CSC9R6 Computer Design

Practical Digital Logic

References

- (for this part of CSC9R6)
  - Hamacher et al: Computer Organization App A.
  - In library
    - Floyd: Digital Fundamentals Ch 1, 3-6, 8-10
      - web page: www.prenhall.com/floyd/
    - Mano: Computer System Architecture Ch 1 and 2
    - Green: Digital Electronics Ch 1, 3 - 11
- Tools
  - Digital Works
  - Electronics Workbench

What's it all about?

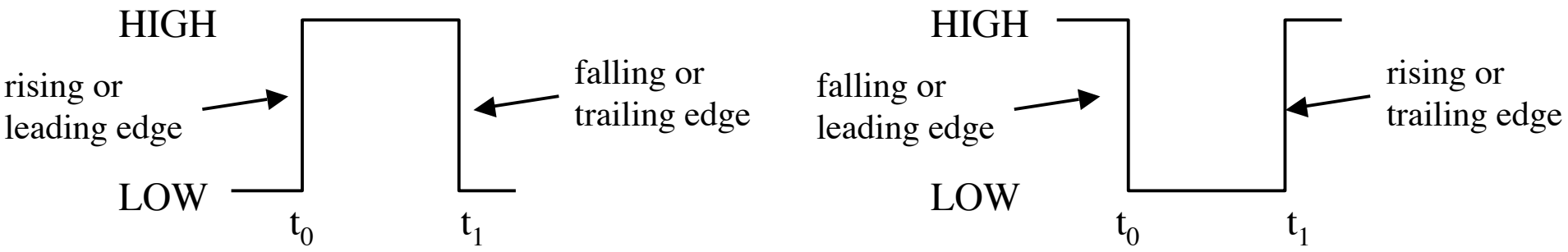
- Computers manipulate information
  - (already seen - Brookshear machine, programming languages, application programs)
- This part of the course looks at the hardware operations of a computer system
  - can mean CPU, I/0, RAM, devices … anything with a circuit or chip
- Specifically, we look at
  - computer organisation
  - the way basic hardware components work
  - the way they are connected together

Binary Information

- How is data represented inside a digital computer?
  - Using groups of digits (bits) in the binary number system (0 to 1 - very reliable)
- At an even lower level, binary information is physically represented
  - By electrical signals (logic levels HIGH and LOW) e.g. in TTL 2 - 5 volts = 1 and 0 - 0.8 volts = 0
- The digital circuits accept signals of this type and produce signals of this type.

Waveforms

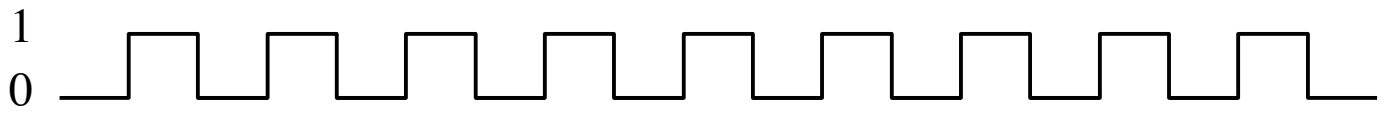
- Voltage levels constantly change back and forth from 0 to 1 during processing. This gives a **digital waveform.**
- A **Positive** Pulse is when a logic level switches from 0 to 1 and back to 0. (So a Negative pulse is …?)



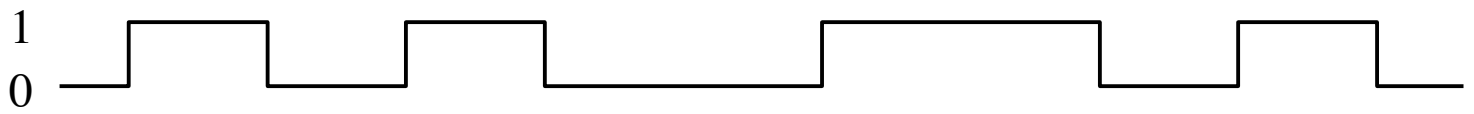
- Ideal pulses - change happens instantaneously.

Periodic Pulses

- Often in a digital system the waveform is a series of pulses.
- E.g. the clock



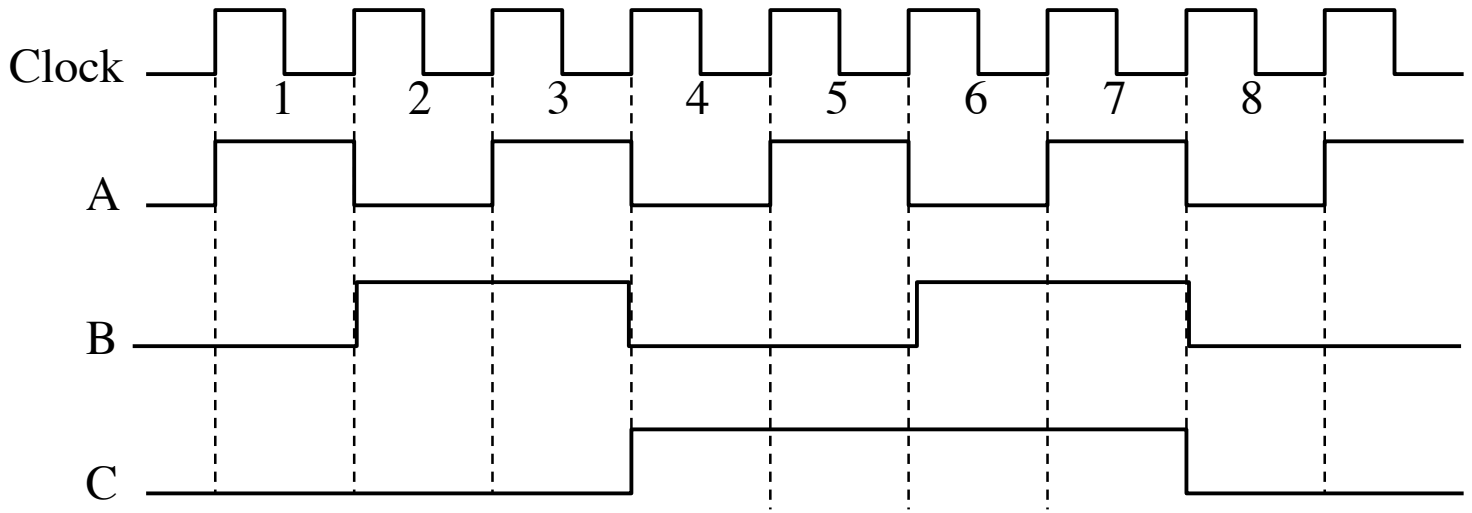
- A waveform carries information



- For example, 1010011010

Timing Diagrams

- A timing diagram is a graph of digital waveforms showing the relationship between the waveforms.



Gates (Logic Operations)

- Binary information is manipulated and processed through **gates**
  - Gates are the hardware embodiment of particular functions - for a given gate certain patterns of input signals produce certain patterns of output.

Gates

- Gates have several representations:
  - A graphic symbol (syntax really)
  - A name (more syntax)
  - A function (semantics)
  - Boolean algebra (familiar from 3111)
    - Algebraic expression (uses literals and logic ops)
  - A truth table (shows all possible inputs and output)
- We're going to see all of them! Each is important for different tasks.

AND

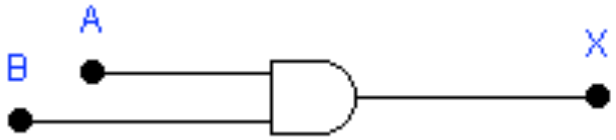
- Algebraically

  x = A.B

  or

  x = AB

- Pictorially



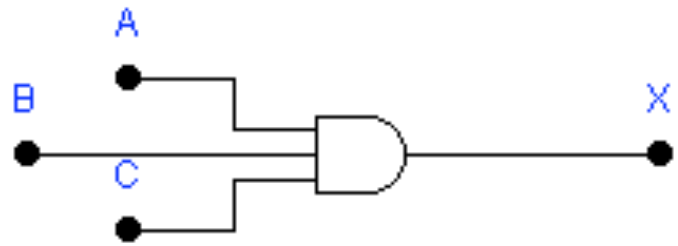
- Truth Table

A	B	x
0	0	0
0	1	0
1	0	0
1	1	1

  - i.e. true if both inputs are true
  - You can think of 1 (HIGH) as TRUE and 0 (LOW) as FALSE (because AND is just the familiar Boolean AND)

AND (there's more)

- AND can also be extended to more inputs
- x = A.B.C



- **Truth Table**

A	B	C	x
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

- Only outputs a 1 if all inputs are 1

OR

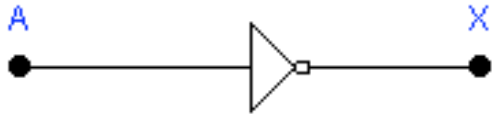
- x = A + B

- **Truth table**

A	B	x
0	0	0
0	1	1
1	0	1
1	1	1

NOT

- **x = A'**
- **also written x = $\overline{A}$**

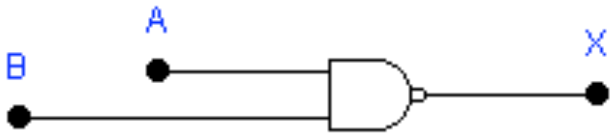


- Truth table

A	x
0	1
1	0

NAND (Not AND)

- **x = (A.B) '**



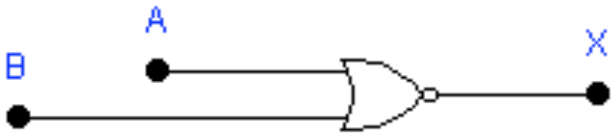
- Truth Table

A	B	x	[AND was]
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

- NAND is a *universal* gate

NOR (Not OR)

- **x = (A + B) '**



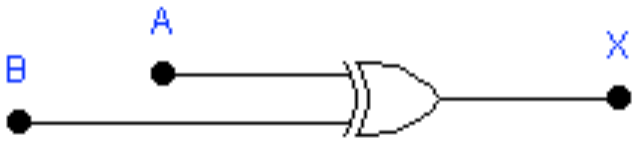
- Truth table

A	B	x	[OR was]
0	0	1	0
0	1	0	1
1	0	0	1
1	1	0	1

- NOR is a *universal* gate

XOR (Exclusive OR)

- **x = (A + B).(A.B) '**
  = A.B' + A'.B



- Truth Table

A	B	x
0	0	0
0	1	1
1	0	1
1	1	0

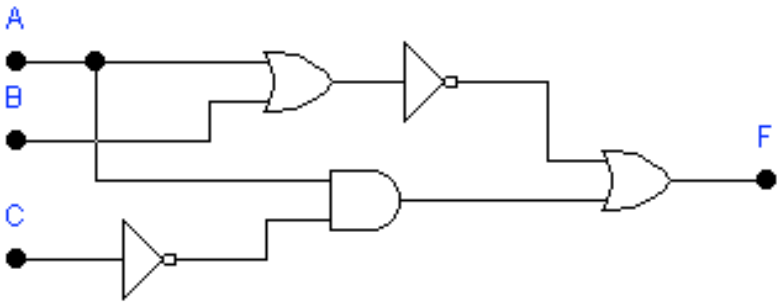
- either A or B but not both.

Putting it Together

- Using different notations also allows us to design in a convenient way (algebra) and translate to hardware, or to take a circuit and determine its function.

Example 1

- Start with a circuit



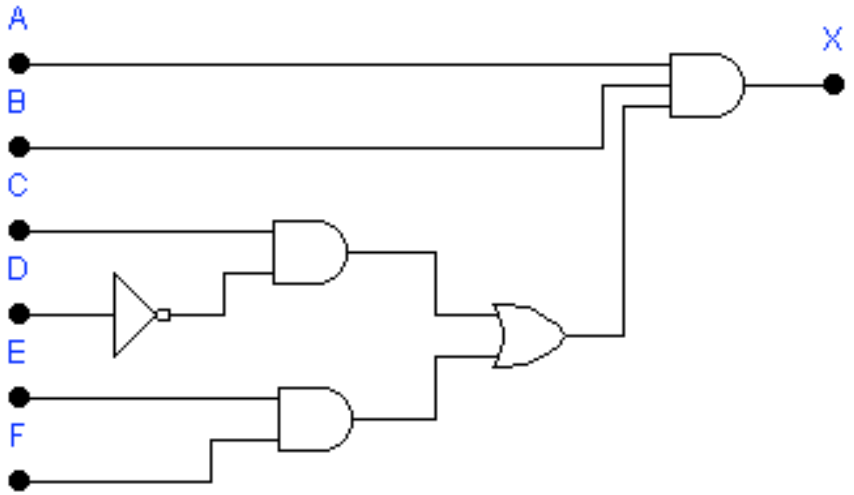
F = (A+B) ' + (A.C ')

A	B	C	(A+B) '	A.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			

Example 2

- Start with an expression

  X = A.B. (C.D ' + E.F)



Putting it together

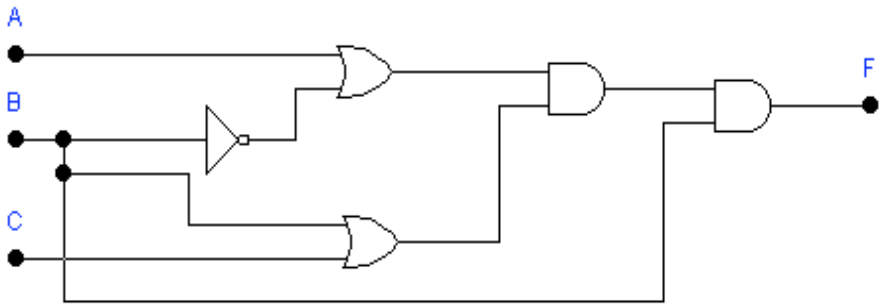
- From these simple components more complex circuits can be built to carry out essential functions. Eg.
  - comparison - comparator
  - arithmetic - adder, ALU
  - code conversion
  - encoding - encoder
  - decoding - decoder
  - data selection - multiplexer, demultiplexer
  - storage - flip-flops, registers
  - counting - counter
- Small and medium sized integration - circuits of < 100 gates

Boolean Algebra

- How to play around with the algebraic representation of a circuit.
- Why? - because it can lead us to better designs
  - efficient
  - cheaper
  - smaller

Example

- F = B.((A+B').(B+C))



- Needs 2 AND gates, 2 OR gates, 1 NOT gate, therefore is expensive.

- ***But*** F is also A.B

A	B	C	A+B'	B+C	(A+B').(B+C)	F	A.B
0	0	0					
0	0	1					
0	1	0					
0	1	1					
1	0	0					
1	0	1					
1	1	0					
1	1	1					

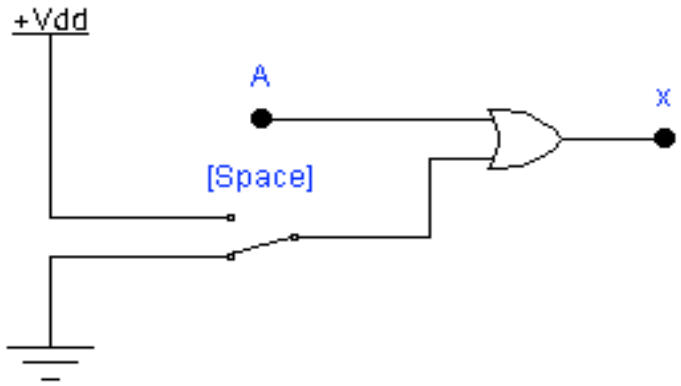
- We can show this using Boolean algebra (later)

Laws for OR

- Basic laws for manipulating expressions using OR (+)

OR1	A + 1 = 1
OR2	A + 0 = A
OR3	A + A = A
OR4	A + A' = 1

- Illustration of 1 and 2

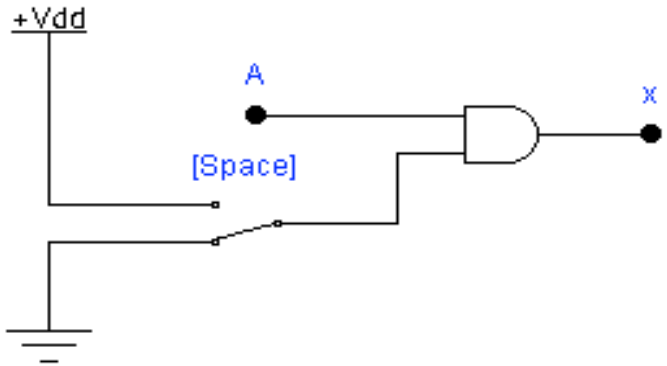


Laws for AND

- Basic laws for manipulating expressions using AND (.)

AND1	A.1 = A
AND2	A.0 = 0
AND3	A.A = A
AND4	A.A' = 0

- Illustration of 1 and 2



Laws for NOT

- Basic law for manipulating expressions using NOT (')

  NOT (A')' = A

Commutative laws

- It doesn't matter which order the inputs are in.

  AND A.B = B.A

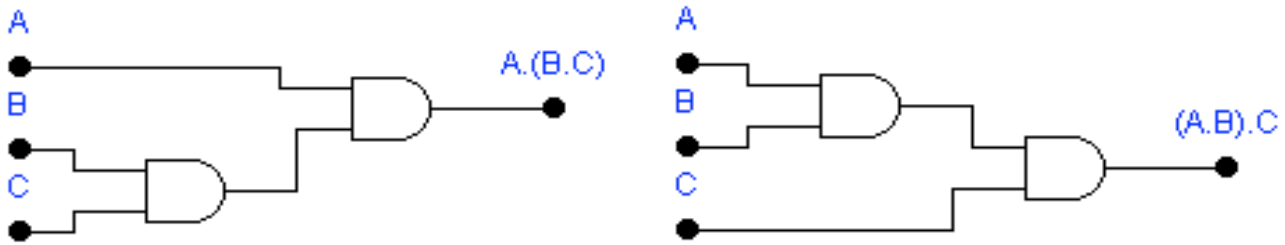
  OR A + B = B + A

Associative laws

- It doesn't matter where the brackets are.

  AND (A.B).C = A.(B.C)

  OR (A + B) + C = A + (B + C)



- So, often we leave out the brackets.
- Note: this only works if they're all the same operator!
  - e.g. (A.B) + C ≠ A.(B + C)

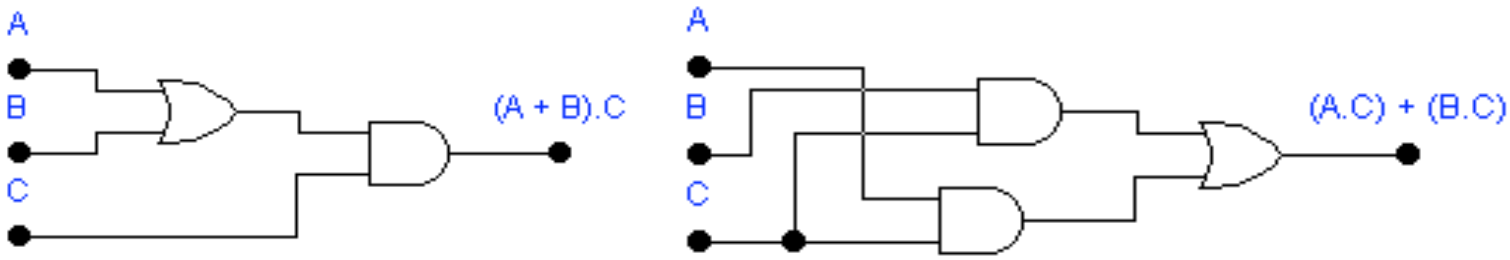
Distributive laws

- Changing the top level operator.

  DIST1 (A.B) + C = (A + C).(B + C)

  DIST2 (A + B).C = (A.C) + (B.C)

  DIST3 A.(B + C) = A.B + A.C



De Morgan's Laws

- **Very important**!!
- How to distribute a NOT inside a bracket.

  DM 1 $(A + B)' = A'.B'$

  DM 2 $(A.B)' = A' + B'$

- Why care about De Morgan's laws?
  - Because we can turn an AND into an OR and vice versa.
  - So if we only want one kind of gate (particularly NAND or NOR gates) in a circuit then De Morgan's laws help us derive that circuit.

Absorption Theorems

ABS1 A + (A.B) = A

ABS2 A.(A + B) = A

- You can also verify by truth table

A	B	A + B	A.(A + B)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

- Proof of ABS1

  … goes here …

- Proof of ABS2

  … goes here …

More absorption

- A + (A'.B) = A + B

  … proof goes here …

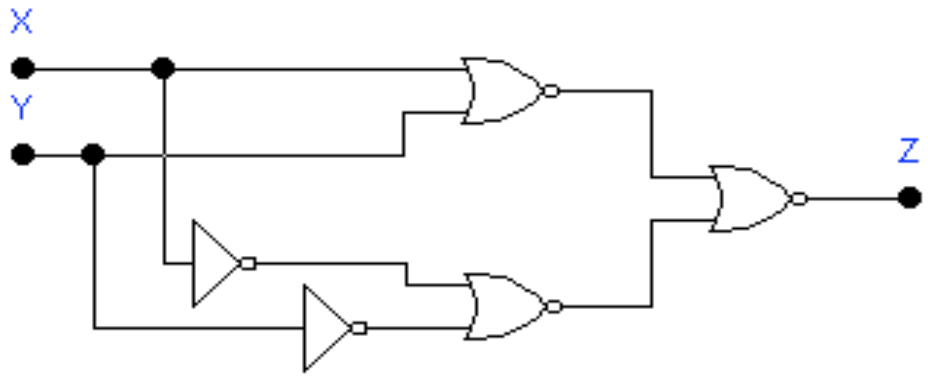
- A.(A' + B) = A.B

  … proof goes here …

Applying Boolean Algebra: Example

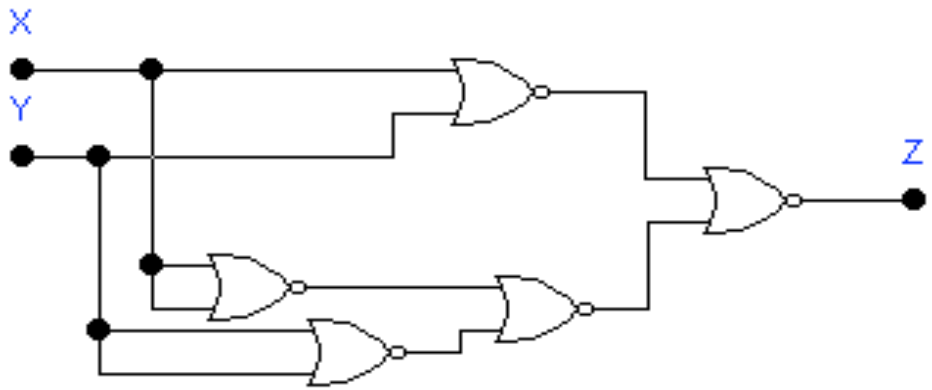
- Show (X + Y).(X' + Y') = ((X + Y)' + (X' + Y')')'
- To give a NOR gate implementation

Let	Z	= (X + Y).(X' + Y')	
then	(Z')'	= (((X + Y).(X' + Y'))')'	NOT
		= ((X + Y)' + (X' + Y')')'	DM2



Going even further!

- We can even remove the NOT gates.



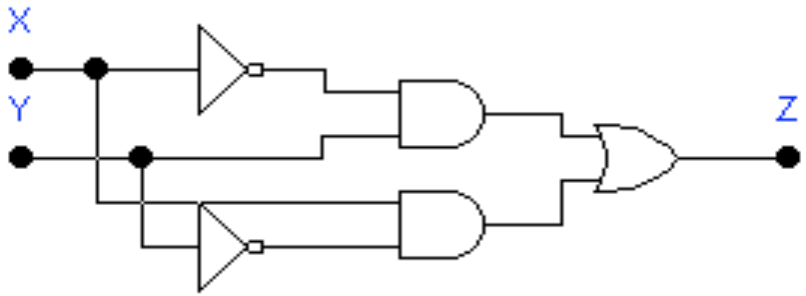
- making use of the equivalence A ' = (A + A) '

Example (cont.)

- An alternative derivation:

  Z= …

  = X'.Y + X.Y'



- Depends what kind of gates you want to use.

Complementation and De Morgan's laws

- Complement of A is A'; complement of A' is A (where A is any Boolean expression).
- Transformation using De Morgan's laws to obtain the complement of any expression.
- Substitute AND for OR and vice versa, and complement every literal.
- E.g.

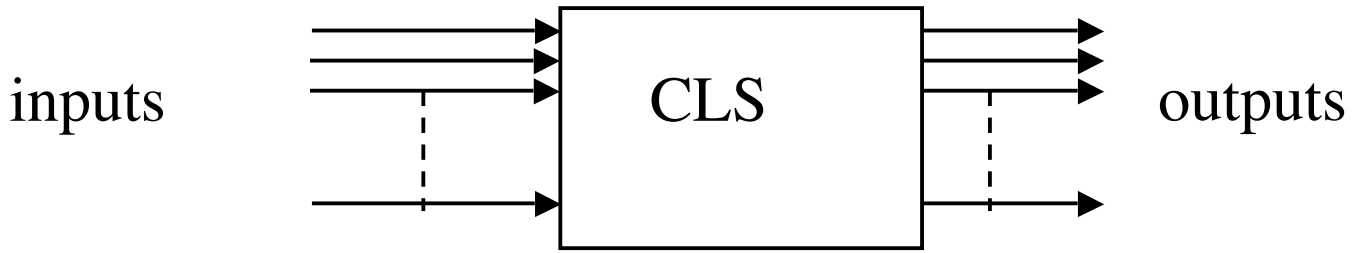
  (A.B)' = A' + B'

  (A + B) ' = A'.B'

- We'll come back to Boolean algebra again.

Combinational Logic Systems

- **Definition:** A Combination Logic System (CLS) is a binary digital 'black box' system with a set of *inputs* and a set of *outputs*. Its *outputs* depend only on its currents inputs (and not on the history of the system - there is no feedback)



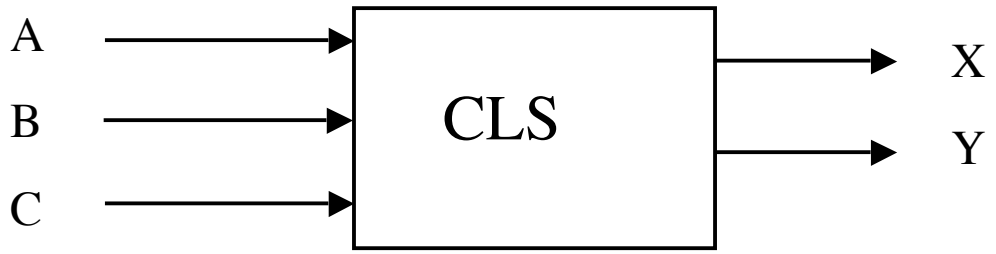
- A major part of digital electronic design is the construction of more complex CLSs using the fundamental gates as building blocks.

Description of a CLS

- The operation of a CLS is completely described by its input/output relationship. This describes the value of the output for each and every possible combination of the inputs to the system.
- Two ways to do this:
  - Truth table
  - Boolean algebra

Truth Table for a CLS

- Consider a CLS with N inputs and M outputs. There are 2^N possible combinations of inputs and therefore 2^N rows in the truth table. For each row M columns are required (one for each of the M outputs).
- **Example** 3 inputs (A, B, C), 2 outputs (X, Y).



- 2^3 rows = 8 rows
- 3 columns for inputs, 2 columns for outputs in each row.

Example (cont.)

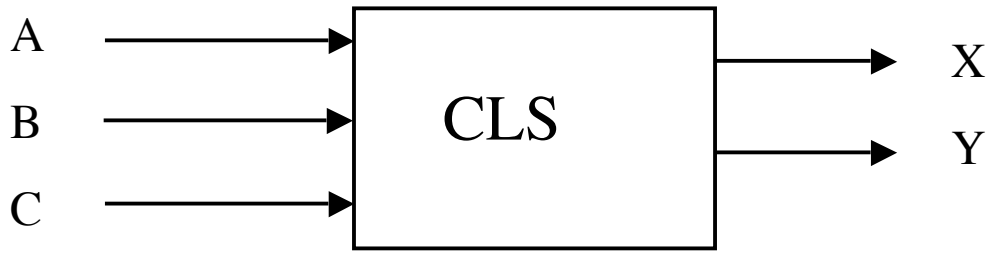
A	B	C	X	Y
0	0	0	1	1
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

X and Y outputs depend on the particular CLS being described. A value for each of X and Y is required for every possible combination of inputs.

- Consider more inputs - Tedious!!
- A more compact description of the CLS is obtained using Boolean algebra (fewer gates = lower cost).

Boolean Functional Representation of a CLS

- The operation of the CLS is given by the Boolean expression relating each output of the system to its inputs.
- **Example** (I/O as before)



- Let output X be $F_x = (A'.B')+(A'.C')+A.B.C$
- Let output Y be $F_y = (A.B.C)'$
- How do we move between the two representations?

Truth Table to Boolean Function

- Consider an N input, single output system.
- **Step 1:**
  - Identify the rows in the truth table that give 1 at the output.
- **Step 2:**
  - For each such row, write an expression containing all literals. Eg.

A	B	C	X	
⋮				
0	0	1	1	A'.B'.C
⋮				

- **Step 3:**
  - Combine each such expression with the others using '+'.
- This is **a** Boolean representation of the CLS.

Example

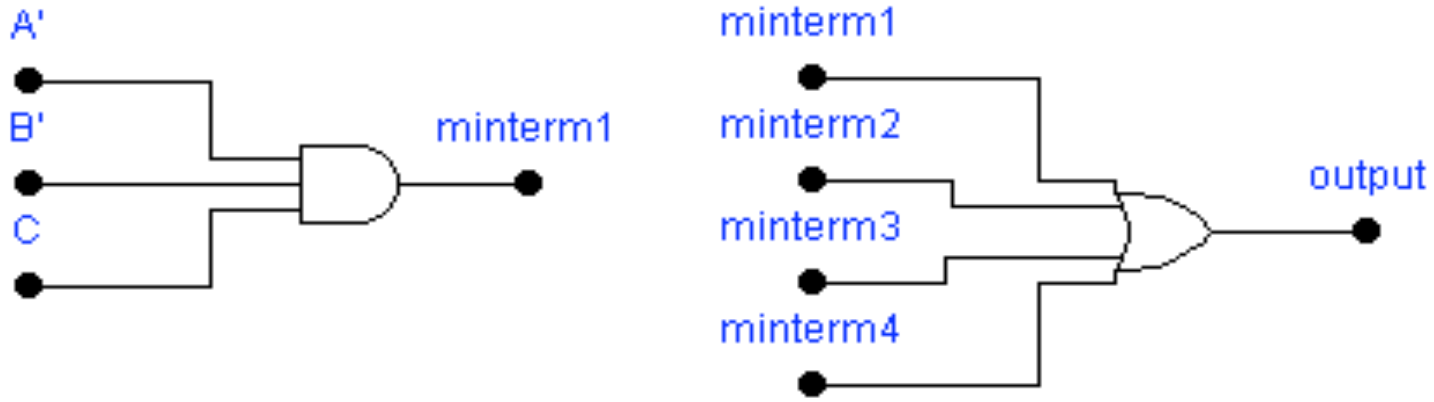
A	B	C	X
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

- F_x = ????

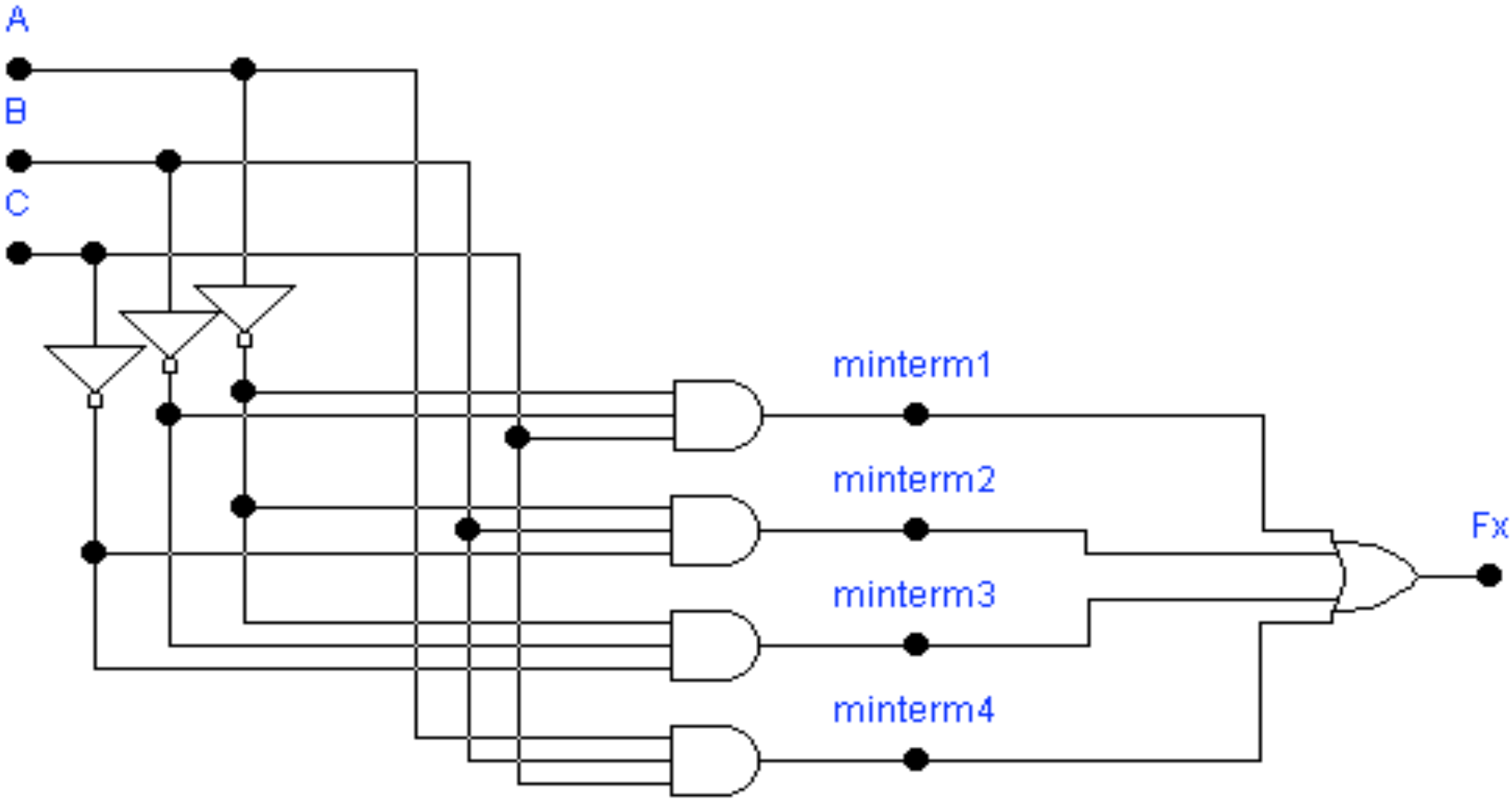
- This form of Boolean expression has a special name: **The canonical sum of products form (CSOP)**
- Because it is the 'OR' of a set of 'AND' terms, and each 'AND' term contains all of the input literals or their complements.
- Each 'AND' term is called a *minterm*

Implementation of CSOP

- The CSOP form suggests that the CLS is realised by a circuit consisting of a collection of 'AND' gates followed by a multi-input 'OR' gate. Each 'AND' gate implements a minterm, and the 'OR' gate gives the final output.
- E.g.



Example



Conversion to NAND only

- The CSOP form may be written as

  $F_x = M_1 + M_2 + M_3 + \ldots\ldots\ldots$ where M_i are the minterms.

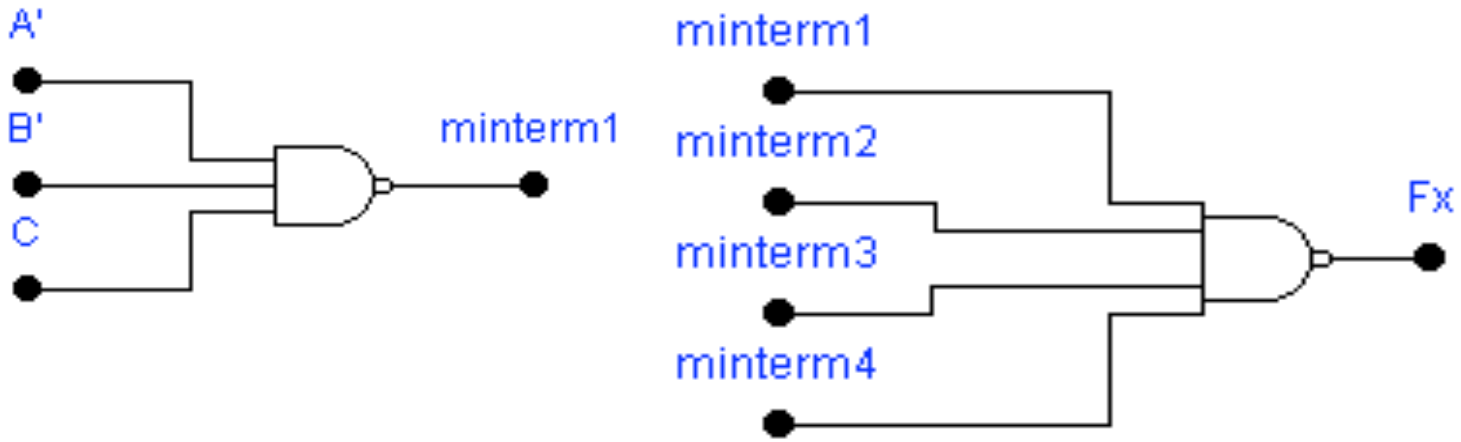
- Therefore

$(F_x)'$	$= (M_1 + M_2 + M_3 + \ldots\ldots\ldots)'$	
	$= M_1'.M_2'.M_3'. \ldots\ldots\ldots$	DM1
$((F_x)')'$	$= (M_1'.M_2'.M_3'. \ldots\ldots\ldots)'$	
F_x	$= (M_1'.M_2'.M_3'. \ldots\ldots\ldots)'$	NOT

- i.e. NAND representation (instead of using the OR gate of the previous implementation)
- Not only that, but each minterm is also implemented by a NAND gate

Example

- E.g. let minterm M_I = A'.B'.C
- Then we have



Design of a CLS

- So design of a CLS has 5 steps:
  1. Statement of the problem
  2. Assign I/O variables to letter symbols
  3. Derive a truth table defining the relationship between inputs and outputs
  4. Derive the Boolean functions for each output
  5. Draw the circuit diagram

Example: Half Adder

- Basic function - add two bits together: (weird name because in full adder you also consider the carry - so add 3 bits together)
- Variables
  - Input: A, B
  - Output: S(sum), C(carry)
- Truth Table

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

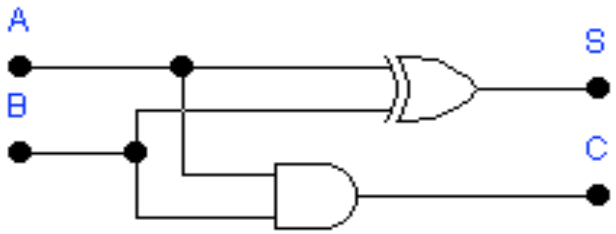
Half Adder (2)

- Boolean functions for S and C (in CSOP)

  S = A'.B + A.B'

  C = A.B

- Desired circuit is



Half Adder (3)

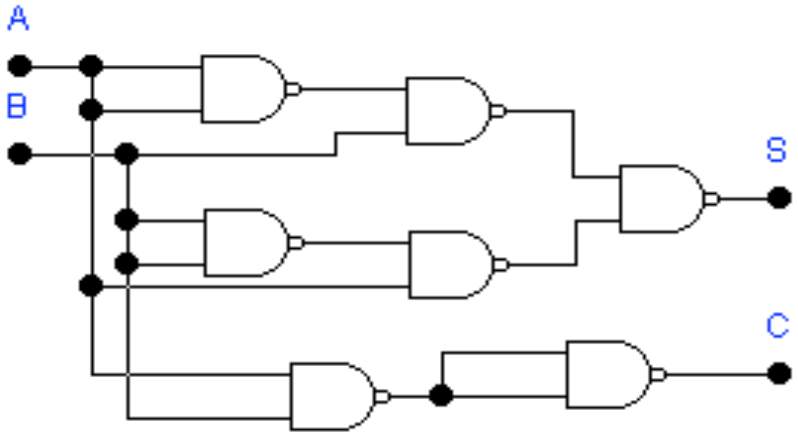
- Could also convert to NAND form

  S' = (A'.B + A.B')' C = ((A.B)')'

  = (A'.B)'.(A.B')'

  S = (S')' = ((A'.B)'.(A.B')')'

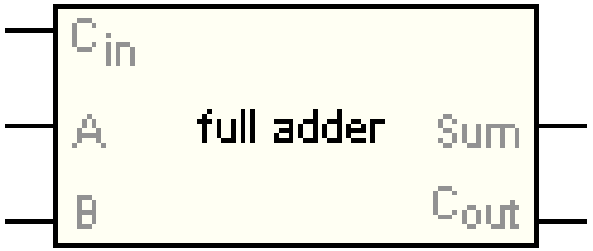
- To implement complement using NAND, note A' = (A.A)'



Half Adder and Full Adder

- These components are available as macros in DW.
  - Schematic is

  - Called a half adder because doesn't perform full adding function, but ok for LSB - no carry-in.
- Full Adder
  - Required for other than LSB



Boolean expression to truth table

- Given a function F_x , first expand to CSOP form

F_x = (A+B).C'

= A.C' + B.C'

DISTR

= (A.C'.1) + (B.C'.1)

AND1

= (A.C'.(B +B')) + (B.C'.(A + A'))

OR4

= A.B.C' + A.B'.C'. + A.B.C' + A'.B.C'

DISTR

= A.B.C' + A.B'.C'. + A'.B.C'

OR3

- Then mark a 1 in the appropriate row.

A	B	C	X
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0