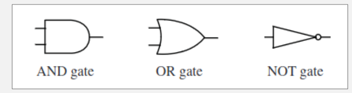
Key Points

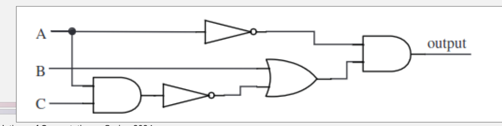
- terminology – logic gates, logic circuits, combinatorial logic circuit, feedback loop, disjunctive normal form
- processes / algorithms
  - building a logic circuit from a proposition
  - constructing a proposition from a logic circuit
  - using boolean algebra to simplify circuits
- theorems
  - every compound proposition is computed by a logic circuit with one output wire
  - every combinatorial logic circuit with one output computes the value of some compound proposition
  - it is possible to build a proposition with only $\wedge$, $\vee$, $\neg$ and in disjunctive normal form for any truth table where at least one of the output values is true
- applications to computers

Logic Circuits

- *logic gates* are electronic components that compute values of simple propositions



  - input and output wires can be in one of two states (on, off), which corresponds to the boolean values $\mathbb{T}$, $\mathbb{F}$
- *logic circuits* are built from connecting inputs and outputs of logic gates to each other



Constructing Circuits From Propositions

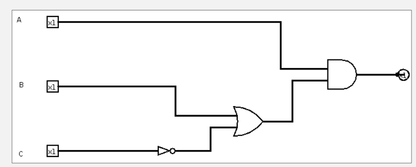
- algorithm
  - if the proposition contains operators other than $\wedge$, $\vee$, $\neg$, convert the proposition to a logically equivalent one using only $\wedge$, $\vee$, $\neg$
  - determine the *main operator* – the one that is applied last
  - add the corresponding logic gate to the circuit
  - repeat the last two steps for each of the compound propositions joined by this main operator, connecting their outputs to the inputs of this main operator
  - create one input for each propositional variable and connect them to the appropriate inputs

a) $A \wedge (B \vee \neg C)$ **b)** $(p \wedge q) \wedge \neg(p \wedge \neg q)$
c) $(p \vee q \vee r) \wedge (\neg p \vee \neg q \vee \neg r)$ **d)** $\neg(A \wedge (B \vee C)) \vee (B \wedge \neg A)$

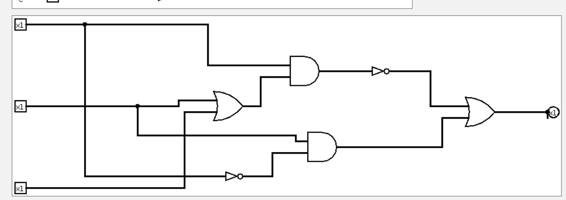
Constructing Circuits From Propositions

a) $A \wedge (B \vee \neg C)$ **b)** $(p \wedge q) \wedge \neg(p \wedge \neg q)$
c) $(p \vee q \vee r) \wedge (\neg p \vee \neg q \vee \neg r)$ **d)** $\neg(A \wedge (B \vee C)) \vee (B \wedge \neg A)$

(a)

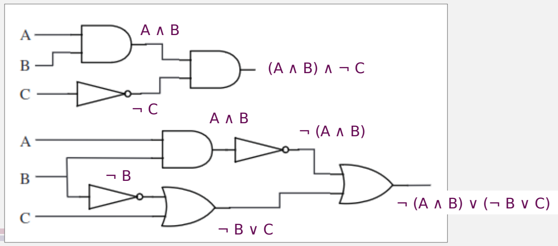


(d)



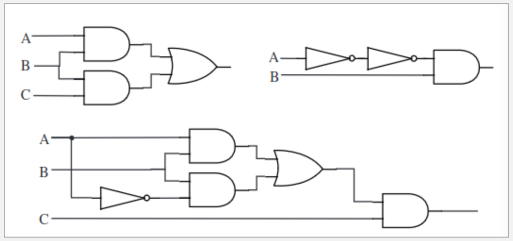
Constructing Propositions From Circuits

- algorithm
  - label the circuit's inputs with the name of a propositional variable
  - label each gate's output with the proposition consisting of the propositions represented by the gate's inputs combined with operator represented by the gate
  - the output from the final logic gate is the proposition



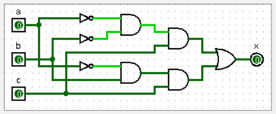
Simplifying Circuits

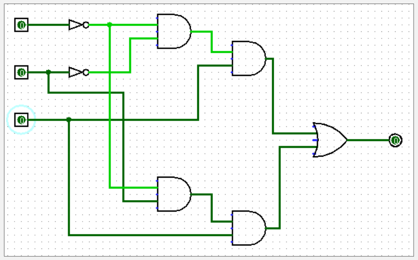
- algorithm
  - convert the circuit to propositional logic
  - use boolean algebra to simplify the proposition
  - construct the circuit corresponding to the simplified proposition



Simplifying Circuits

- also be alert to the possibility of reusing outputs from gates



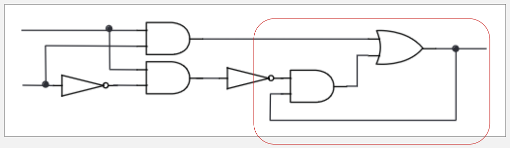


Theorems

- every compound proposition is computed by a logic circuit with one output wire
- rationale
  - apply the algorithm for converting propositions into circuits

Combinatorial Logic Circuits

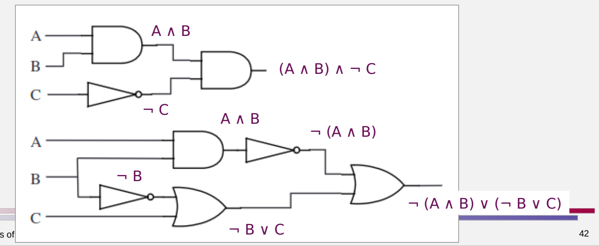
- a *combinatorial logic circuit* has no feedback loops
- a *feedback loop* occurs when an output of a gate is connected back to an input of the same gate



  - circuits with feedback loops do not compute compound propositions, but they are important for computer memories

Theorems

- every combinatorial logic circuit with one output computes the value of some compound proposition
- rationale
  - each wire represents the value of some proposition
  - the proposition represented by an output wire consists of the propositions represented by the input wires, joined by the logical operation corresponding to the gate



Disjunctive Normal Form

- a compound proposition is in *disjunctive normal form* if
  - it is a disjunction of conjunctions of simple terms, and
    - disjunction = ∨, conjunction = ∧, simple term = *p* or ¬*p*
  - each propositional variable occurs at most once in each conjunction, and
    - occurs as either *p* or ¬*p*, but not both
  - each conjunction occurs at most once in the disjunction
    - no repeats

$$(p \wedge q \wedge r) \vee (p \wedge \neg q \wedge r \wedge s) \vee (\neg p \wedge \neg q)$$
$$(p \wedge \neg q)$$
$$(A \wedge \neg B) \vee (\neg A \wedge B)$$
$$p \vee (\neg p \wedge q) \vee (\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge \neg q \wedge \neg r \wedge w)$$

Theorems

- [theorem 1.3] it is possible to build a proposition with only ∧, ∨, ¬ and in disjunctive normal form for any truth table where at least one of the output values is $\mathbb{T}$
- rationale
  - for each row of the table where the output value is $\mathbb{T}$, build a conjunction of simple terms –
    - for each variable *p* whose value is true in that row, include *p* in the conjunction
    - for each variable *q* whose value is false in that row, include ¬*q* in the conjunction
  - take the disjunction of all such conjunctions

p	q	r	output	
$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$	
$\mathbb{F}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$	$(\neg p \wedge \neg q \wedge r)$
$\mathbb{F}$	$\mathbb{T}$	$\mathbb{F}$	$\mathbb{F}$	
$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$	$(\neg p \wedge q \wedge r)$
$\mathbb{T}$	$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$	
$\mathbb{T}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{F}$	
$\mathbb{T}$	$\mathbb{T}$	$\mathbb{F}$	$\mathbb{F}$	
$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$	$p \wedge q \wedge r$

the conjunction is $\mathbb{T}$ only for the specific combination of values in that row

the disjunction is true only if at least one of the disjunctions is $\mathbb{T}$

Theorems

- [theorem 1.3] it is possible to build a proposition with only ∧, ∨, ¬ and in disjunctive normal form for any truth table where at least one of the output values is true
  - what if all of the output values are false?

p	q	r	output
F	F	F	F
F	F	T	𝔽
F	T	F	F
F	T	T	𝔽
T	F	F	F
T	F	T	F
T	T	F	F
T	T	T	𝔽

this is a contradiction – not really useful to express

workaround: accept 𝔽 as a proposition in disjunctive normal form

Applications to Computers

Why use logic circuits in computers?

- on, off can be interpreted as 1, 0
- numbers can be represented in binary

0	1	0	0	0	1	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
x128	x64	x32	x16	x8	x4	x2	x1

64 + 4 + 2

70

- arithmetic can be performed on numbers
  - can create truth tables which correspond to arithmetic involving binary numbers
  - theorem 1.3 means a logic circuit can be constructed for those truth tables
    - actually carrying out that process may only be practical for small circuits, but the goal of the proof is that it is possible

Half Adder

- for adding two 1-bit numbers

1) $0 + 0 = 0$

2) $0 + 1 = 1$ or $1 + 0 = 1$

3) $1 + 1 = 10$

Sum

the definition

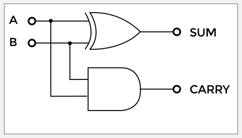
the corresponding truth table

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

propositions corresponding to the truth table

sum bit
(¬A ∧ B) ∨ (A ∧ ¬B)
≡ A ⊕ B

carry bit
(A ∧ B)

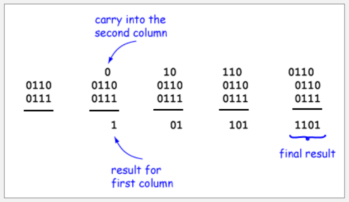


corresponding circuit
(half adder)

https://projects.raspberrypi.org/en/projects/halfadder
https://www.elprocus.com/half-adder-and-full-adder/
https://delightlylinux.wordpress.com/2014/09/17/binary-lesson-9-binary-addition/

Full Adder

- after the first (rightmost) column, each column involves adding three bits
  - the current bit from each number (A and B), plus the carry bit from the column to the right (Cin)



Input			Output	
A	B	Cin	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

truth table for adding three bits

https://www.eecis.udel.edu/~davis/cpeg222/AssemblyTutorial/Chapter-08/ass08_3.html

Full Adder

propositions corresponding to the truth table

Input			Output	
A	B	Cin	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

sum bit

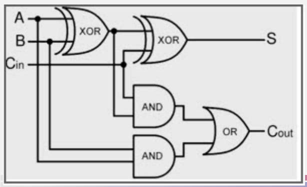
(¬A ∧ ¬B ∧ Cin) ∨
(¬A ∧ B ∧ ¬Cin) ∨
(A ∧ ¬B ∧ ¬Cin) ∨
(A ∧ B ∧ Cin)

carry bit

(¬A ∧ B ∧ Cin) ∨
(A ∧ ¬B ∧ Cin) ∨
(A ∧ B ∧ ¬Cin) ∨
(A ∧ B ∧ Cin)

can use boolean algebra to simplify these propositions to

sum bit – (A ⊕ B) ⊕ Cin
carry bit – (Cin ∧ (A ⊕ B)) ∨ (B ∧ A)

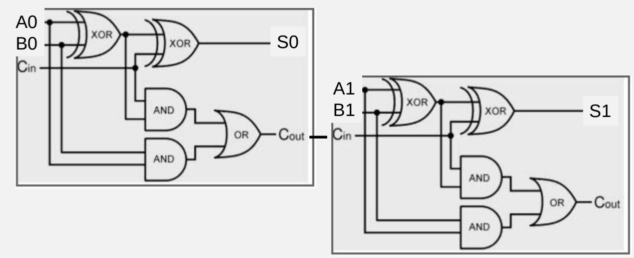


corresponding circuit

https://www.eecis.udel.edu/~davis/cpeg222/AssemblyTutorial/Chapter-08/ass08_3.html

Adders

- string *n* full adders together to add *n*-bit numbers
- e.g. 2-bit adder A1 A0 + B1 B0 = S1 S0



https://www.elprocus.com/half-adder-and-full-adder/