Boolean Algebra

- Boolean Algebra is named after George Boole, Professor of Mathematics in UCC in 19th Century
- Two value system *true* and *false*
- Set of basic axioms can be used to derive more complicated functions
- We can build digital circuits using these rules to implement binary logic in a processor

Boolean Axioms and One Variable Theorems

	Axiom		Dual
A1	$B = 0$ if $B \neq 1$	A1'	$B = 1$ if $B \neq 0$
A2	$\overline{0} = 1$	A2'	$\overline{1} = 0$
A3	$0 \wedge 0 = 0$	A3'	$0 \vee 0 = 0$
A4	$1 \wedge 1 = 1$	A4'	$1 \vee 1 = 1$
A5	$0 \wedge 1 = 1 \wedge 0 = 0$	A5'	$1 \vee 0 = 0 \vee 1 = 1$

	Theorem		Dual	Name
T1	$B \wedge 1 = B$	T1'	$B \vee 0 = B$	Identity
T2	$B \wedge 0 = 0$	T2'	$B \vee 1 = 1$	Null Element
T3	$B \wedge B = B$	T3'	$B \vee B = B$	Idempotency
T4	$B \wedge \overline{B} = 0$	T4'	$B \vee \overline{B} = 1$	Complements
T5	$\overline{\overline{B}} = \mathrm{B}$			Involution

Boolean Several Variable Theorems

	Theorem	Name
T6	$B \wedge C = C \wedge B$	Commutative
T6'	$B \vee C = C \vee B$	Commutative
T7	$(B \wedge C) \wedge D = B \wedge (C \wedge D)$	Associative
T7'	$(B \vee C) \vee D = B \vee (C \vee D)$	Associative
T8	$(B \wedge C) \vee (B \wedge D) = B \wedge (C \vee D)$	Distributive
T8'	$(B \vee C) \wedge (B \vee D) = B \vee (C \wedge D)$	Distributive
T9	$B \wedge (B \vee C) = B$	Covering
T9'	$B \vee (B \wedge C) = B$	Covering
T10	$(B \wedge C) \vee (B \wedge \overline{C}) = B$	Combining
T10'	$(B \vee C) \wedge (B \vee \overline{C}) = B$	Combining
T11	$\overline{B \wedge C} = \overline{B} \vee \overline{C}$	De Morgan's Law
T11'	$\overline{B \vee C} = \overline{B} \wedge \overline{C}$	De Morgan's Law

Simplifying Statements

Can use the theorems and axioms to simplify equations

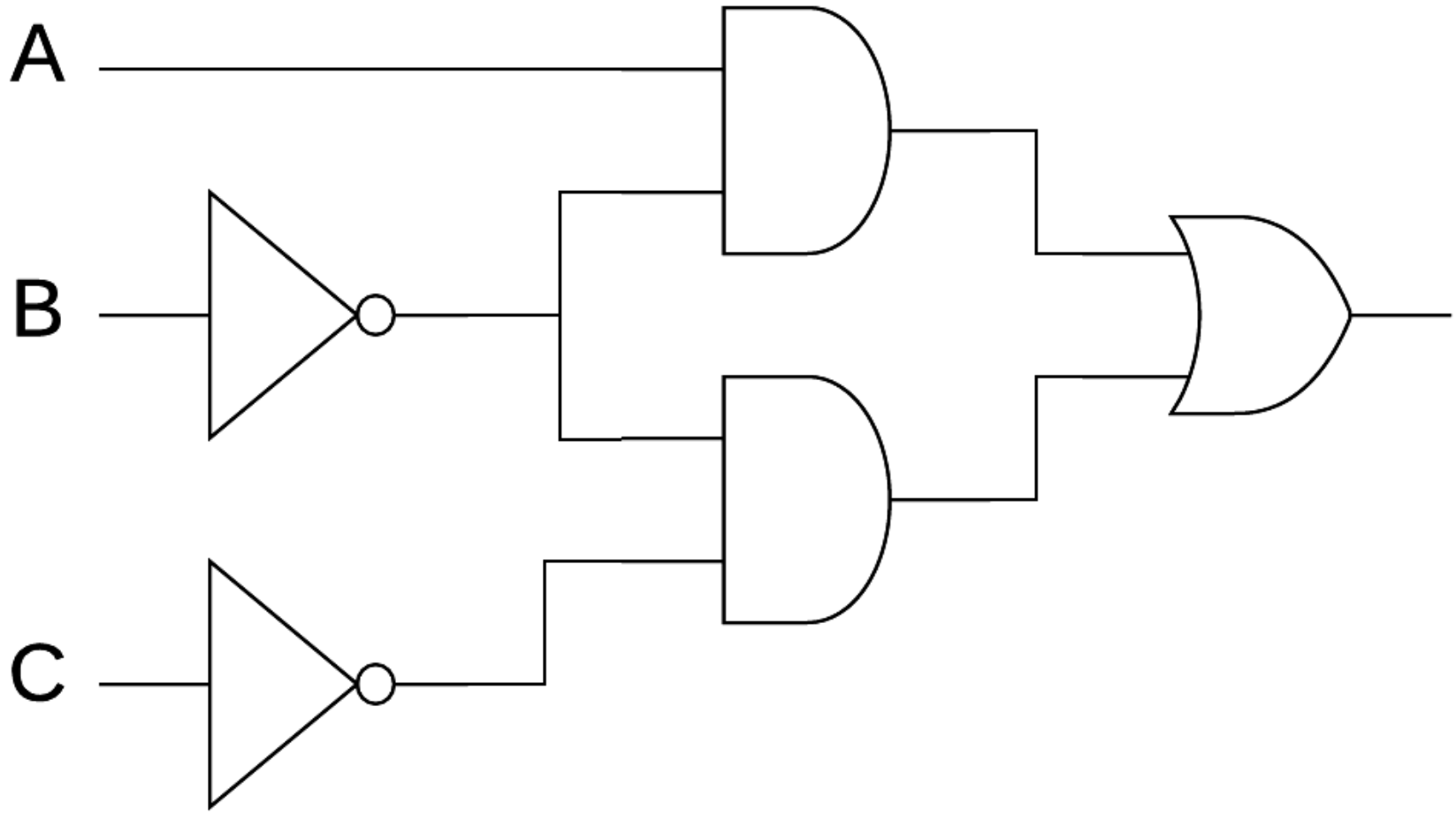
$$\begin{aligned}\overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C} + A\overline{B}C &= \overline{B}\overline{C}(\overline{A} + \overline{A}) + A\overline{B}C \\ &= \overline{B}\overline{C}(1) + A\overline{B}C \\ &= \overline{B}\overline{C} + A\overline{B}C\end{aligned}$$

Better simplification

$$\begin{aligned}\overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C} + A\overline{B}C &= \overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C} + A\overline{B}C + A\overline{B}\overline{C} \\ &= \overline{B}\overline{C}(\overline{A} + A) + A\overline{B}(C + \overline{C}) \\ &= \overline{B}\overline{C}(1) + A\overline{B}(1) \\ &= \overline{B}\overline{C} + A\overline{B}\end{aligned}$$

Basic Logic Circuits

Taking the statement from before $\overline{B}\overline{C} + A\overline{B}$ we can make a basic circuit



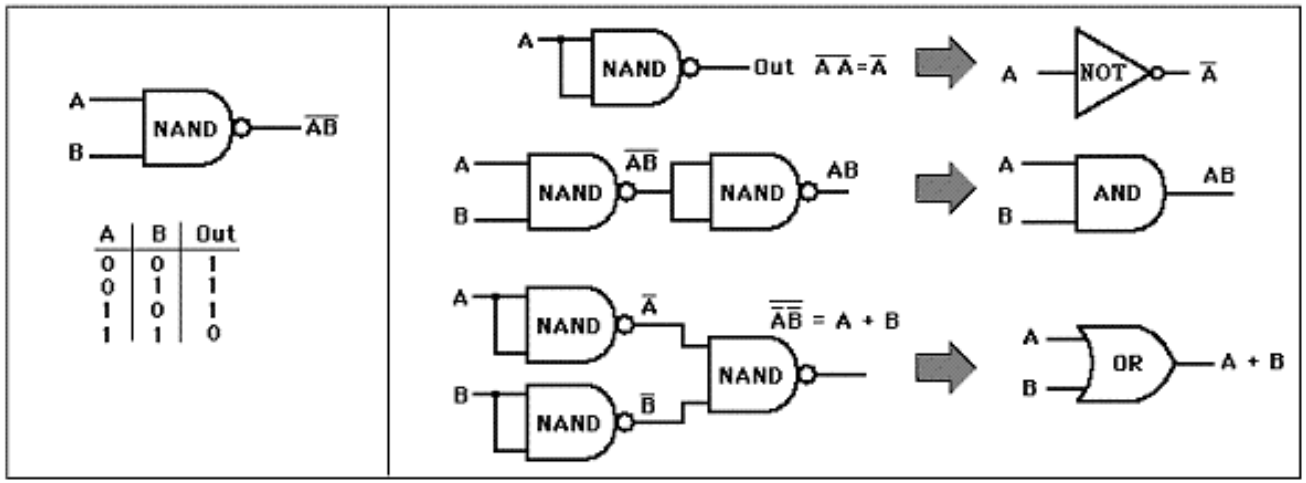
Karnaugh Maps

- Instead of using algebraic expansion we can use Karnaugh Maps to simplify statements from truth tables
- Each square in the map represents a *minterm*

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

NOR/NAND gates

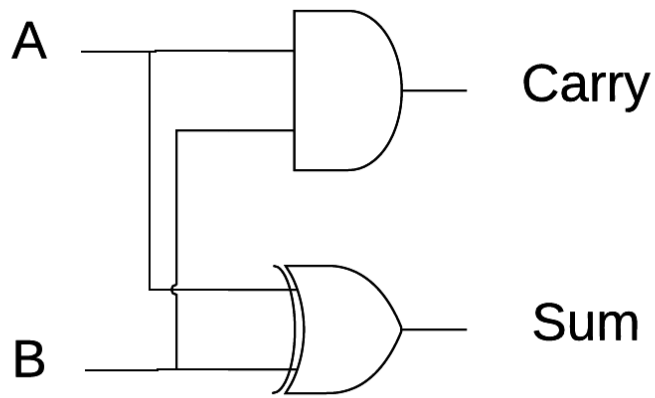
- NOR is an inverted OR gate
- NAND is an inverted AND gate
- Turns out every other logic gate can be implemented using entirely NOR or NAND gates
- Makes for interesting if uninspiring circuits
- Useful for implementing the sum of minterms from K-map
- See also https://pragprog.com/magazines/2012-03/the-nor-machine



Adding circuits

- 1-bit addition - half adder
- Two inputs, A and B are added together and a sum and carry are generated
- Can be built using an AND and XOR gate

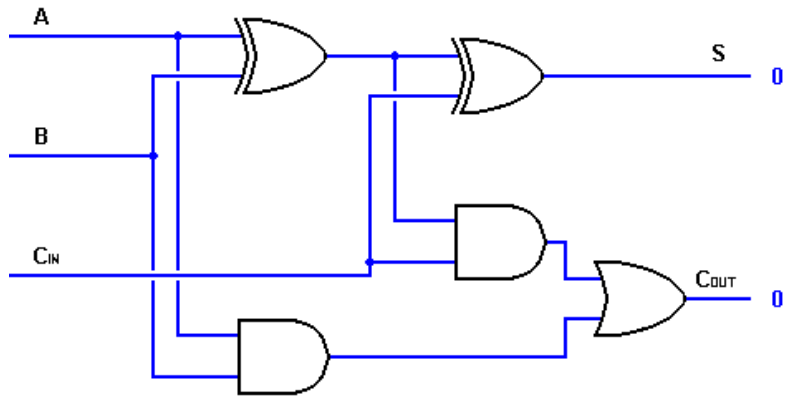
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Adding circuits

- 1-bit addition and carry in - full adder
- Three inputs, A, B and carry in are added together and a sum and carry out are generated
- Can be built using an AND and XOR gate

A	B	Carry in	Sum	Carry Out
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



Scaling up adders

- Modern computers are 32 or 64 bit processors
- Each integer represented by that many bits
- To add 64 bits, we need 64 full adders chained together
- Called a ripple-carry adder
- Introduces long delays as carry for bit 63 depends on bit 62 which depends on bit 61 etc

Faster adders

- Carry-lookahead adder
- Break the 64 adders into groups (say 16 4-bit blocks)
- Add circuitry to quickly determine if the block will generate a carry
- Then insert the carry into the next block
- Prefix Adder further complicates the carry generation circuit
- Reduces latency of transmission of carry to most significant bit
- As always a trade off between speed and power/complexity