullet (\overline{x}_1 + x_3)$

# Axioms and theorems of Boolean algebra

- Duality
  - a dual of a Boolean expression is derived by replacing • by +, + by •, 0 by 1, and 1 by 0, and leaving variables unchanged
  - any theorem that can be proven is thus also proven for its dual!
  - is a "meta-theorem" (a theorem about theorems)

- duality:
  16. $X + Y + \ldots \Leftrightarrow X \bullet Y \bullet \ldots$
- generalized duality:
  17. $f(X_1,X_2,\ldots,X_n,0,1,+,\bullet) \Leftrightarrow f(X_1,X_2,\ldots,X_n,1,0,\bullet,+)$

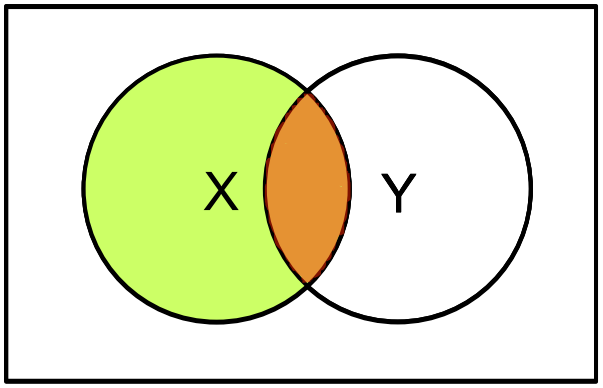
- Differs from deMorgan's Law
  - Duality is a statement about theorems
  - Duality is not a way of manipulating (re-writing) expressions

# Axioms and theorems of Boolean algebra

- Proofs of the axioms and theorems of Boolean algebra can be established in various ways
  - For example, the absorption theorem (X + X • Y = X) may be proven deductively, exhaustively or using set theory

LHS = X + X • Y
= X • 1 + X • Y by identity
= X • (1 + Y) by distributivity
= X • 1 by null
= X by identity

| X | Y | X • Y | X + X • Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |




Using set theory, the surrounding box represents the universe, logical AND is given by the intersection of sets, and logical OR is given by the union of sets

# Precedence of operations

- In the absence of parentheses, operations in a logic expression must be performed in the order:

  NOT, AND, and then OR

- Thus

  $x_1 \cdot x_2 + \overline{x_1} \cdot \overline{x_2}$ has the same effect as $(x_1 \cdot x_2) + ((\overline{x_1}) \cdot (\overline{x_2}))$

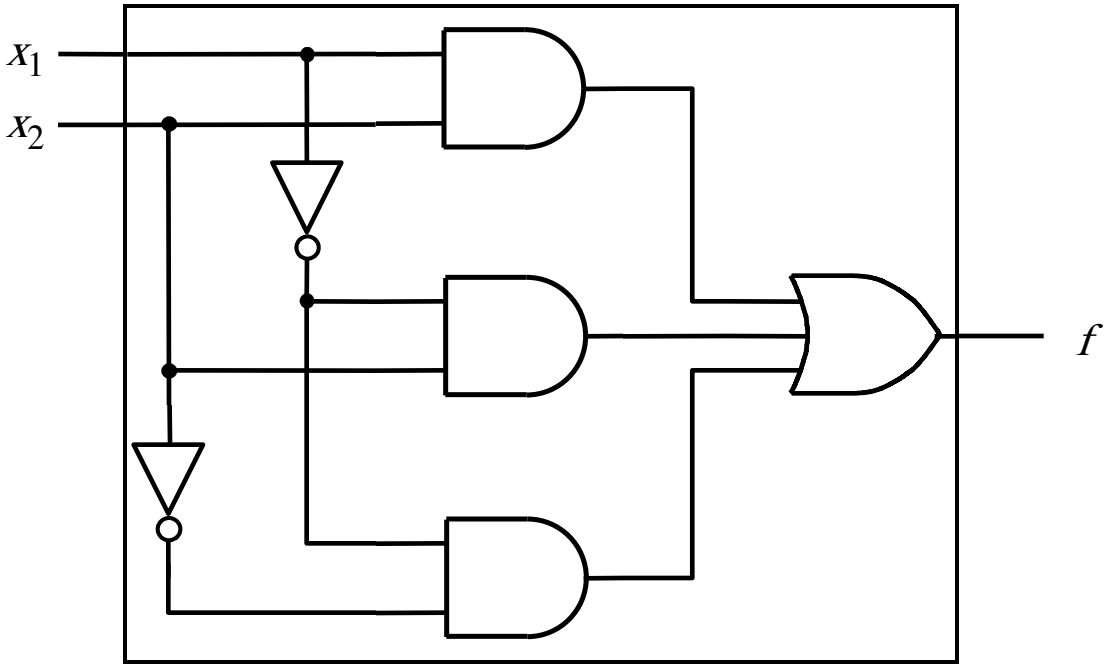
- Also, to simplify the appearance of logic expressions it is customary to omit the · operator when there is no ambiguity
  - The preceding expression can therefore be written as

    $x_1\, x_2 + \overline{x_1}\, \overline{x_2}$

# Synthesizing digital logic circuits using AND, OR and NOT gates

| $x_1$ | $x_2$ | $f(x_1, x_2)$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(a) Function to be realized




(b) Canonical sum-of-products realization

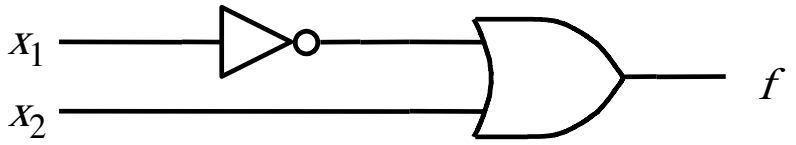
$f(x_1,x_2) = \overline{x}_1\overline{x}_2 + \overline{x}_1x_2 + x_1x_2$

$f(x_1,x_2) = \overline{x}_1\overline{x}_2 + \overline{x}_1x_2 + \overline{x}_1x_2 + x_1x_2$ *idempotency*

$f(x_1,x_2) = \overline{x}_1(\overline{x}_2 + x_2) + (\overline{x}_1 + x_1)x_2$ *distributivity*

$f(x_1,x_2) = \overline{x}_1 \cdot 1 + 1 \cdot x_2$ *complementarity*

$f(x_1,x_2) = \overline{x}_1 + x_2$ *identity*




(c) Minimal-cost realization

# Evaluating circuit cost

- The primary contributor to circuit cost is the number of transistors used
- #(Transistors used) ≈ circuit (chip) area
- Circuits with fewer gates and gates with fewer wires (inputs) require fewer transistors ⇒ less chip area
  - Important: the lowest level of abstraction we will use to design *digital circuits* is the GATE LEVEL, so we won't worry about how gates are implemented using transistors until the end of the course
- Unless stated otherwise, we will use the sum of the number of gates and the total number of gate inputs within a circuit as a measure of circuit cost
  - With this metric, circuit (b) on the previous slide has a cost of 17, whereas circuit (c) has a cost of 5

# Synthesis technique for logic circuits

1. Add a product (AND) term for each row in the truth table with function value *f = 1*
2. Form the sum (OR) of these terms to produce the function *f*
3. Simplify the expression using Boolean algebraic manipulation
4. Draw the circuit, complementing inputs where necessary, using AND gates for the product terms and ORing these together

# Minterms

The result of ANDing variables together

- For a function of *n* variables, a product term in which each of the *n* variables, $x_i$, appears exactly once is called a *minterm*
  - The variables may appear in a minterm either in uncomplemented or complemented form
  - For a given row of the truth table, the minterm is formed by including $x_i$ if $x_i = 1$ and by including $\overline{x}_i$ if $x_i = 0$
  - *Note that $m_i$=1 in row i of the truth table and $m_i$=0 in all other rows*

- Three-variable minterms

| Row number | $x_1$ | $x_2$ | $x_3$ | Minterm |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$ |
| 1 | 0 | 0 | 1 | $m_1 = \overline{x}_1\overline{x}_2 x_3$ |
| 2 | 0 | 1 | 0 | $m_2 = \overline{x}_1 x_2\overline{x}_3$ |
| 3 | 0 | 1 | 1 | $m_3 = \overline{x}_1 x_2 x_3$ |
| 4 | 1 | 0 | 0 | $m_4 = x_1\overline{x}_2\overline{x}_3$ |
| 5 | 1 | 0 | 1 | $m_5 = x_1\overline{x}_2 x_3$ |
| 6 | 1 | 1 | 0 | $m_6 = x_1 x_2\overline{x}_3$ |
| 7 | 1 | 1 | 1 | $m_7 = x_1 x_2 x_3$ |

Note that minterms are numbered according to which variables appear in uncomplemented form

# Sum-of-Products form

- A function *f* can be represented by an expression that is written as a *sum of minterms*, where each minterm is ANDed with the value of *f* for the corresponding value-tion of input variables
- For example, for the function of slide L01/S26,

$$f(x_1, x_2) = m_0 \cdot 1 + m_1 \cdot 1 + m_2 \cdot 0 + m_3 \cdot 1$$
$$= m_0 + m_1 + m_3 \qquad = \Sigma(m_0, m_1, m_3)$$
$$= \overline{x}_1\overline{x}_2 + \overline{x}_1 x_2 + x_1 x_2 \qquad = \Sigma m(0, 1, 3)$$

| $x_1$ | $x_2$ | $f(x_1, x_2)$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

- A logic expression consisting of product (AND) terms that are summed (ORed) is said to be in the *sum-of-products (SOP)* form.
- If each product term is a minterm, then the expression is called a *canonical sum-of-products* for the function *f*
  - Every Boolean fn has just ONE ***canonical*** SOP representation

# Example 2.3

- Consider the function $f(x_1, x_2, x_3) = \Sigma m(2, 3, 4, 6, 7)$

- The canonical SOP expression is given using minterms
  $f(x_1, x_2, x_3) = m_2 + m_3 + m_4 + m_6 + m_7$
  $= \overline{x}_1 x_2 \overline{x}_3 + \overline{x}_1 x_2 x_3 + x_1 \overline{x}_2 \overline{x}_3 + x_1 x_2 \overline{x}_3 + x_1 x_2 x_3$

- The expression can be simplified using algebraic manipulation:
  $f = \overline{x}_1 x_2 (\overline{x}_3 + x_3) + x_1 (\overline{x}_2 + x_2) \overline{x}_3 + x_1 x_2 (\overline{x}_3 + x_3)$
  $= \overline{x}_1 x_2 + x_1 \overline{x}_3 + x_1 x_2$ …SOP form
  $= (\overline{x}_1 + x_1) x_2 + x_1 \overline{x}_3$ …Not in SOP form
  $= x_2 + x_1 \overline{x}_3$ …SOP form

# Maxterms

- The principle of *duality* suggests that if it is possible to synthesize a function *f* by considering the rows in the truth table for which *f* = *1*, then it should be possible to synthesize *f* by considering the rows for which *f* = *0*
- This alternative approach uses the complements of minterms, which are called *maxterms*

# Three-variable minterms and maxterms

$M_0 = \overline{m_0} = \overline{\overline{x}_1\overline{x}_2\overline{x}_3} = \overline{\overline{x}}_1 + \overline{\overline{x}}_2 + \overline{\overline{x}}_3$

| Row number | $x_1$ | $x_2$ | $x_3$ | Minterm | (derived using DeMorgan) Maxterm | $m_0$ | $M_0$ ($\overline{m}_0$) |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$ | $M_0 = x_1 + x_2 + x_3$ | 1 | 0 |
| 1 | 0 | 0 | 1 | $m_1 = \overline{x}_1\overline{x}_2x_3$ | $M_1 = x_1 + x_2 + \overline{x}_3$ | 0 | 1 |
| 2 | 0 | 1 | 0 | $m_2 = \overline{x}_1x_2\overline{x}_3$ | $M_2 = x_1 + \overline{x}_2 + x_3$ | 0 | 1 |
| 3 | 0 | 1 | 1 | $m_3 = \overline{x}_1x_2x_3$ | $M_3 = x_1 + \overline{x}_2 + \overline{x}_3$ | 0 | 1 |
| 4 | 1 | 0 | 0 | $m_4 = x_1\overline{x}_2\overline{x}_3$ | $M_4 = \overline{x}_1 + x_2 + x_3$ | 0 | 1 |
| 5 | 1 | 0 | 1 | $m_5 = x_1\overline{x}_2x_3$ | $M_5 = \overline{x}_1 + x_2 + \overline{x}_3$ | 0 | 1 |
| 6 | 1 | 1 | 0 | $m_6 = x_1x_2\overline{x}_3$ | $M_6 = \overline{x}_1 + \overline{x}_2 + x_3$ | 0 | 1 |
| 7 | 1 | 1 | 1 | $m_7 = x_1x_2x_3$ | $M_7 = \overline{x}_1 + \overline{x}_2 + \overline{x}_3$ | 0 | 1 |

Observe the value of the example minterm $m_0$ and maxterm $M_0$ for each valuation of the variables $x_1$, $x_2$, and $x_3$

*Note that maxterm $M_i$=0 in row i of the truth table and $M_i$=1 for all other rows*

# Product-of-Sums form

- If a given function $f$ is specified by a truth table, then its complement $\overline{f}$ can be represented by a sum of minterms for which $\overline{f} = 1$, which are the rows of $f$ where $f = 0$.
- For example, for the L01/S26 function, $f(x_1,x_2) = \Sigma m(0,1,3)$, $\overline{f}(x_1,x_2) = m_2 = x_1\overline{x}_2$
- If we complement this expression using DeMorgan's theorem, the result is $\overline{\overline{f}} = f = \overline{x_1\overline{x}_2} = \overline{x}_1 + x_2 = M_2$
- The key point is that $f = \overline{m}_2 = M_2 = \Pi(M_2) = \Pi M(2)$

- A logic expression consisting of sum (OR) terms that are factors of a logical product (AND) is said to be of the *product-of-sums (POS)* form
- If each sum term is a maxterm, then the expression is called a *canonical product-of-sums* for the given function

# Example 2.4

$f(x_1, x_2, x_3) = \Sigma m(2, 3, 4, 6, 7)$

- Consider again the function of Example 2.3. Instead of using the minterms, we can specify this function as a product of maxterms for which $f = 0$, namely,

  $f(x_1, x_2, x_3) = \Pi M(0, 1, 5)$

- Then the canonical POS expression is given by:

  $f(x_1, x_2, x_3) = M_0 \cdot M_1 \cdot M_5$

  $= (x_1 + x_2 + x_3)(x_1 + x_2 + \overline{x}_3)(\overline{x}_1 + x_2 + \overline{x}_3)$

- A simplified POS expression can then be derived as:

  $f = ((x_1 + x_2) + x_3)((x_1 + x_2) + \overline{x}_3)(x_1 + (x_2 + \overline{x}_3))(\overline{x}_1 + (x_2 + \overline{x}_3))$

  $= ((x_1 + x_2) + x_3\overline{x}_3)(x_1\overline{x}_1 + (x_2 + \overline{x}_3))$

  $= (x_1 + x_2)(x_2 + \overline{x}_3)$ … *by the uniting theorem*

  What's the cost of this expression?

- And note that use of the distributive property leads to:

  $f = x_2 + x_1\overline{x}_3$

  Is this expression in SOP or POS form? What's its cost?

# Exercise for you to do

1. Derive canonical SOP and POS expressions for the three-valued function below.
2. Minimize each expression and sketch the resulting circuits.
3. Compare the costs of all designs

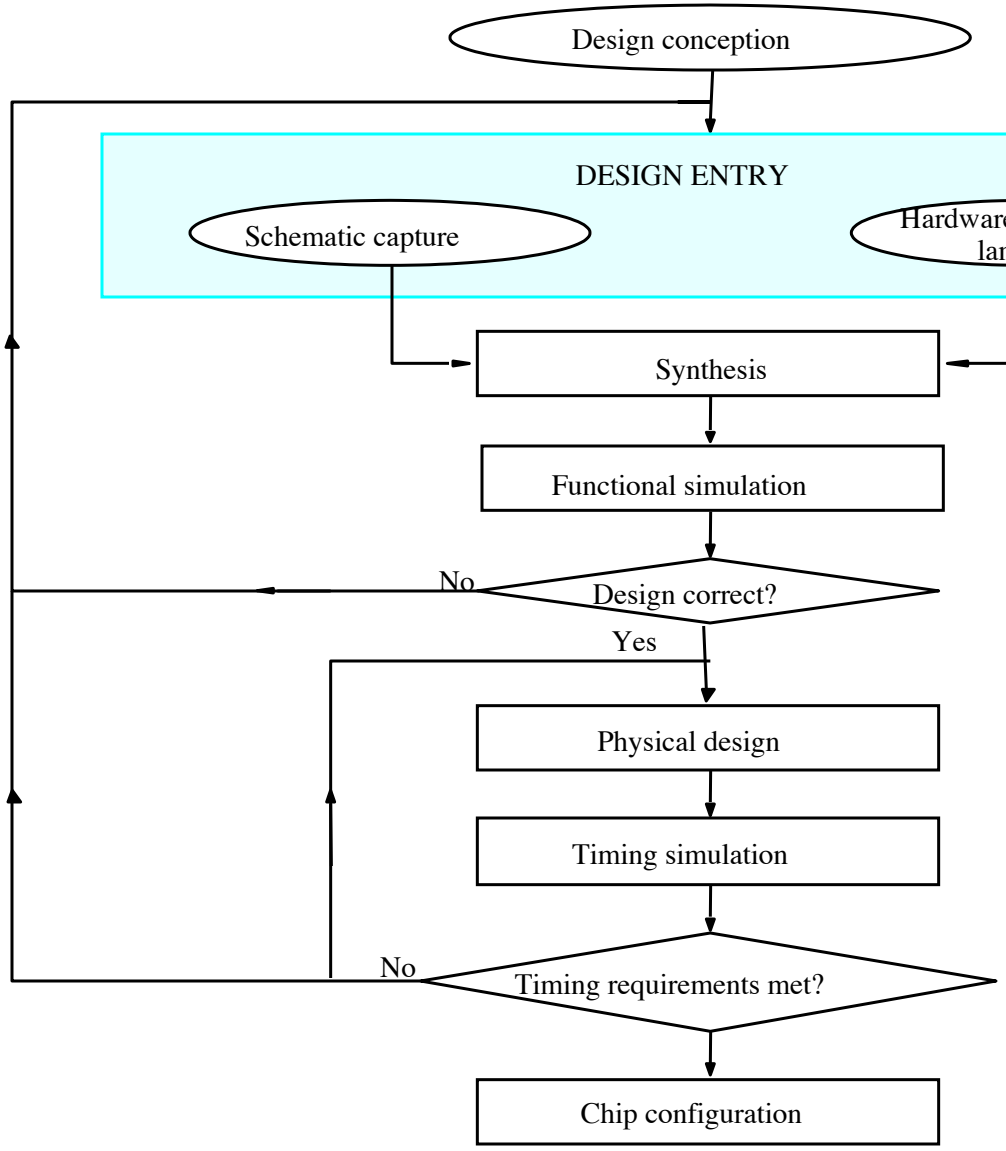
| Row Number | $x_1$ $x_2$ $x_3$ | $f(x_1, x_2, x_3)$ |
|---|---|---|
| 0 | 0 0 0 | 0 |
| 1 | 0 0 1 | 1 |
| 2 | 0 1 0 | 0 |
| 3 | 0 1 1 | 0 |
| 4 | 1 0 0 | 1 |
| 5 | 1 0 1 | 1 |
| 6 | 1 1 0 | 1 |
| 7 | 1 1 1 | 0 |

# Summary so far

1. Logic design is concerned with finding the cheapest implementation of *combinational* (a.k.a. *combinatorial*) logic functions
   - Circuit cost is measured by summing the number of gates and the total number of gate inputs
2. The laws of Boolean algebra can be used to minimize the cost of a combinational logic function
   - Typically, these are applied one at a time with the objective of eliminating one or more variables from a term or whole terms from the function
3. There are two canonical representations of combinational logic functions:
   - Canonical sum-of-products represents a function as a *sum (OR) of minterms*
   - Canonical product-of-sums represents a function as a *product (AND) of maxterms*

Computer Aided Design

# A typical CAD flow for logic design




Hardware Description Language

- HDL is similar to a computer programming language except that it is used to describe hardware; NOT to be executed on a processor
  - Similar to a markup language
- We focus on one of two IEEE standard languages that are supported by virtually all vendors that provide digital design technology
  - We study VHDL (Very High Speed Integrated Circuit HDL)
  - The other is Verilog

# VHDL

Use of a graphical tool to provide a logic circuit diagram

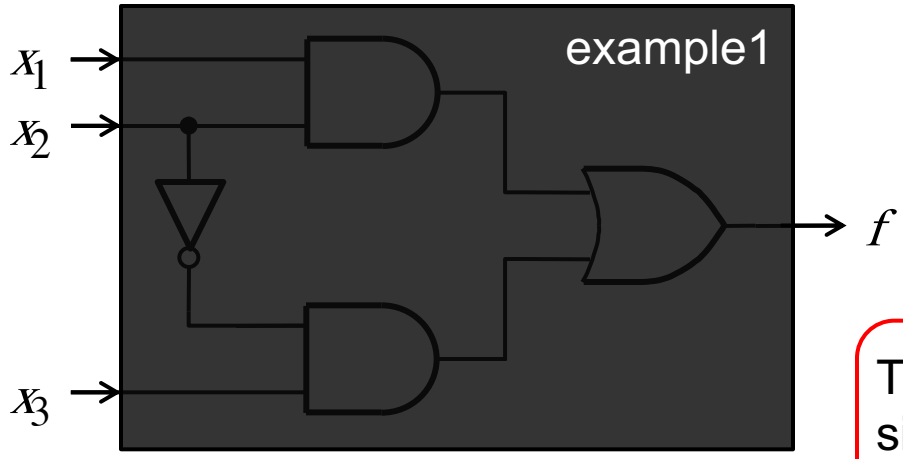
- In comparison to *schematic capture*, use of a hardware description language, such as VHDL, to capture a design's intent offers a number of advantages
  - VHDL source code is plain text, which means it can easily be edited, copied and searched; the designer can also easily include documentation explaining how the design works
  - Being a standard language, VHDL provides design portability – a design specified in VHDL can be implemented in different types of chips and with CAD tools from different vendors
  - Portability is useful because digital circuit technology changes rapidly – by using a standard language, the designer can focus on the functionality of the circuit without being overly concerned with the details of the implementation technology
  - Together with its wide use, these features encourage code sharing and reuse, which aids productivity

# A little VHDL history

- Developed in the 1980s, when integrated circuit technology was rapidly advancing, as a standard means of documenting complex digital circuits
- Became IEEE standard 1076 in 1987, and was revised in 1993 as IEEE 1164. Updates occurred in 2000 and 2002, as well as most recently, in 2008.
- Originally conceived for documenting circuits, and for modelling circuit behaviour, which allowed it to be used as input to programs for simulating circuit operation
- More recently, it has become popular to use it as a design entry method for CAD tools that synthesize hardware implementations
- VHDL is quite complex; we will restrict ourselves to the study of that subset of the language used for synthesis

# A simple logic fn and its VHDL entity declaration

- A VHDL *entity declaration* expresses what the design unit "looks like" to the outside world, i.e., describes its interface




```
ENTITY  example1 IS
    PORT ( x1, x2, x3    : IN     BIT ;
           f             : OUT    BIT ) ;
END  example1 ;
```

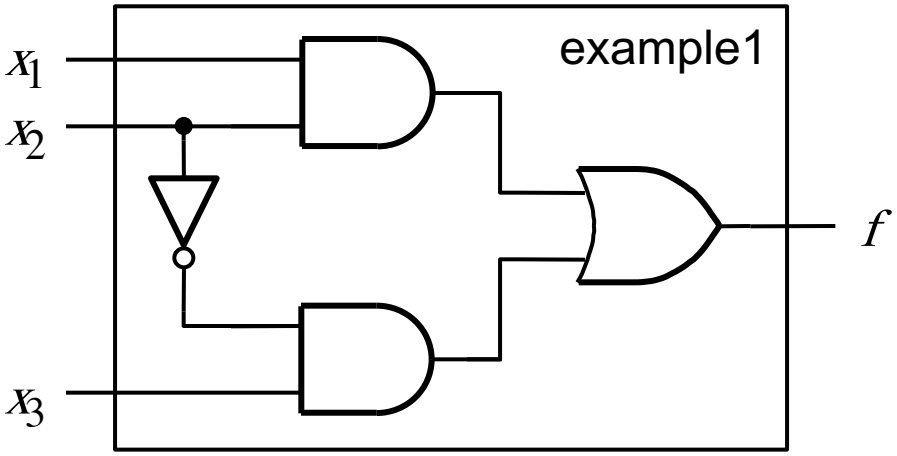
The "mode" of a signal indicates its direction

Signals of type BIT can assume values 0 and 1

Valid identifiers start with a letter, and comprise alpha-numeric and underscore characters

# A simple logic fn and its VHDL architecture

- The *architecture* of a VHDL design unit describes what the design "looks like" on the inside i.e. what its **behaviour** and/or **structure** is



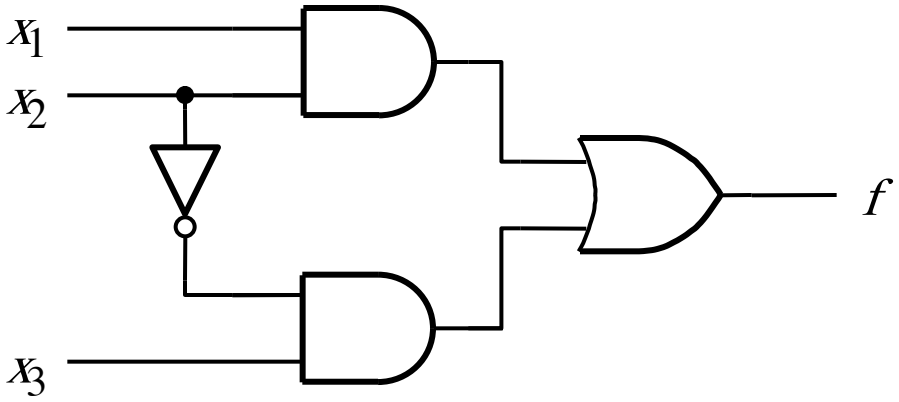

Here, the functionality of the circuit is expressed using a *simple signal assignment statement*

In VHDL, all statements need to be terminated by a semi-colon (;)

Note that in VHDL, the Boolean operators all have the same precedence, hence we must use parentheses to indicate the intended meaning

```
ARCHITECTURE LogicFunc OF example1 IS
BEGIN
        f <= (x1 AND x2) OR (NOT x2 AND x3) ;
END LogicFunc ;
```

# Complete VHDL design entity



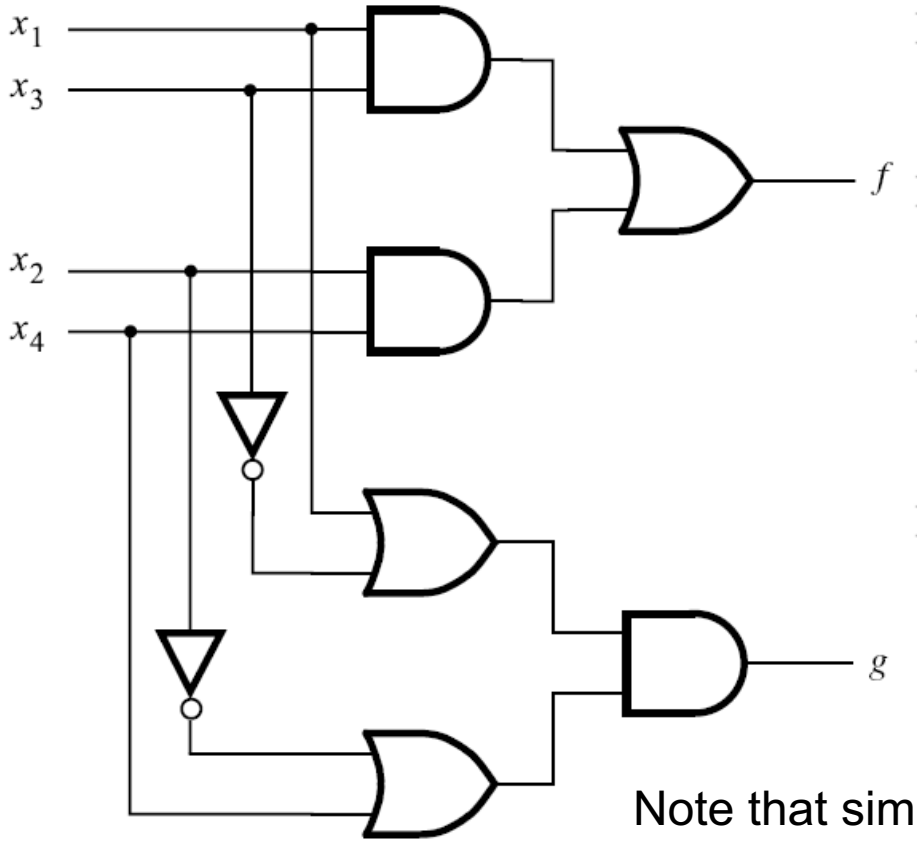

In our code we'll use capital letters to represent KEYWORDS in the code; obviously these can't be used as identifiers

Note that VHDL is not case sensitive

```
ENTITY example1 IS
    PORT ( x1, x2, x3   : IN    BIT ;
           f            : OUT   BIT ) ;
END example1 ;

ARCHITECTURE LogicFunc OF example1 IS
BEGIN
    f <= (x1 AND x2) OR (NOT x2 AND x3) ;
END LogicFunc ;
```

# Four-input example



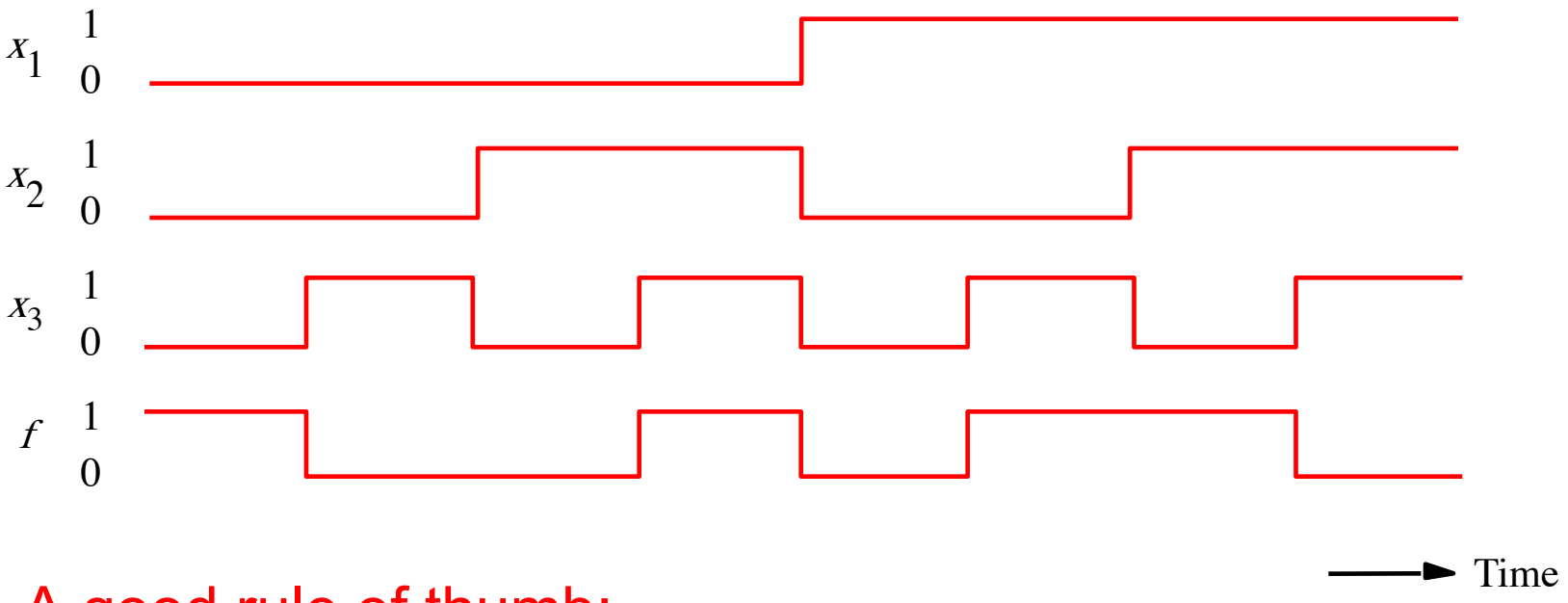

```
ENTITY example2 IS
    PORT ( x1, x2, x3, x4   : IN     BIT ;
           f, g             : OUT    BIT ) ;
END example2 ;

ARCHITECTURE LogicFunc OF example2 IS
BEGIN
    f <= (x1 AND x3) OR (x2 AND x4) ;
    g <= (x1 OR NOT x3) AND (NOT x2 OR x4) ;
END LogicFunc ;
```

Note that simple signal assignment statements are examples of *concurrent assignment statements* – they are all "evaluated" at the same time when a signal on the RHS changes value; therefore the order the statements are listed in does not matter

# Exercise for you to attempt

- For the timing diagram depicted below, synthesize the function $f(x_1, x_2, x_3)$ in the simplest SOP form and describe your implementation using VHDL




- A good rule of thumb:

  Can you visualize the circuit that should be synthesized from your VHDL code?