Imperial College London

Lecture 11

Digital Logic, Boolean Algebra, Flip-Flops

Professor Peter YK Cheung
Dyson School of Design Engineering

URL: www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/
E-mail: p.cheung@imperial.ac.uk

Basic Boolean Operators & Logic Gates

- Inverter (NOT Gate)
- AND Gate
- OR Gate
- Exclusive-OR (XOR) Gate

- NAND Gate = AND Gate + Inverter
- NOR Gate = OR Gate + Inverter
- Exclusive-NOR Gate = XOR Gate + Inverter

The basic logic gates are the inverter (or NOT gate), the AND gate, the OR gate and the exclusive-OR gate (XOR). If you put an inverter in front of the AND gate, you get the NAND gate etc.

Truth Tables

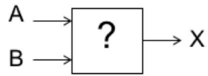
- Truth Tables specifies how a logic circuit's output depends on the logic levels present at the inputs.

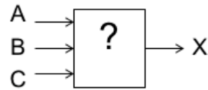
Inputs / Output

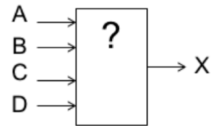
A	B	x
0	0	1
0	1	0
1	0	1
1	1	0

A	B	C	x
0	0	0	0
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

A	B	C	D	x
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1





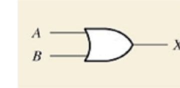


One of the common tool in specifying a gate function is the truth table. All possible combination of the inputs A, B … etc, are enumerated, one row for each possible combination. Then a column is used to show the corresponding output value.

If two ligic circuits share identical truth table, they are functionally equivalent.

Shown here are example of truth tables for logic gate with 2, 3 and 4 inputs.

Different representations – OR gate



Distinctive shape symbol

Rectangular outline symbol

Symbol or Schematic

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

Truth table

0 = LOW
1 = HIGH

Truth Table & Boolean Expression

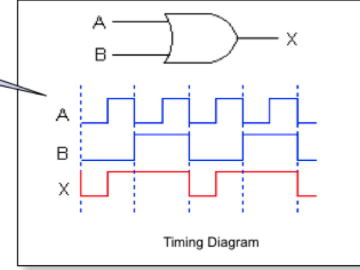
X = A + B

Boolean expression

Boolean: A or B
Multiple bits: A | B

Python

Timing Diagram



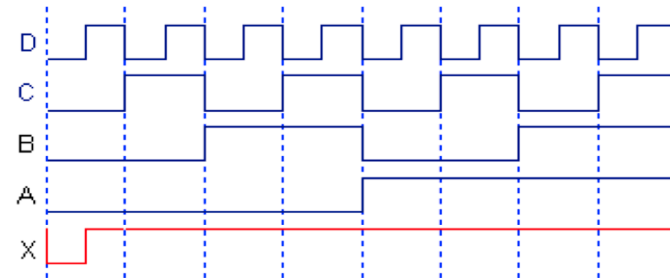
The output of an OR gate is HIGH whenever one or more inputs are HIGH

Here we show five different representation of the OR gate or OR function. They are:

1. Sch … gic symbol
2. Tru
3. Boo
4. Tim
5. Exp … g language (e.g. Python)

A
B
C
D
X

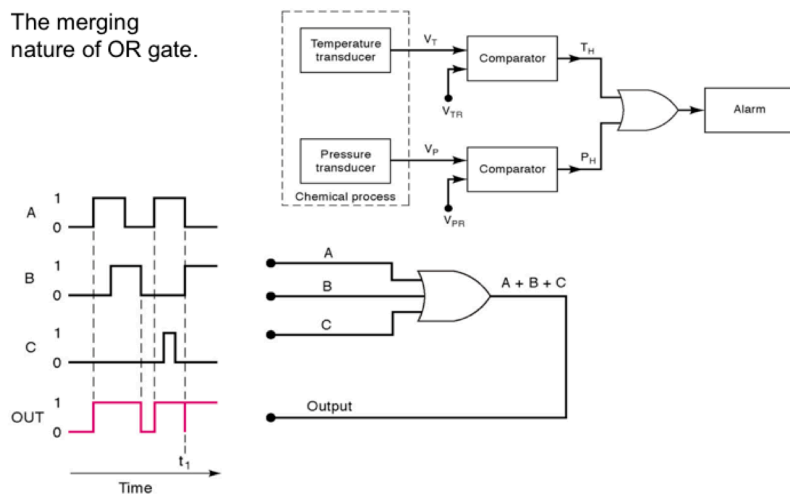
X = A+B+C+D



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

In summary, OR operation produces as result of 1 whenever any input is 1. Otherwise 0.
An OR gate is a logic circuit that performs an OR operation on the circuit's input.
The expression x=A+B is read as "**x equals A OR B**"

Two more examples of OR function



Self check questions:

What is the only set of input conditions that will produce a LOW output for any OR gate?

Answer: all inputs LOW

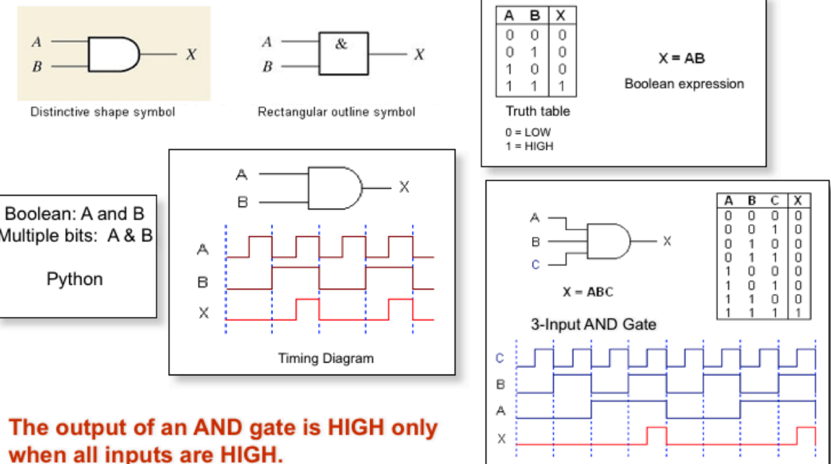
Write the Boolean expression for a six-input OR gate.

Answer: X=A+B+C+D+E+F

If the A input in example shown above is permanently kept at the 1 level, what will the resultant output waveform be?

Answer: constant HIGH

The AND Operation & AND Gate



The AND operation is performed the same as ordinary multiplication of 1s and 0s.

An AND gate is a logic circuit that performs the AND operation on the circuit's inputs.

An AND gate output will be 1 **only** for the case when **all** inputs are 1; for all other cases the output will be 0.

The expression **x=A•B** is read as "x equals A AND B."

Self-check questions:

What is the only input combination that will produce a HIGH at the output of a five-input AND gate?
Answer: all 5 inputs = 1

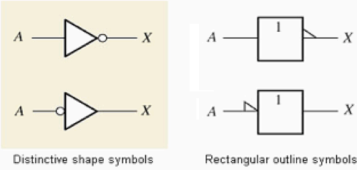
What logic level should be applied to the second input of a two-input AND gate if the logic signal at the first input is to be inhibited(prevented) from reaching the output?

Answer: A LOW input will keep the output LOW

True or false: An AND gate output will always differ from an OR?

Anaswer: False

The NOT Operation & Inverter



Distinctive shape symbols

Rectangular outline symbols

A	X
0	1
1	0

Truth table

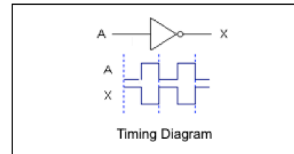
0 = LOW
1 = HIGH

$X = \overline{A}$

Boolean expression

Boolean: not A
Multiple bits: ~A

Python



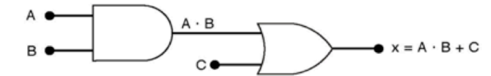
Timing Diagram

The output of an inverter is always the complement (opposite) of the input.

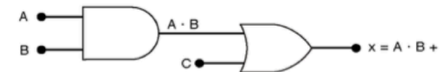
The NOT gate (inverter) is simple, but important. Note the difference between a Boolean operator "not A", where A is a Boolean variable (i.e. True or False), and that for a multiple bit variable. In multiple bit case, ~A results in EACH BIT within A being inverted. This is also known as "bitwise" operation.

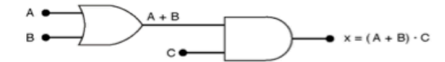
Describing logic circuits algebraically

- Any logic circuit, no matter how complex, can be completely described using the three basic Boolean operations: OR, AND, NOT.
- Example: logic circuit with its Boolean expression



- How to interpret A•B+C?
  - Is it A•B ORed with C ? Is it A ANDed with B+C ?
- Order of precedence for Boolean algebra: AND before OR. Parentheses make the expression clearer, but they are not needed for the case on the preceding slide.
- Therefore the two cases of interpretations are :



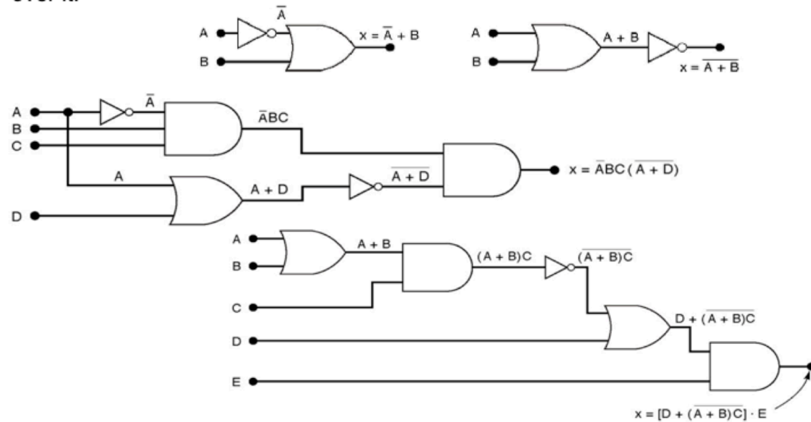


Using symbolic diagram or truth table to specify or describe logic gates and logic functions is cumbersome. A much better way is to use algebraic expression. Here a "dot" represents the AND operation, and a "+" represents and OR operation. Furhtermore, a bar over a variable or a '/' in front of the variable represents an inversion (NOT function).

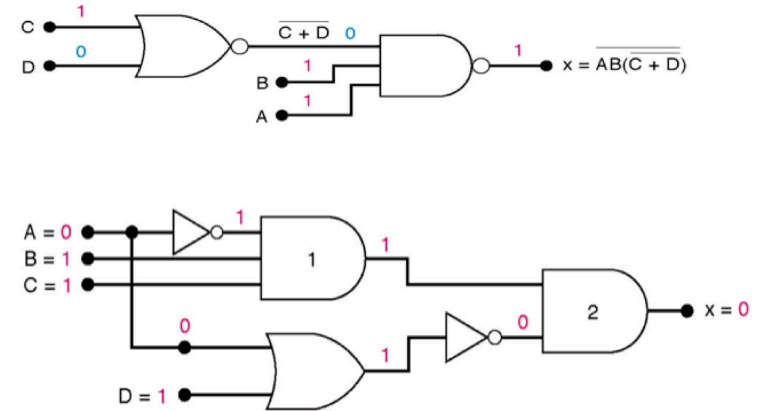
The convention is that AND has precedence over OR.

Precedence rules in Boolean algebra:

1. First, perform all inversions of single terms
2. Perform all operations with parentheses
3. Perform an AND operation before an OR operation unless parentheses indicate otherwise
4. If an expression has a bar over it, perform the operations inside the expression first and then invert the result



Inversion in Boolean expression has a bar over the Boolean variable. Here are a number of examples.

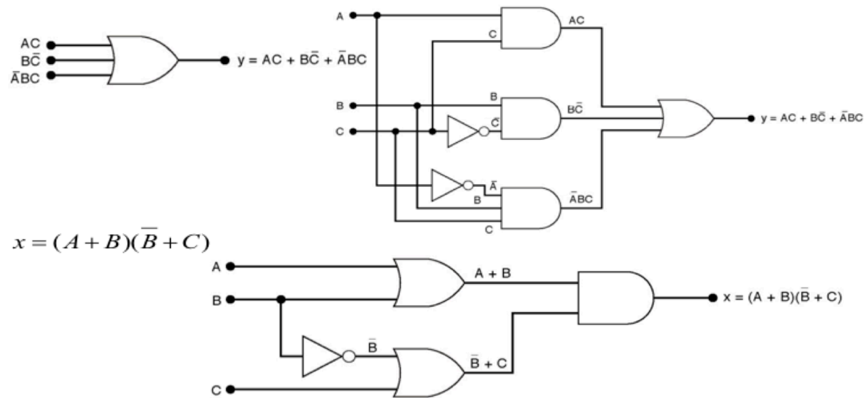


We often do not draw the full inverter, but use a circle to indicate inversion. Therefore shown here on the top circuit, there is a 2-input OR gate followed by an inverter, making it a NOR gate.

The evaluate the output of this circuit for inputs shown, we propagate the input values through the gates from left to right.

Implementing Circuits From Boolean Expressions

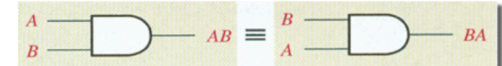
- When the operation of a circuit is defined by a Boolean expression, we can *draw a logic-circuit* diagram directly from that expression.



Laws of Boolean Algebra

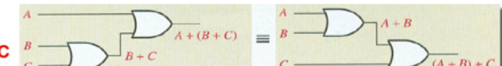
- Commutative Laws

A + B = B + A

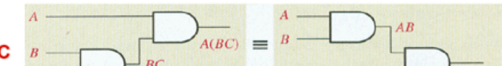


A • B = B • A

- Associative Laws

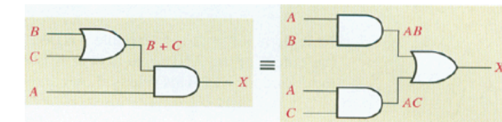


A + (B + C) = (A + B) + C



A • (B • C) = (A • B) • C

- Distributive Law



A • (B + C) = A • B + A • C
A (B + C) = A B + A C

Given a Boolean expression, we can easily translate it to symbolic representation of gates. This is quite easy to do.

Just like normal algebra, Boolean algebra allows us to manipulate the logic equation and perform transformation and simplification.

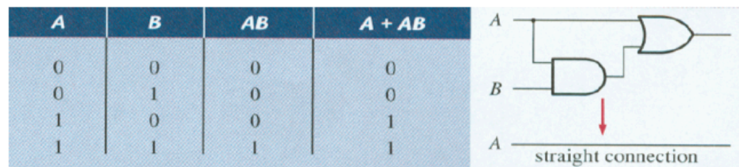
Boolean algebra obeys the same laws as normal algebra:

1. the commutative law – the order of the Boolean **variables** do not matter
2. the associative law – the order of the Boolean **operators** do not matter
3. The distributive law –one can distribute a Boolean operator into the parenthesis

Rules of Boolean Algebra

1. $A + 0 = A$
2. $A + 1 = 1$
3. $A \cdot 0 = 0$
4. $A \cdot 1 = A$
5. $A + A = A$
6. $A + \overline{A} = 1$
7. $A \cdot A = A$
8. $A \cdot \overline{A} = 0$
9. $\overline{\overline{A}} = A$
10. $A + AB = A$
11. $A + \overline{A}B = A + B$
12. $(A + B)(A + C) = A + BC$

- Rules 1 to 9 are obvious.
- Rule 10: A + AB = A



A	B	AB	A + AB
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

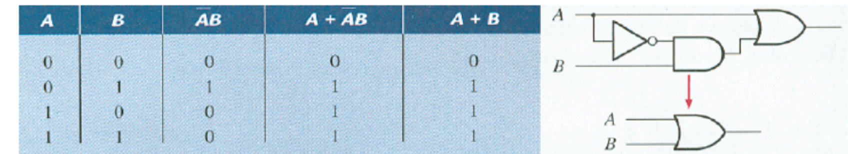
There are also a number of rules to help simplification of Boolean expression.

The first 9 rules listed here are obvious.

Rule 10: Less obvious, but it is clearly shown here that it is true.

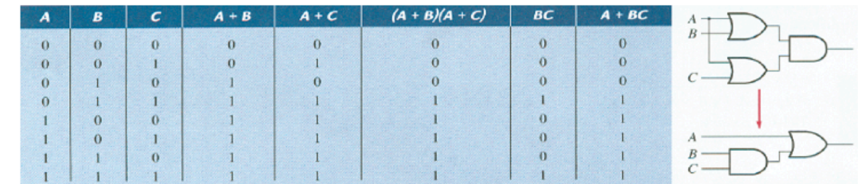
Rules 11 and 12 of Boolean Algebra

- Rule 11: $A + \overline{A}B = A + B$



A	B	$\overline{A}B$	$A + \overline{A}B$	A + B
0	0	0	0	0
0	1	1	1	1
1	0	0	1	1
1	1	0	1	1

- Rule 12: (A + B)(A + C) = A + BC



A	B	C	A + B	A + C	(A + B)(A + C)	BC	A + BC
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	1	1	1	1
1	0	0	1	1	1	0	1
1	0	1	1	1	1	0	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Rule 11 and Rule 12 are more difficult. You may need to remember them in order to apply them for the purpose of simplifying Boolean expressions.

Using Boolean Algebra to simplify expressions

$y = A\overline{B}D + A\overline{B}\,\overline{D}$ → $y = A\overline{B}$

$z = (\overline{A} + B)(A + B)$ → $z = B$

$x = ACD + \overline{A}BCD$ → $x = ACD + BCD$

Here are three examples of simplification of Boolean expressions

DeMorgan's Theorems

- Theorem 1 $\overline{(x + y)} = \overline{x} \cdot \overline{y}$
- Theorem 2 $\overline{(x \cdot y)} = \overline{x} + \overline{y}$

Remember:

"Break the bar, change the operator"

- DeMorgan's theorem is very useful in digital circuit design
- It allows ANDs to be exchanged with ORs by using invertors
- DeMorgan's Theorem can be extended to any number of variables.

$F = \overline{X.Y} + \overline{P.Q}$ ← 2 NAND plus 1 OR

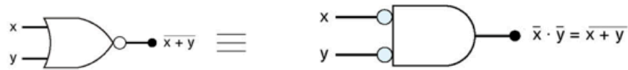
$= \overline{X} + \overline{Y} + \overline{P} + \overline{Q}$ ← 1 OR with some input invertors

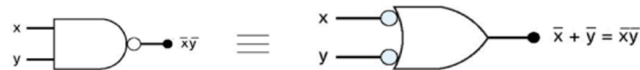
$z = \overline{(\overline{A} + C) \cdot (B + \overline{D})}$ → $z = A\overline{C} + \overline{B}D$

De'Morgan's Theorems is important to Boolean logic. They allow us to exchange OR operation with AND operation and vice versa.

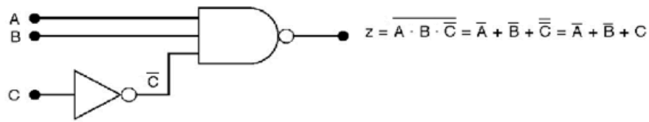
Applying De'Morgan, we can also simplify Boolean expression in many cases.

Implications of DeMorgan's Theorems(I)





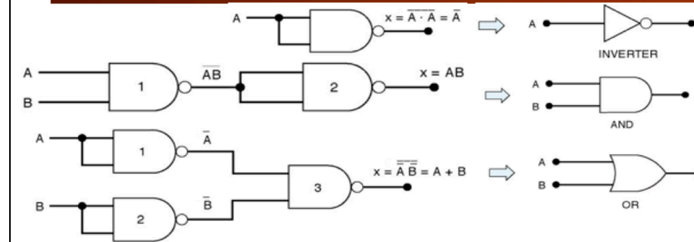
- Determine the output expression for the circuit below and simplify it using DeMorgan's Theorem

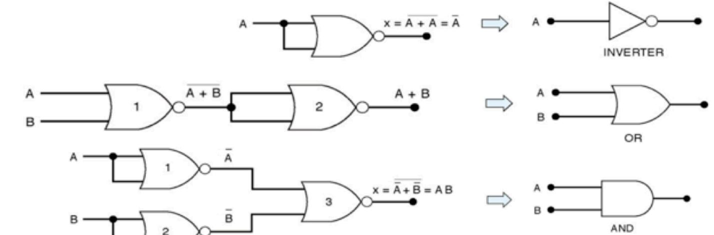


De'Morgan requires an inverter output.

Here is symbolic representation of De'Morgan: move the inversion to the inputs, and change OR to AND, or AND to OR.

Universality of NAND and NOR gates



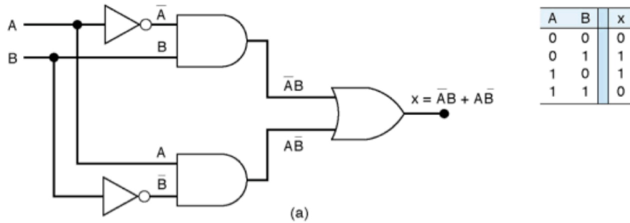


Let us assume that we ONLY have 2-input NAND gate. From this ,we can get an inverter, an AND gate, and, thanks to De'Morgan, we can also get an OR gate. In other words, if we have a 2-input NAND gate, we can build the three basic logic operators: NOT, AND and OR. As a result, we can build ANY logic circuit and implement any Boolean expression. Taken to limit, give me as many NAND gate as I want, in theory I can build a Pentium processor. This shows the universality of the NAND gate.

Similarly, one can do the same for NOR gates.

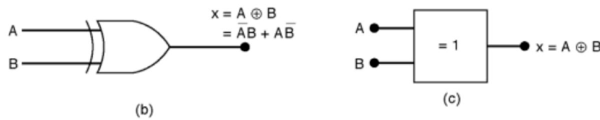
Exclusive-OR

- Exclusive-OR (XOR) produces a HIGH output whenever the two inputs are at opposite levels.



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

XOR gate symbols

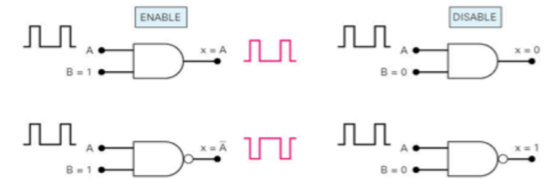


A, B → [= 1] → $x = A \oplus B$

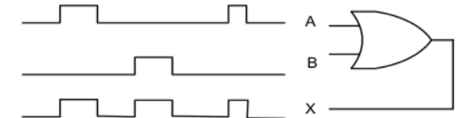
(c)

Functional view of Gates

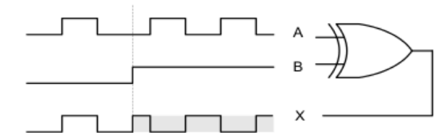
- AND gate function act as enable/disable circuits:-



- OR gate performs signal merging function:-



- XOR gate performs selectable inversion function:-



The exclusive-OR gate is high only if the two inputs are DIFFERENT, i.e. either A is high OR B is high but not both.

The XOR gate is often used as logic comparator (output is 0 if the two inputs are equal).

Now let us take a functional view of logic operators.

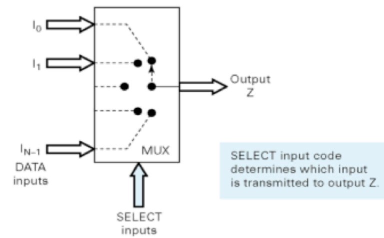
The AND operator can be interpreted as having an enabling or disabling function. (We sometimes call this a 'gating function" as if it perform a gate keeping (i.e blocking) function.) Input B here is a the gating control – if B = 1, it lets A through, otherwise if B = 0, it blocks A.

The OR operator can be interpreted as a merging function. It combines both A and B high level and merge them to form output X.

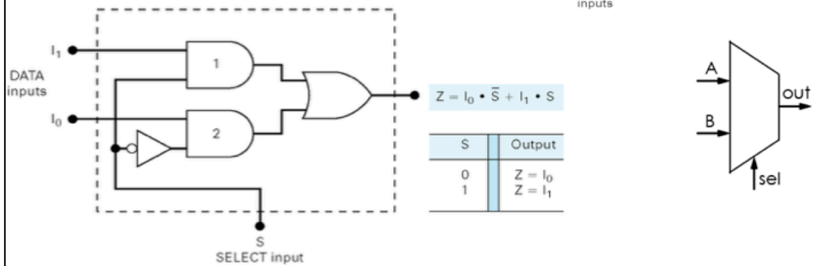
The XOR gate can be viewed as a selectable inverter. If B = 0, A is pass to the output. If B = 1, A is inverted. So B determines inversion, or no inversion of A.

Multiplexers (Data Selectors)

- Multiplexer (MUX) selects one of many input to send to output



SELECT input code determines which input is transmitted to output Z.



$Z = I_0 \cdot \bar{S} + I_1 \cdot S$

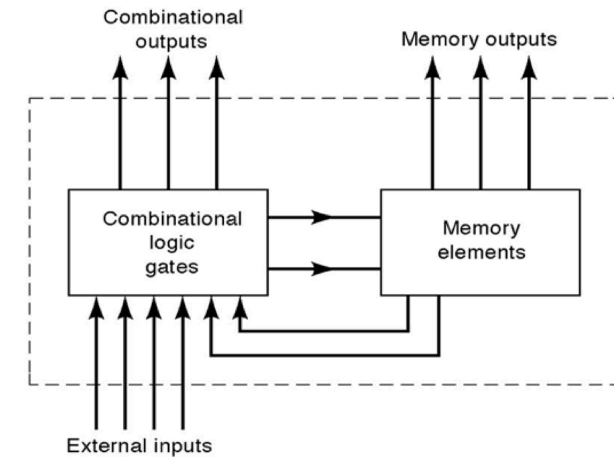
S	Output
0	$Z = I_0$
1	$Z = I_1$

A
B
out
sel

The multiplexer is another useful gate function. It has a number of data inputs, a number of select inputs, and an output. The select inputs determines WHICH of the data input is sent to the output. Shown here is a 2-to-1 multiplexer with inputs I0 and I1. If S is 0, Z = I0, if S is 1, Z= I1.

One can combine a number of 2-to-1 multiplexer to form larger multiplexers. For example, if we use THREE 2-1 MUX, we can build a 4-to-1 multiplexer.

General digital system diagram



Combinational logic gates evaluate Boolean expressions. They can do computation, decoding (e.g. mapping of one binary number to another binary number), selection (such as multiplexers and de-multiplexers) and any function that can be expressed in Boolean expression form. One key characteristic of combinational logic is that outputs completely determined by the input values at a given time. As soon as input changes, soon afterwards, the output will change if necessary. Since the logic gates themselves have delay, the change may happen with some delay.

In other words, combinational logic gates DO NOT HAVE MEMORY (or storage). Its outputs only depend on current inputs and not previous inputs.

Another class of digital circuits, which can be built with gates, have memory. The currently output depends on previous inputs as well as current inputs. These can be built from NAND (or NOR) gates alone. The "memory" property comes from feedback – feeding the outputs of gates back to the inputs.

On this course, we will NOT concern ourselves with how to implement such a storage components using NAND gates. We will just use memory circuits as a building block.

Any digital circuit, no matter how complex, can be model by the combination of the combinational circuit connected with some memory circuits as shown here. This model may look simple, but it is universal!

Properties of Sequential Circuits

- So far we have seen **combinational logic**
  - the output(s) depends only on the current values of the input variables
- Here we will look at **sequential logic** circuits
  - the output(s) can depend on present and also past values of the input and the output variables
- Sequential circuits exist in one of a defined number of states at any one time
  - they move "sequentially" through a defined sequence of transitions from one state to the next
  - The output variables are used to describe the state of a sequential circuit either directly or by deriving state variables from them

Circuits whose outputs depends on current inputs and past history are called sequential circuits. This is because the circuit will follow some sequences, hence the past is taken into account.

In sequential circuits, the output could take on different values. These are known as states. A sequential circuit will go through these different states in a certain sequence. The exact sequence depends on what happens to the input.

Adding memory into a sequential circuit adds the time domain into the mix. Straight forward Boolean expression is no longer sufficient because a Boolean variable has no notion of time. Therefore when we use Boolean equations to describe behaviour of sequential circuits, we need to somehow introduce the time element.

When a sequential circuit goes from one state (as defined by some of its output) to another state, we call this a "state transition".

Synchronous and Asynchronous Sequential Logic

- Synchronous
  - the timing of all state transitions is controlled by a common clock signal
  - changes in all variables occur simultaneously
- Asynchronous
  - state transitions occur independently of any clock and normally dependent on the timing of transitions in the input variables
  - changes in more than one output do not necessarily occur simultaneously
- Clock
  - A clock signal is a square wave of fixed frequency
  - Often, transitions will occur on one of the edges of clock pulses
    - i.e. the rising edge or the falling edge

In sequential circuits, if the transition from a state to another state is governed by a single regular, repetitive, clock signal, we call this a **synchronous circuit**. All changes in such circuits are synchronous (and governed) by the clock signal. Almost all digital circuits in the real-world are synchronous. For example, when you buy a computer with an Intel processor running at 3GHz, this number refers to the clock signal frequency that determines how the processor perform its Boolean evaluation in the logic gates on the chip.

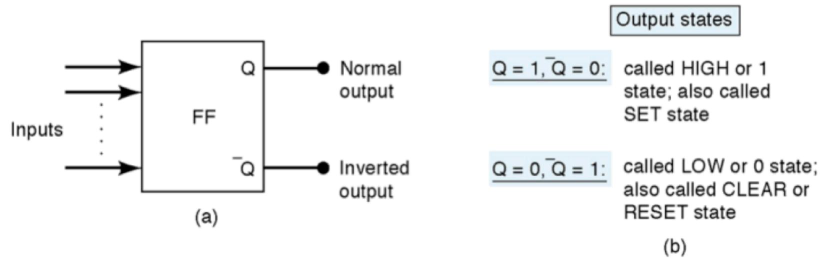
Occasionally, some digital circuits goes from one state to another state NOT governed by a clock signal. These are called **asynchronous circuits**.

In theory, we can implement any logic function in either the synchronous or asynchronous way. However, using a clock signal to mark when state transitions should occur makes life much easier for designer, and generally results in most faster circuits at the system level.

On this course, we will only consider sequential circuits that are synchronous to a single clock signal.

Flip-Flops

- Flip-flops are the fundamental element of sequential circuits
- Flip-flops are essentially 1-bit storage devices
  - outputs can be set to store either 0 or 1 depending on the inputs
  - even when the inputs are de-asserted, the outputs retain their prescribed value
- Flip-flops often have 2 complimentary outputs: Q and $\overline{Q}$



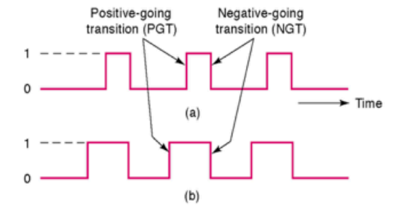
The most basic sequential circuit is a flip-flop (FF). The circuit can store only two states: '0' or '1'. Unlike combinational logic gates, a flip-flop has memory. It is effectively a 1-bit storage element.

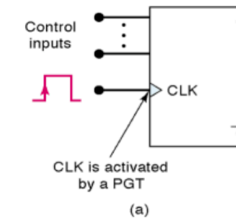
Many FFs have two outputs Q and $\overline{Q}$.

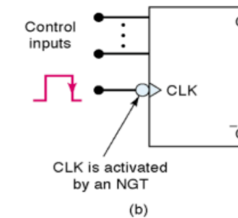
All FFs we considered on this course have a clock and a data signal input.

Clock Signals and Clocked FFs

- Digital systems can operate
  - Asynchronously: output can change state whenever inputs change
  - Synchronously: output only change state at clock transitions (edges)
- Clock signal
  - Outputs change state at the transition (edge) of the input clock
  - Positive-going transitions (PGT)
  - Negative-going transitions (NGT)



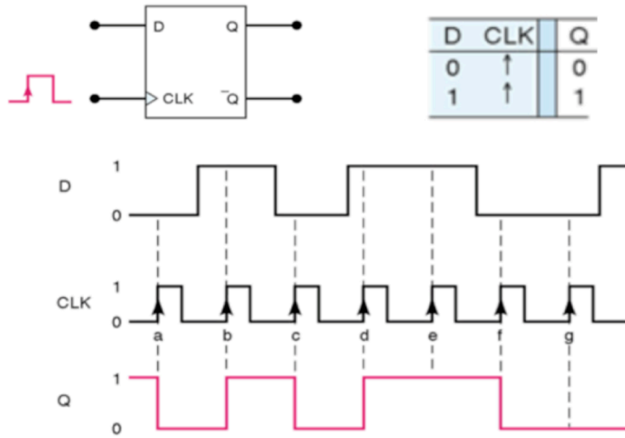




All synchronous circuits use a clock to determine when they change states. The change could occur on the rising edge (positive going) or the falling edge (negative going) of the clock signal. Such "edge-triggered" flip-flops have a small triangle next to the clock wire. Further, we use a circuit to indicate a FF that is triggered on the falling (negative) edge of the clock.

Clocked D Flip-Flop

- D-FF that triggers only on positive-going transitions:



D	CLK	Q
0	↑	0
1	↑	1

In all textbooks, they go through many different TYPES of flip-flops. Actually, the only useful type and is found almost universally in all digital circuits is the D-type flip-flop (D-FF). (D stands for data.)

The truth table specifying the behaviour of a D-FF is shown here. The D-FF stores the current state (i.e. Q is set or '1' or reset or '0'). D input can change as much as it likes, Q does not respond.

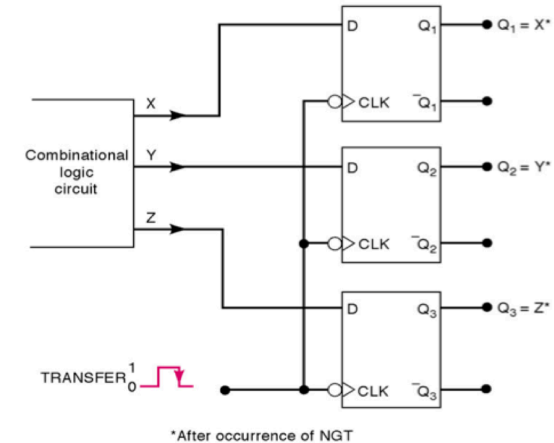
However, when CLK input goes from low to high (positive edge), the logic value of D is sampled and stored in the D-FF, and Q changes.

This is like taking a photograph. The scene (i.e input D) may change, and it does not affect the picture (Q output) until you press the trigger button (the CLK signal).

You can therefore view the D-FF in two ways:

1. It capture the data input on the command of CLK, and keep this data at the output Q for use by other circuits.
2. It blocks any changes on the data input and ignore them. These changes (such as glitches from combinational gates) will be ignore until the active edge of the clock comes along.

Parallel Data Transfer using D-FF

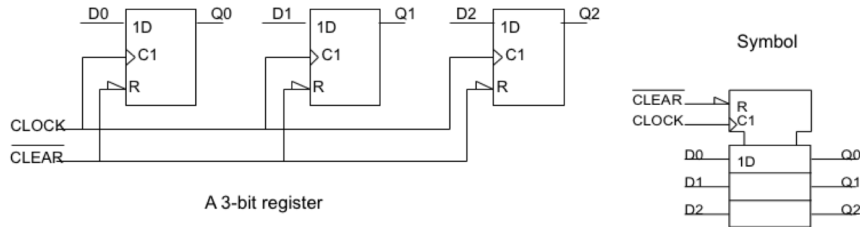


*After occurrence of NGT

D-FFs are usually used to "register" or "latch" logic values from combinational logic circuits. Shown here are three signals X, Y, Z from logic gates. The logic gates perform Boolean computation and, after some delay, reach final values. These values are not visible on the outputs Q1, Q2 and Q3 until a negative edge occurs on the CLK input. At that moment, the values of X, Y and Z are captured by the three D-FFs, and stored. These values are presented on the Q outputs (and its inverse).

Registers

- A register is a digital electronic device capable of storing several bits of data
  - Normally made from D-type flip-flops with asynchronous RESET inputs
  - Operates on the bits of the data word in parallel (parallel in / parallel out)
- Operation
  - Data on each data input is stored in the flip-flop on the rising edge of CLOCK
  - The data can be read from the Q outputs
  - New data can be reloaded by re-CLOCKing the register
  - The register can be cleared (zeroed) by asserting the CLEAR inputs



A 3-bit register

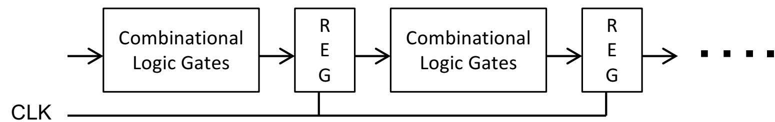
Symbol

CLEAR
CLOCK
R
C1
1D
D0
D1
D2
Q0
Q1
Q2

If you combine a number of D-FFs together as shown here (for three D-FFs), all driven by the same CLOCK signals, but have separate D inputs, we form a useful digital building block called a **register**.

You will find registers in all microprocessors and microcontrollers. They are used to store a temporary variables whose values are then fed to adders or other computational circuits (which are the combinational logic gates).

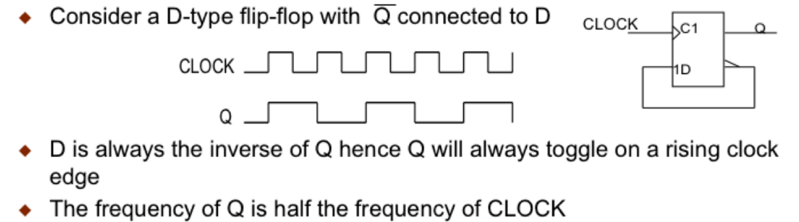
In many digital systems, you will find combinational logic gates and registers interleaving each other, one after another like this:



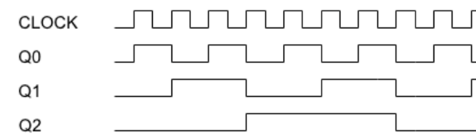
This way of organising digital circuit is known as "pipelining". It is the same idea as a production pipeline where the registers act as shelves (storage units) between teams of workers or machines doing the assembly (logic gates). The CLK signal makes the transition form one stage of the pipeline to the next.

Shown above is a compact symbol for a three bit registers. The CLEAR signal is low active meaning that it is normally '1'. When it goes low, the content of the D-FF are all reset (i.e. zeroed).

Divide frequency by 2 circuit



- Consider a D-type flip-flop with $\overline{Q}$ connected to D
- D is always the inverse of Q hence Q will always toggle on a rising clock edge
- The frequency of Q is half the frequency of CLOCK
- Example: 3-bit counter



Decimal	Q2	Q1	Q0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

- $f_{Q1} = f_{Q0}/2, \quad f_{Q2} = f_{Q1}/2$

If we connect the not_Q output back to the D input of the D-FF, we have a frequency divide-by-2 circuit. The D-FF acts as a toggle: on each rising edge of the clock, the output changes states (i.e. toggles). This results in Q having a frequency half that of CLOCK.

If you now use the Q output of one D-FF as the CLOCK input to the next, we can implement a binary counter as shown here (Q0 is the least significant bit or LSB).

We now divide the frequency of CLOCK, by 2, then by 4 and then by 8 etc.