Digital Electronics

Consider from different levels of abstraction:

- Transistors built from semiconductors
- Logic gates built from transistors
- Logic functions built from gates
- Flip-flops built from logic
- Counters and Sequences from flip-flops
- Microprocessors from sequencers
- Computers from microprocessors

Combinational Logic

Logic and Logic Gates

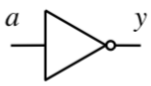
Logic Variables (Binary Variables, Boolean Variables)

Can only take two values (0 or 1). In Electronics, the high voltage is known as 'high' while the low voltage is known as "low". Since only two voltage levels are used, the circuits have a greater immunity to electrical noise.

Basic logic circuits with one or more inputs and one output are known as gates. They are used as the building blocks in the design of more complex digital logic circuits. In order to represent logic functions, the symbols of the gates can be used, truth tables can be used, or Boolean algebra can be used.

NOT Gate

Symbol



Truth-table

a	y
0	1
1	0

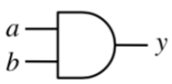
Boolean

$$y = \bar{a}$$

- A NOT gate is also known as an inverter.
- The bubble on the output of a gate implies that is an inverting (or complemented output of the rest of the date).

AND Gate

Symbol



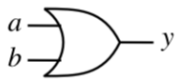
Truth-table

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

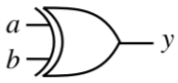
Boolean

$$y = a.b$$

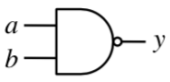
OR Gate

Symbol	Truth-table	Boolean
	$a\ b \mid y$ $0\ 0 \mid 0$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 1$	$y = a + b$

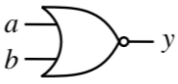
XOR Gate

Symbol	Truth-table	Boolean
	$a\ b \mid y$ $0\ 0 \mid 0$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 0$	$y = a \oplus b$

NOT AND (NAND) Gate

Symbol	Truth-table	Boolean
	$a\ b \mid y$ $0\ 0 \mid 1$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 0$	$y = \overline{a.b}$

NOT OR (NOR) Gate

Symbol	Truth-table	Boolean
	$a\ b \mid y$ $0\ 0 \mid 1$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 0$	$y = \overline{a + b}$

Boolean Algebra

Combinational Logic Circuits: Circuits that do not have an internal stored state, i.e. they have no memory. Therefore, the output is solely a function of the current inputs.

Simple AND rules:

$a.0 = 0$
$a.a = a$
$a.1 = a$
$a.\bar{a} = 0$

Simple OR rules:

$a + 0 = a$
$a + a = a$
$a + 1 = 1$
$a + \bar{a} = 1$

Law: AND takes precedence over OR, i.e a.b + c.d = (a.b) + (c.d)

Commutation

$a + b = b + a$
$a.b = b.a$

Association
$(a + b) + c = a + (b + c)$
$(a.b).c = a.(b.c)$

Distribution
$a.(b + c + \cdots) = (a.b) + (a.c) + \cdots$
$a + (b.c. \ldots) = (a + b).(a.c) \ldots$

Absorption
$a + (a.c) = a$
$a.(a + c) = a$

Expansion Technique
- A useful technique is to expand each term until it includes one instance of each variable or its compliment. From here, it may be possible to simplify the expression by cancelling terms in this expanded form.
- e.g Proving absorption rule
  - $a + a.b = a.\left(b + \bar{b}\right) + \ a.b = a.b + a.b + a.\bar{b} = a.b + a.\bar{b} = a.\left(b + \bar{b}\right) = \ a$

DeMorgan's Theorem
- In a simple expression, change all operators from OR to AND (or visa versa), complement each term and then complement the whole expression.

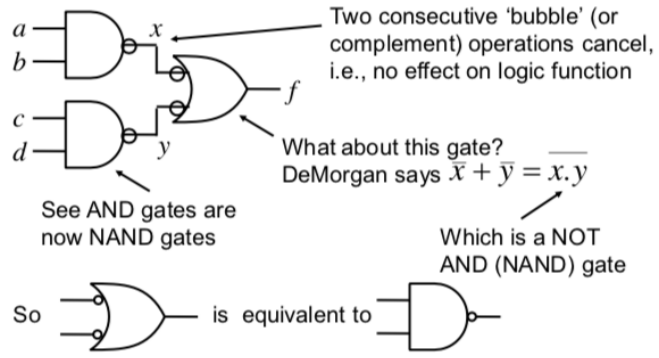
$$a+b+c+ \ldots = \overline{\bar{a}.\bar{b}.\bar{c}. \ldots}$$
$$a.b.c. \ldots = \overline{\bar{a}+\bar{b}+\bar{c}+ \ldots}$$

- For two variables, we can show it is true by using a truth table.

a	b	$\overline{a+b}$	$\overline{a.b}$	$\bar{a}$	$\bar{b}$	$\bar{a}.\bar{b}$	$\bar{a}+\bar{b}$
0	0	1	1	1	1	1	1
0	1	0	1	1	0	0	1
1	0	0	1	0	1	0	1
1	1	0	0	0	0	0	0

- It can be extended to more variables by induction
  - $\overline{a + b + c} = \overline{(a + b)}.\bar{c} = \left(\bar{a}.\bar{b}\right).\bar{c} = \ \bar{a}.\bar{b}.\bar{c}$

Bubble Logic
- We sometimes only wish to use NAND or NOR gates, as they are simpler and faster.
- To do this, we can use bubble logic, by placing two consecutive bubbles (which would cancel) on both sides of connections to gates. Then where there is a bubble before the start of all inputs of a gate, the gate is simply complemented.



Truth Tables

- Where f is a function of the inputs and f is the output, the **minterms** are the things which lead to f having a high value.

x	y	z	f	minterms
0	0	0	1	$\bar{x}.\bar{y}.\bar{z}$
0	0	1	1	$\bar{x}.\bar{y}.z$
0	1	0	1	$\bar{x}.y.\bar{z}$
0	1	1	1	$\bar{x}.y.z$
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	1	$x.y.z$

Minterms

- A minterm must contain all variables (in either complement or uncomplemented form)
    - Variables in a minterm are ANDed together (conjunction)
    - One minterm for each term of f that is true.
- A Boolean function expressed as the disjunction (ORing) of its minterms is said to be in disjunctive normal form (DNF).
    - A boolean function expressed as the ORing of ANDed variables (**not necessarily minterms**) are in **SUM OF PRODUCTS form (SOP).**

Maxterms

- A maxterm of n Boolean variables is the disjunction of all the variables either in complemented or uncomplemented form.
    - $\bar{f} = x.y.z + \bar{x}.y.z$ …
    - By applying De Morgan's law, you get:
    - $f = (\bar{x} + \bar{y} + \bar{z}).(x + \bar{y} + \bar{z})$
        - This the **Conjunctive Normal Form (ANDing of maxterms)**
        - Function expressed as the ANDing of ORed variables (not necessarily maxterms) is the **PRODUCT OF SUMS form (POS).**
- Maxterms are effectively the minterms of $\bar{f}$ with each variable complemented.

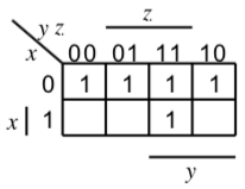
Logic Minimisation

Karnaugh Mapping

- K-Maps are a powerful visual tool for carrying out simplification and manipulation of logical expressions having up to 5 variables

- It is a rectangular array of cells, with each possible state of the input variables corresponding uniquely to one cell. The corresponding output state is written in each cell.

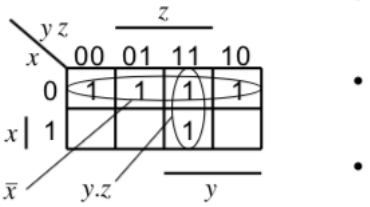
x	y	z	f
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



Note that the logical state of the variables follows a Gray code, i.e., only one of them changes at a time

The exact assignment of variables in terms of their position on the map is not important

- Having plotted the minterms, you find groups having a size equal to a power of 2: 1, 2, 4, 8 etc
    - The larger the group, the better, since they contain fewer variables.
    - The groups can wrap around edges and corners.



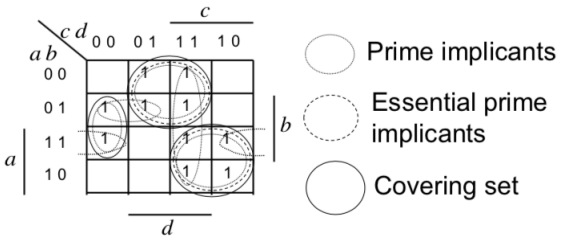
So, the simplified func. is,

$f = \bar{x} + y.z$ as before

- When you group the 1s, you produce a simplified expression in the Sum-of-Products form, which is suitable for implementations using AND followed by OR gates (or just NAND gates – using bubble logic)
- Instead if you group by 0 (simplify the complement of the function) and then apply De Morgan's, you find a Product-of-Sums expression for the function
    - This is suitable for implementation using OR followed by AND gates, or just NOR gates.
- **Don't Care Conditions**
    - Sometimes if we do not care about the output value of a logic circuit for some particular inputs (they can never occur), they are known as don't care conditions.
    - In any simplification, they may be treated as 0 or 1, depending upon which gives the simplest result – therefore they are entered as 'X's.

Definitions

Cover: A term is said to cover a minterm if that minterm is part of that term.
Prime Implicant: A term that cannot be further combined.
Essential Prime Implicant: A prime implicant that covers a minterm that no other prime implicant covers.
Covering Set: A minimum set of prime implicants which includes all essential terms plus any other prime implicants required to cover all minterms.



Quine-McCluskey (Q-M) Method

The K-map method is not practical beyond 6 variables. It is also much harder to complete by a computer. Therefore, the Q-M Method is able to find the minimised representation of any Binary Expression. It is a tabular method that ensures that all the prime implicants are found.

The Q-M Method has two steps:

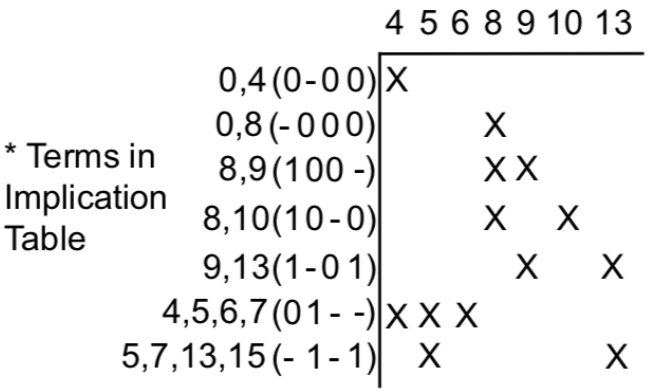
1. Create the QM implication table, which is used to find all the prime implicants.
    a. List all the minterms (and don't cares) in terms of their minterms indices represented as a binary number.
    b. The entries are grouped according to the number of 1s in the binary representation.
    c. The 1st column contains the minterms, while after applying the method, the 2nd column contains one fewer variable terms. Similarly for subsequent columns.
    d. Get between columns using the **uniting theorem**
        i. Compare elements in the first group (by number of 1s) with all elements in the 2nd group. If they differ by a single bit, it means the terms are adjacent (in a K-map).
        ii. Adjacent terms are placed in the 2nd column with the single bit that differs replaced by a dash.
        iii. Terms in the 1st column that contribute to a term in the second are ticked (they aren't prime implicants).
        iv. This is then repeated for all the groups in the first column and then second column, etc.
        v. If you get through a column and a term is not covered, then it is marked with an asterix (*) and it is a prime implicant.

– Minterms are: 4,5,6,8,9,10,13
– Don't cares are: 0,7,15.

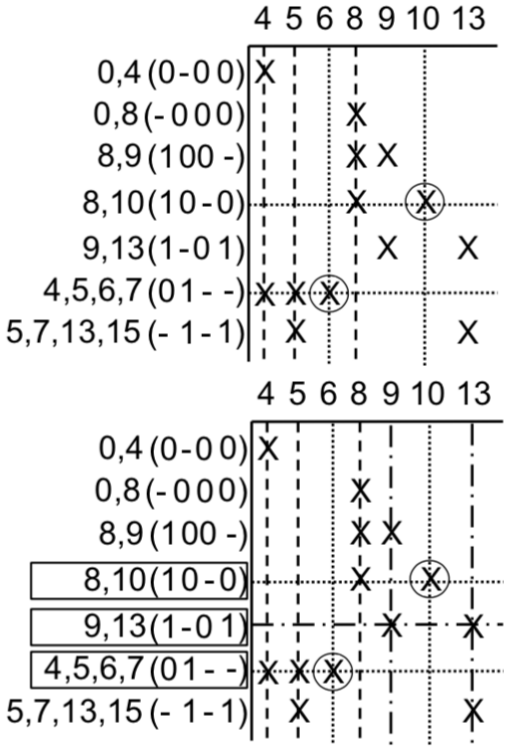
Column 1	Column 2	Column 3
0 0 0 0✓	0 - 0 0*	0 1 - - *
0 1 0 0✓	- 0 0 0*	- 1 - 1*
1 0 0 0✓	0 1 0 - ✓	
0 1 0 1✓	0 1 - 0✓	
0 1 1 0✓	1 0 0 - *	
1 0 0 1✓	1 0 - 0*	
1 0 1 0✓	0 1 - 1✓	
0 1 1 1✓	- 1 0 1✓	
1 1 0 1✓	0 1 1 - ✓	
1 1 1 1✓	1 - 0 1*	
	- 1 1 1✓	
	1 1 - 1✓	

2. The minimum cover set is found using the prime implicant table
    a. Here, you write all the minterms (excluding the don't cares) as the column names.

b. And the found prime implicants as the rows, with an ‘X’ on the minterms that this prime implicant covers.



c. Now, we look for the essential prime implicants – when there is only a single X in any column – this means there is a minterm covered by one and only one prime implicant.
d. These terms are generally circled and the column which they cover is drawn through. Any other minterms that the required prime implicants covered should also been drawn through.
e. From here, you attempt to find as few other prime implicants required to cover the remaining minterms.



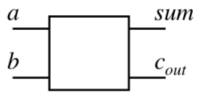
Binary Adders

Half Adder

Adds two, single bit binary numbers (a and b) and returns a sum and a carry bit.

- Has the following truth table:

a	b	c_{out}	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

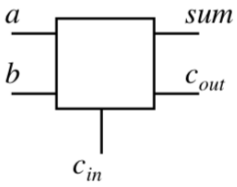


- By inspection:

$$sum = \bar{a}.b + a.\bar{b} = a \oplus b$$
$$c_{out} = a.b$$

Full Adder

Adds together two single bit binary numbers (a, b and c (c being a carry input)).



c_{in}	a	b	c_{out}	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$sum = \bar{c}_{in}.\bar{a}.b + \bar{c}_{in}.a.\bar{b} + c_{in}.\bar{a}.\bar{b} + c_{in}.a.b$$
$$sum = \bar{c}_{in}.(\bar{a}.b + a.\bar{b}) + c_{in}.(\bar{a}.\bar{b} + a.b)$$

From DeMorgan

$$\bar{a}.\bar{b} + a.b = \overline{(a+b).(\bar{a}+\bar{b})}$$
$$= \overline{(a.\bar{a} + a.\bar{b} + b.\bar{a} + b.\bar{b})}$$
$$= \overline{(a.\bar{b} + b.\bar{a})}$$

So,

$$sum = \bar{c}_{in}.(\bar{a}.b + a.\bar{b}) + c_{in}.\overline{(\bar{a}.b + a.\bar{b})}$$
$$sum = \bar{c}_{in}.x + c_{in}.\bar{x} = c_{in} \oplus x = c_{in} \oplus a \oplus b$$

AND

c_{in}	a	b	c_{out}	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

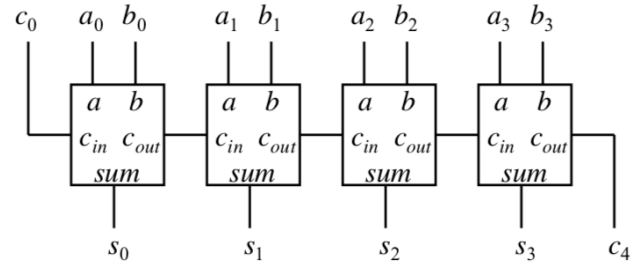
$$c_{out} = \bar{c}_{in}.a.b + c_{in}.\bar{a}.b + c_{in}.a.\bar{b} + c_{in}.a.b$$
$$c_{out} = c_{in}.(\bar{a}.b + a.\bar{b}) + a.b.(c_{in} + \bar{c}_{in})$$
$$c_{out} = c_{in}.(a \oplus b) + a.b$$

While this is done through simplification, it could just as easily be done using a K-Map.

Ripple Carry Adder

A ripple carry adder is simply n full adders cascaded together (allowing us to add together two n bit binary numbers).

Example, 4 bit adder



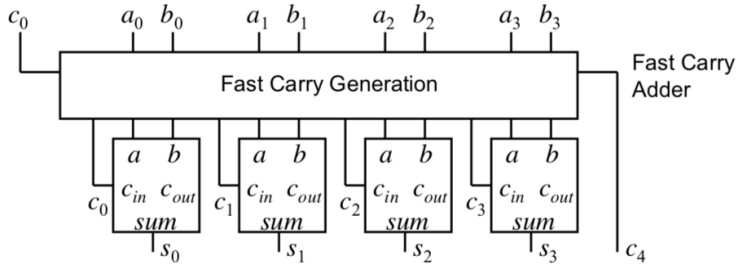
While being effective, it has issues as each adder can only work when the carry has been generated by the previous adder, therefore it has to go through each adder simultaneously, meaning the system takes a lot of time.

In order to speed up the ripple carry adder means we **abandon the compositional approach to the adder design,** instead designing the adder as a block of 2-level combinational logic with 2n inputs (+1 for carry in) and n outputs (+1 for carry out).

This has a low delay (with 2 gate delays), but it needs some gates with a large number of inputs and is very complex to design and implement (the truth table would be astonishingly long).

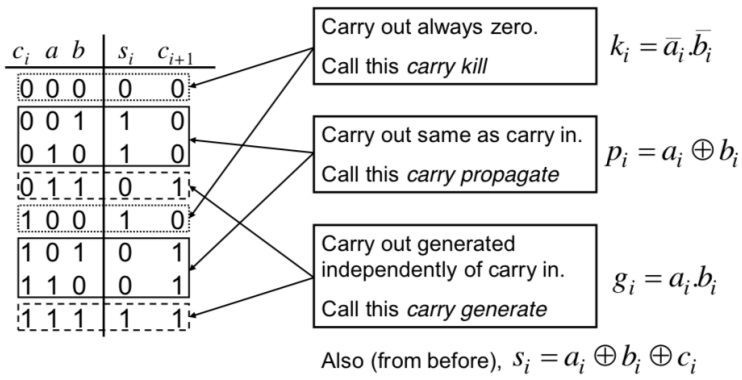
This is not feasible in any way. Therefore, another approach is to make use of full-adder blocks, but to generate the carry signals independently using fast carry generation logic. This means we do not have to wait for the carry signals to ripple from adder to adder.

Fast Carry Generation



We can easily show that:

$$c_{i+1} = a_i.b_i + c_i.(a_i + b_i)$$



So, $c_{i+1} = g_i + c_i.p_i$

Therefore, you can easily find expressions for any carry in terms of the 0th carry and a's and b's of previous items, such as:

So for example to generate c_4, i.e., i = 0,

$$c_4 = g_3 + p_3.(g_2 + p_2.(g_1 + p_1.g_0)) + p_3.p_2.p_1.p_0.c_0$$

$$c_4 = G + Pc_0$$

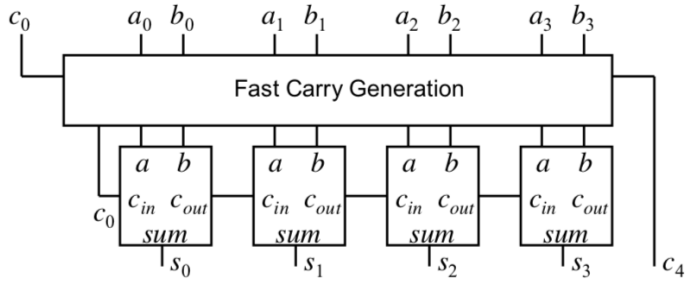
where,

$$G = g_3 + p_3.(g_2 + p_2.(g_1 + p_1.g_0))$$

$$P = p_3.p_2.p_1.p_0$$

This is reasonably rapid to evaluate the function.

We could generate all the carrys within an adder block using the previous equations described but in order to reduce complexity, another better approach is to implement 4-bit adder blocks with only the last carry (out of the block) being generated using fast carry generation. Within each adder block, conventional Ripple Carry adding is used.



Multilevel Logic

We can minimise Boolean expressions to yield '2-level' logic implementations (SOP or POS). However, multilevel logic is often better:

- Commercially available gates usually available with a restricted number of inputs, typically, 2 or 3.
- System composition from sub-systems reduces design complexity – eg a ripple adder made from full adders.
- Allows Boolean optimisation across multiple outputs, eg common sub-expression elimination.

In order to better design multilevel logic, we can recursively factor out common literals, to create a similar expression. Then we can express the original function in terms of the groups of literals created, eg:

$$z = a.d.f + a.e.f + b.d.f + b.e.f + c.d.f + c.e.f + g$$
$$z = (a.d + a.e + b.d + b.e + c.d + c.e).f + g$$
$$z = ((a+b+c).d + (a+b+c).e).f + g$$
$$z = (a+b+c).(d+e).f + g$$

- Now express z as a number of equations in 2-level form:

$$x = a+b+c \qquad y = d+e \qquad z = x.y.f + g$$

- 4 gates, 9 literals, 3-levels

Hazards

Gate Propagation Delay

There will be a finite delay before the output of a gate responds to a change in its inputs – propagation delay.

The cumulative delay owing to a number of gates in cascade can increase the time before the output of a combinational logic circuit becomes valid.

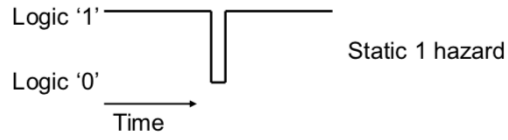
In addition to slowing down the operation of combinational logic circuits, gate delay can also give rise to 'Hazards' at the output. These are unwanted brief logic level changes in response to changing inputs.

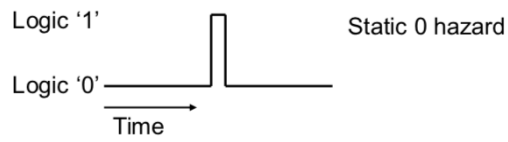
Timing Diagrams

To visually represent hazards, we will use a timing diagram. This shows the logical value of a signal as a function of time. The diagram makes a number of assumptions:

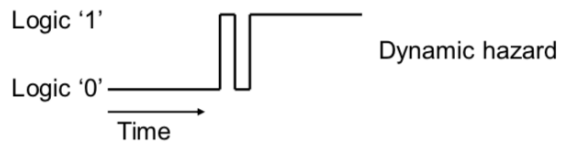
- Signal only has two levels. In reality, the signal may look more wobbly, due to electrical noise
- Transition between logic levels takes place instantaneously, in reality it will take finite time.

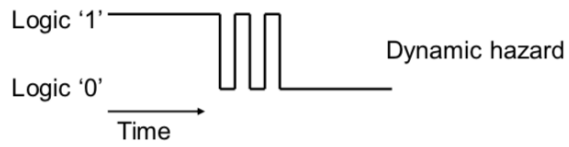
Static Hazards: The output undergoes a momentary transition when one input changes when it is supposed to remain unchanged.



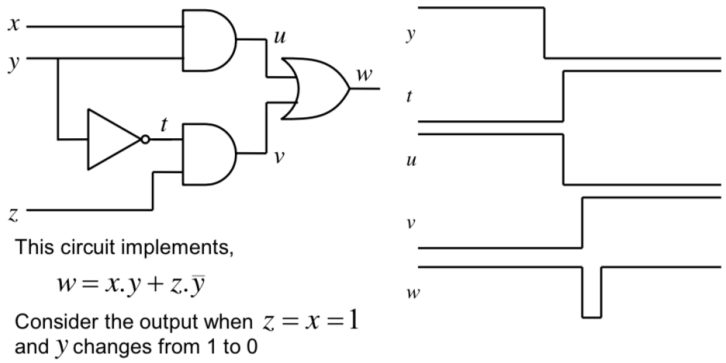


Dynamic Hazard: The output changes more than once when it is supposed to change just once.





In order to remove a static hazard, you should K-Map the inputs and outputs, and attempt to draw a prime implicant between the essential prime implicants. In the case of a 1 hazard, this should be working with the 1s, in the case of a 0 hazard, add another term which overlaps the essential terms (representing the complement).



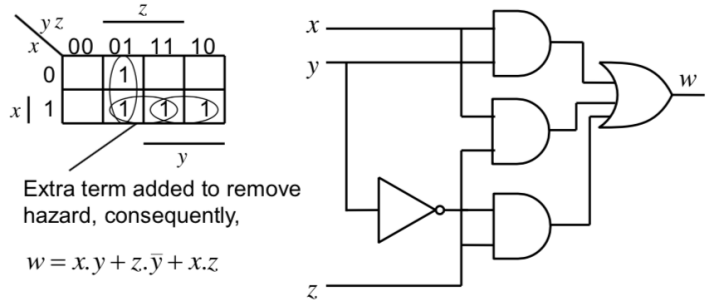
This circuit implements,

$$w = x.y + z.\bar{y}$$

Consider the output when $z = x = 1$ and y changes from 1 to 0

GOES TO

$$w = x.y + z.\bar{y}$$

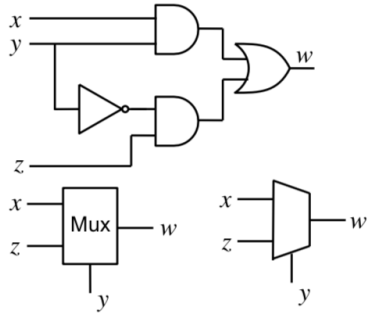


Extra term added to remove hazard, consequently,

$$w = x.y + z.\bar{y} + x.z$$

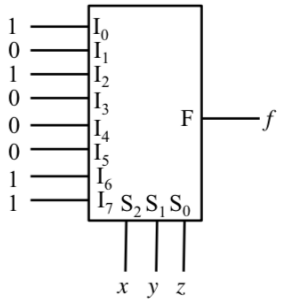
Multiplexors and Demultiplexors

A **Multiplexor (Mux) / Selector** chooses 1 of many inputs to make as its single output under the direction of control inputs. The hazard example was actually a 2-to-1 mux, it can select either input x or z to appear at output w under control of y:



x	z	y	w
0	0	0	0
0	1	0	1
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	0
1	0	1	1
1	1	1	1

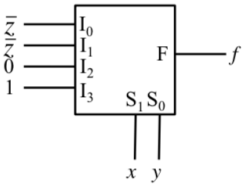
A Mux can also be used to implement combinational logic functions. For example an 8 input Mux can be used to implement functions having 3 variables expressed as a sum of minterms, eg:



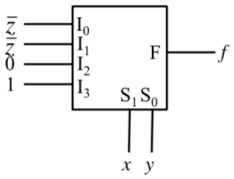
The control inputs are used to select the minterms required at the output. **The Mux is sometimes called a hardware lookup table.**

You can sometimes use a smaller minterm by utilising some of the inputs (rather than control signals) as an input by simplifying the expression into minterms which depend on the value of it (or using a truth table method), eg:

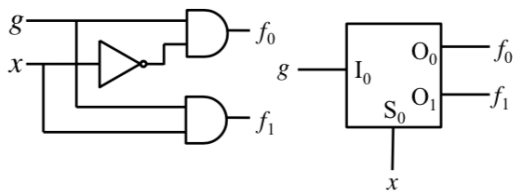
$$f = \bar{x}.\bar{y}.\bar{z} + \bar{x}.y.\bar{z} + x.y.\bar{z} + x.y.z$$
$$f = (\bar{x}.\bar{y} + \bar{x}.y).\bar{z} + x.y.(z + \bar{z})$$
$$f = (\bar{x}.\bar{y} + \bar{x}.y).\bar{z} + x.y$$



x	y	z	f	
0	0	0	1	$I_0 = \bar{z}$
0	0	1	0	
0	1	0	1	$I_1 = \bar{z}$
0	1	1	0	
1	0	0	0	$I_2 = 0$
1	0	1	0	
1	1	0	1	$I_3 = 1$
1	1	1	1	



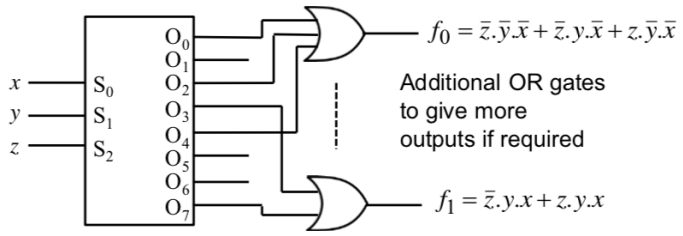
A **Demultiplexor** is the opposite of a Mux, a single input is directly to exactly one of its outputs. The truth table (and circuit) for a 1-to-2 Demux is:



g	x	f_0	f_1
0	0	0	0
1	0	1	0
0	1	0	0
1	1	0	1

In a Decoder, the input is permanently connected to logic 1. A 1-to-n Decoder is possible, being able to Enable (EN) 1 out-of-n logic sub-systems. A 1-of-n Decoder will generate all the possible minterms having n variables. Therefore, a logical expression having DNF form can be implemented by ORing together the required minterms at the decoder output.

Multiple output logic blocks can be created by using multiple can be created by using multiple OR gates at the decoder output – i.e. one for each output.

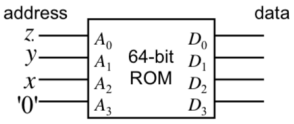


ROM

A ROM is a data storage device:

- Usually written into once (either at manufacture or using a programmer)
- You can read from it at will
- It is essentially a look-up table, where a group of input lines (n) is used to specify the address of locations holding m-bit data words.
    - Total number of bits = m x 2^n

Design amounts to putting minterms in the appropriate address location. This means that no logic simplification is required. This is useful if multiple Boolean functions are to be implemented.

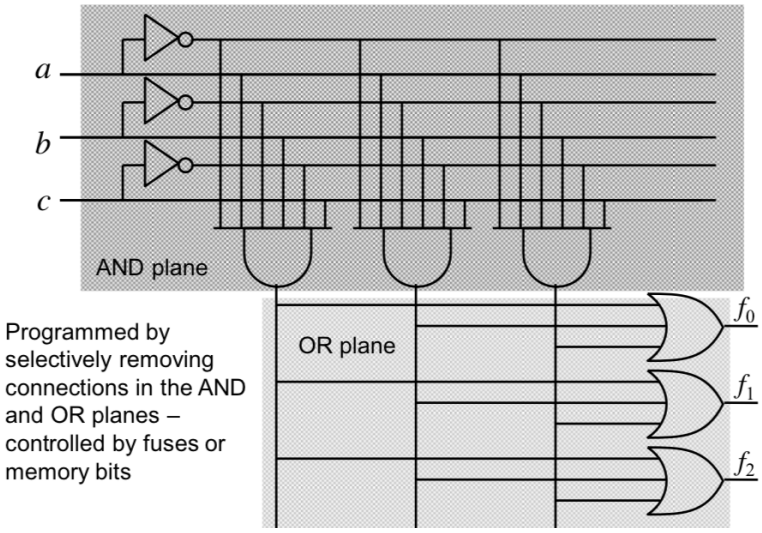


address (decimal)	x	y	z	f	data $D_3 D_2 D_1 D_0$
0	0	0	0	1	X X X 1
1	0	0	1	1	X X X 1
2	0	1	0	1	X X X 1
3	0	1	1	1	X X X 1
4	1	0	0	0	X X X 0
5	1	0	1	0	X X X 0
6	1	1	0	0	X X X 0
7	1	1	1	1	X X X 1

However, the implementation of ROM can be quite inefficient. It becomes of a very large size with only a few non-zero entries. If the number of minterms in the function to be implemented is quite small, then it can be effective.

Programmable Logic Array (PLA)

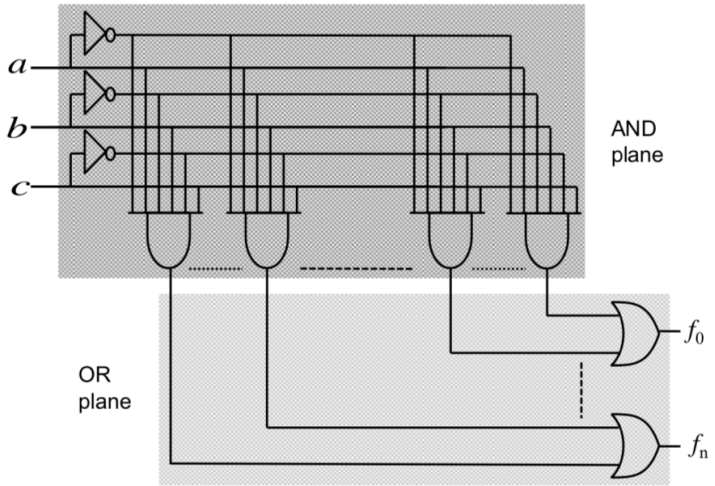
A device which overcomes the problems of a ROM is a PLA. In a PLA, only the required minterms are generated using a separate AND plane. The outputs from this plane are ORed together in a separate OR plane to produce the final output.



Programmable Array Logic (PAL)

A PAL is a modified structure of a PLA which does not have a programmable OR array and so **outputs from the AND array cannot be shared among the OR gates to give the final outputs**.

This simplifies the structure, but at the cost of lower efficiency. It is however, much easier to produce:



Other Memory Devices

Non-volatile storage (the data remains intact even when the power supply is removed) is offered by ROMs and some other memory technologies such as FLASH.

Volatile storage is offered by Static Random Access Memory (SRAM). Here, data can be written into and read out of the SRAM, but is lost once power is removed.

Bus Contention and Tristate Buffer

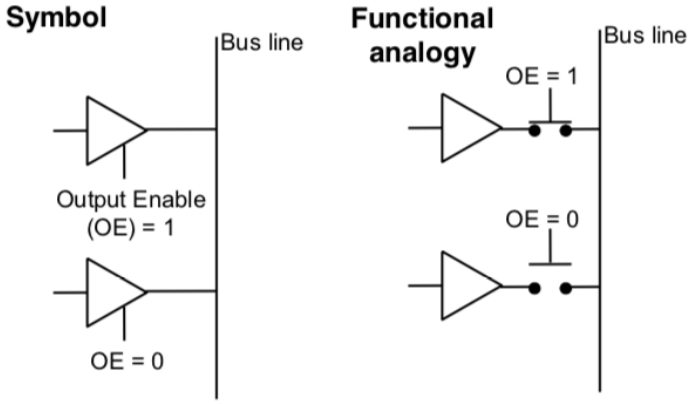
Memory devices are often used in computer systems. The CPU often makes use of busses (a bunch of parallel wires) to access external memory devices. The address bus is used to specify the memory location that is being read or written and the data bus conveys the data to and

from that location. **Therefore, more than one memory device will often be connected to the same data bus.**

Bus contention is when the output of a data pin of a memory is lost (when it has value of 0) when there is another data pin of a different memory which has 1. Therefore, the value of one of the pins is lost.

The answer is solved using **Tristate Buffers** or **Control Signals.**

A tristate buffer is used on the data output of the memory devices., In contrast to a normal buffer, which is either 1 or 0 at the output, a tristate buffer can be electrically disconnected from the bus wire. (HIGH IMPEDANCE CONDITION)



Control Signals

1. Memory devices have **a Control Input (OE)** which determines whether the output buffers are enabled.
2. They also have a **Write Enable (WE)** which determines whether data is written or read (clearly not needed on a ROM).
3. AND a **Chip Select (CS)** which determines if the chip is activated.

N.B. The signals can be active low, depending upon the particular device.

Sequential Logic

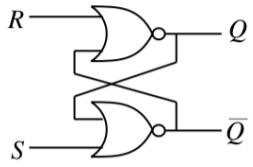
In Sequential Logic the output depends not only on the latest inputs but only on the condition of earlier inputs. These circuits are known as sequential, and implicitly they contain memory elements.

Definitions

- Memory stores data – usually one bit per element
- A snapshot of the memory is called the state
- A one bit memory is often called a bistable – ie it has 2 stable internal states
- Flip-flops and latches are particular implementations of bistables.

RS Latch

An RS Latch is a memory element with 2 inputs: Reset (R) and Set (S) and 2 outputs ($Q \ and \ \bar{Q}$).



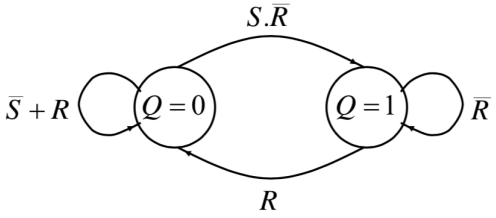
S R	Q'	$\overline{Q}'$	comment
0 0	Q	$\overline{Q}$	hold
0 1	0	1	reset
1 0	1	0	set
1 1	0	0	illegal

Where Q' is the next state and Q is the current state

It can be viewed using this state transition table (which shows the next state logic).

Q S R	Q'	comment
0 0 0	0	hold
0 0 1	0	reset
0 1 0	1	set
0 1 1	0	illegal
1 0 0	1	hold
1 0 1	0	reset
1 1 0	1	set
1 1 1	0	illegal

It can also be viewed using a state diagram:



Clock

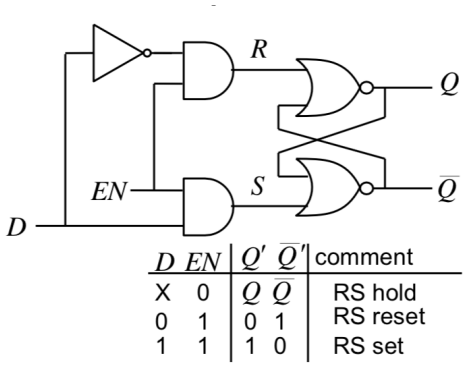
For the RS Latch, the output state changes directly in response to changes in the inputs. This is known as asynchronous operation. However, virtually all sequential circuits employ the notion of synchronous operation – that is the output of a sequential circuit is constrained to change only at a time specified by a global enabling signal. This is generally known as the system clock.

The Clock

1. Creates a square wave at a particular frequency
2. Imposes order on the state changes
3. Allows lots of states to appear to update simultaneously.

(Transparent) D Latch

A Transparent D Latch is an RS Latch where the output state is only permitted to change when a valid enable signal (which could be the system clock) is present.
This is produced by introducing a couple of AND gates in cascade with the R and S inputs that are controlled by an additional input known as the enable (EN) input.



D	EN	Q'	$\overline{Q}'$	comment
X	0	Q	$\overline{Q}$	RS hold
0	1	0	1	RS reset
1	1	1	0	RS set

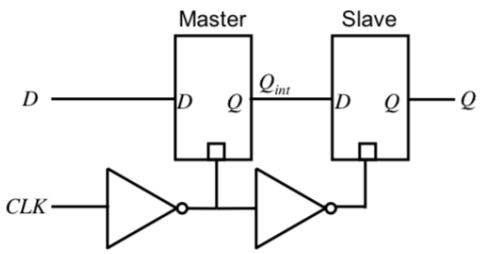
- See Q follows D input provided EN=1.
  If EN=0, Q maintains previous state

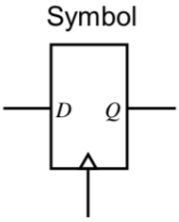
Flip-Flops

Master-Slave D Flip-Flop

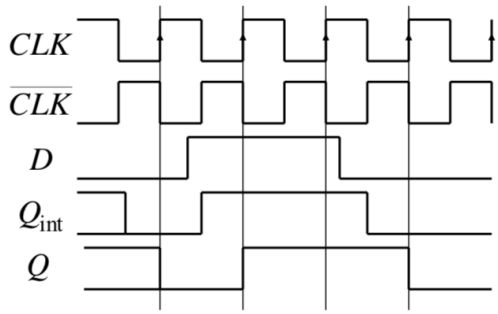
The transparent D latch is 'level' triggered. It exhibits transparent behaviour if EN=1. It is often simpler to design sequential circuits if the outputs change only on the either rising (positive going) or falling (negative going) edges of the clock signal.

We can achieve this kind of operation by combining 2 transparent D latches in a Master-Slave configuration.





As per the timing diagram, you can easily see that the changes only propagate on the rising edge of the clock cycle…



The Master-Slave configuration has now been superseded by new D F-F circuits which are easier to implement and have better performance.

J-K

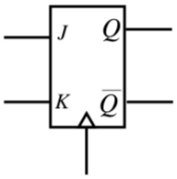
J-K FFs are a lot more complex to build thatn D-types and so have fallen out of favour in modern designs – for FPGAs and VLSIs.

A J-K FF is similar in function to a clocked RS FF, but with the illegal state replaced with a new toggle state.

J	K	Q'	$\overline{Q}'$	comment
0	0	Q	$\overline{Q}$	hold
0	1	0	1	reset
1	0	1	0	set
1	1	$\overline{Q}$	Q	toggle

Where Q' is the next state
and Q is the current state

Symbol



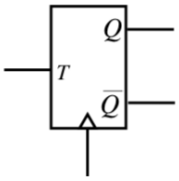
T

A T FF is a J-K FF with its J and K inputs connected together.

T	Q'	$\overline{Q}'$	comment
0	Q	$\overline{Q}$	hold
1	$\overline{Q}$	Q	toggle

Where Q' is the next state
and Q is the current state

Symbol



Asynchronous Inputs

It is common for FF types we have mentioned to also have additional asynchronous inputs. **They take effect independently of any clock or enable inputs.**

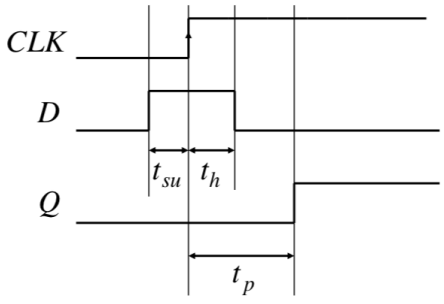
RESET / CLEAR – Force Q to 0
PRESET / SET – Force Q to 1

It is often used to force a synchronous circuit into a known state, say at start-up (hence avoid an unknown / illegal state).

Timing

Setup Time: Minimum duration that the data must be stable at the input before the clock edge.

Hold time: Minimum duration that the data must be stable on the FF input after the clock edge.



t_{su} Set-up time
t_h Hold time
t_p Propagation delay

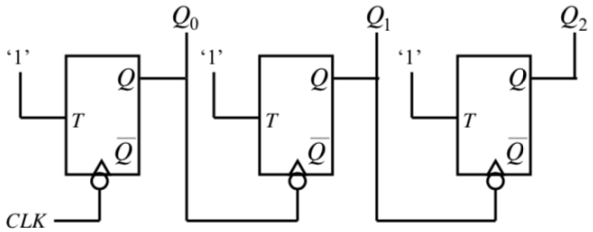
Applications of Flip-Flops

Counters

- Clocked sequential circuit going through predetermined sequence of states
- Eg n-bit binary counter.
  - Has n FFs and 2^n states.
- Used for
  - Counting
  - Producing delays of particular duration
  - Sequencers for control logic in a processor
  - Divider

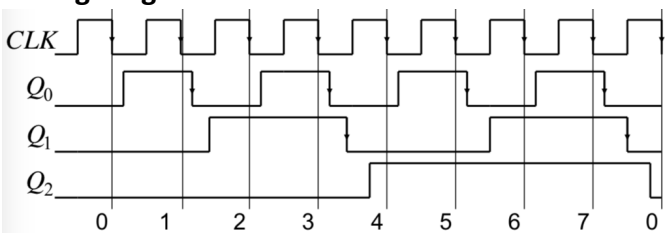
Ripple Counter

A ripple counter can be made by cascading together negative edge triggered T-type FFs operating in toggle mode:
If you observe the frequency of the counter output signals, it can be noted that each has half the frequency of the previous one. This is why counters are also called dividers. Often we wish to have a count which is not a power of 2, eg 0-9. In order to do this, we use FFs which have a reset / clear state and use an AND gate to detect the count of 10 and use its output to reset the FFs.



The FFs are not clocked using the same clock – **this is NOT a synchronous design, this leads to some problems.**

Timing Diagram



Since outputs don't change at the same time, it is hard to know when the count output is actually valid. This is because the propagation delay builds up stage to stage, limiting the maximum clock speed before miscounting occurs.

Synchronous Counter

To assist in designing a counter, a state transition table can be used, with the added columns that define the required FF inputs (or excitation as it is known).

Characteristic Table and Characteristic Equation for D-type FF
A characteristic table for a FF gives the next state of the output (Q') in terms of current state (Q) and current inputs.

Q	D	Q'
0	0	0
0	1	1
1	0	0
1	1	1

Which gives the characteristic equation,

$$Q' = D$$

i.e., the next output state is equal to the current input value

Since Q' is independent of Q the characteristic table can be rewritten as

D	Q'
0	0
1	1

The Characteristic table can be modified to give the excitation table. This table tells us the required FF input value required to achieve a particular next state from a given current state.

Q	Q'	D
0	0	0
0	1	1
1	0	0
1	1	1

Characteristic and Excitation Table for J-K FF

Obviously more interesting than for D-type:

J	K	Q'
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

Truth table

Q	Q'	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Excitation table

For a 0 to 7 counter, 3 D-type FFs are needed:

Current state			Next state			FF inputs		
Q_2	Q_1	Q_0	Q_2'	Q_1'	Q_0'	D_2	D_1	D_0
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	0
0	1	0	0	1	1	0	1	1
0	1	1	1	0	0	1	0	0
1	0	0	1	0	1	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	1	1	1	1	1	1
1	1	1	0	0	0	0	0	0

Since Q’ = D for a D-type FF, the required FF inputs are identical to the Next state.

Once the state transition table has been drawn, you either can spot a solution, or you can K-Map each FF input in terms of the current state.

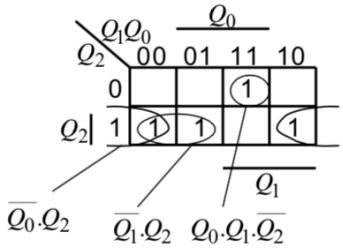
By inspection,

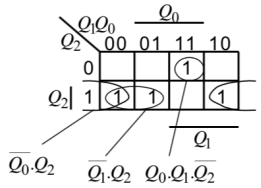
$$D_0 = \overline{Q_0}$$

Note: FF_0 is toggling

Also, $D_1 = Q_0 \oplus Q_1$

Use a K-map for D_2,

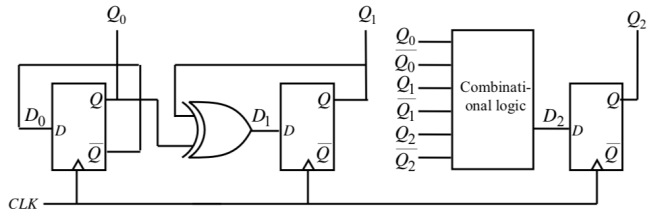




So,

$$D_2 = \overline{Q_0}.Q_2 + \overline{Q_1}.Q_2 + Q_0.Q_1.\overline{Q_2}$$

$$D_2 = Q_2.(\overline{Q_0}. + \overline{Q_1}) + Q_0.Q_1.\overline{Q_2}$$



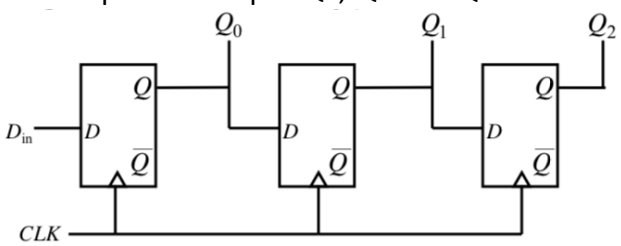
A similar procedure can of course be used to design any counter (or anything going between arbitrary states).

Shift Register

Parallel Loading Shift Register: can be used for parallel to serial conversion in serial data communication.

Serial in, Parallel out Shift Register: can be used for serial to parallel conversion in a serial data communication system.

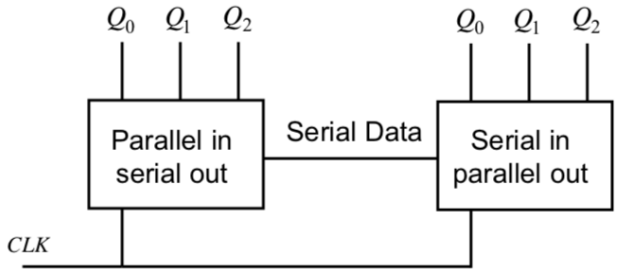
A shift register can be implemented using a chain of D-type FFs. It leads to a serial input of Din and a parallel output Q0, Q1 and Q2.



The Data moves one position to the right on the application of each rising clock edge. This is a **serial in, parallel out shift register arrangement.**

The Preset and Clear inputs (which work irrespective of the clock) can be utilised to provide a parallel data input feature. The data can then be clocked out through Q2 in a serial fashion, ie we have a **parallel in, serial out** arrangement.

Serial Data Link



This is often used since one data bit at a time is sent across the serial data link meaning data can be sent at a high frequency. Additionally, fewer wires are required for the long links.

Synchronous State Machines

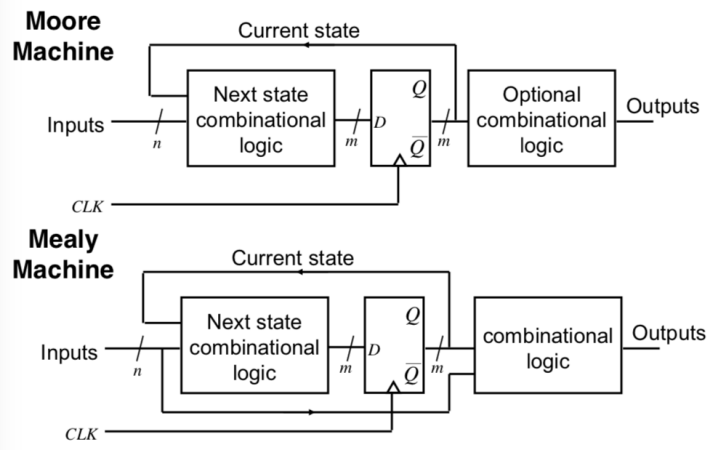
A **Finite State Machine (FSM)** is a deterministic machine (circuit) that produces outputs which depend on its internal state and external inputs.

States – The set of internal memorised values, shown as circles on the state diagram

Inputs – External stimuli, labelled as arcs on the state diagram.

Outputs – Results from the FSM

Moore Machine vs Mealy Machine



Outputs in Mealy Machines depend on the timing of the inputs.

Outputs from a Moore Machine come directly from the clocked FF so they have:

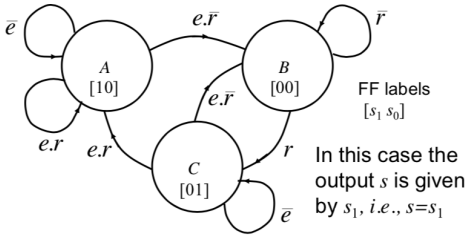
1. **Guaranteed Timing Characteristics**
2. **Glitch Free**

On a Mealy machine there is an output from the combinational logic to the next state combinational logic.

Mealy Machine can have output based on state (and the inputs), rather than just based on the state.

Moore Machine State Diagram

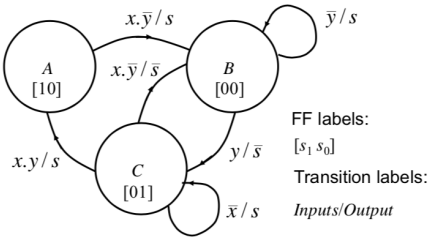
- Example FSM has 3 states (A, B and C), inputs e and r, and output s



- See **inputs only** appear on transitions between states, i.e., next state is given by current state and current inputs
- Outputs determined from current state via combinational logic (if required)

Mealy Machine State Diagram

- Example FSM has 3 states (A, B and C), inputs x and y, and output s



- Inputs **and outputs** appear on transitions between states, i.e., next state is given by current state and current inputs
- Output determined from current state and inputs via combinational logic

For the example of a FSM (the traffic light), see the lecture notes, Page 31.

Problems

Power-Up

A problem could be caused if the FF by chance starts up in one of the unused states – leading to it being stuck in an unanticipated sequence of states which never goes back to a used state.

What to do?

- **Check to see if the FSM can eventually enter a known state from any of the unused states.**
- **If not, add additional logic to do this, i.e. include unused states in the state transition table along with a valid next state.**
- **Alternatively, use asynchronous Clear and Preset FF inputs to set a known (used) state at power up.**

State Assignment

- State assignment is not necessary obvious or straightforward.
- Depends on what you are trying to optimise:
  - Complexity
    - FF implementation may take less chip area than you may think given their gate level representation
    - Wiring complexity can be as big an issue as gate complexity
  - Speed

- Algorithms do exist for selecting the 'optimum' state assignment, but they are not suitable for manual execution.
- **STORAGE**
  - If we have n states we need at least $\log_2 m$ FFs to encode the states.
- **EXAMPLE: Attempting to implement a divide by 5 counter which gives a 1 output for 2 clock edges and is 0 for 3 clock edges.**
- **Sequential State Assignment**
  - **Required output is from FF b**
  - Assign the states sequentially (0, 1, 2 … until all the states are assigned).
  - This generally will leave unused states and require us to generally plot k-maps to determine the next state logic.
- **Sliding State Assignment**

Current state			Next state		
c	b	a	c'	b'	a'
0	0	0	0	0	1
0	0	1	0	1	1
0	1	1	1	1	0
1	1	0	1	0	0
1	0	0	0	0	0

  - We can see that we can use any of the FF outputs as the wanted output
  - We need to plot k-maps to determine the next state logic.
    - This logic is also much simpler.
- **Shift Register Assignment**
  - The FFs are connected together to form a shift register. In addition, the output from the final shift register is connected to the input of the first FF.
    - Therefore, the data continuously cycles through the register
  - We could again use any of the things as an output.
  - The Logic is also much simpler
  - But it requires two more FFs

Current state					Next state				
e	d	c	b	a	e'	d'	c'	b'	a'
0	0	0	1	1	0	0	1	1	0
0	0	1	1	0	0	1	1	0	0
0	1	1	0	0	1	1	0	0	0
1	1	0	0	0	1	0	0	0	1
1	0	0	0	1	0	0	0	1	1

Unused states. Lots!

Because of the shift register configuration and also from the state table we can see that:

$$D_a = e$$
$$D_b = a$$
$$D_c = b$$
$$D_d = c$$
$$D_e = d$$

By inspection we can see that we can use any of the FF outputs as the wanted output

See needs 2 more FFs, but no logic and simple wiring

- **One Hot State Encoding**
  - Shift Register Design Style where only one FF at a time holds a 1.
  - Therefore 1 FF per state.
  - However, can result in simple, fast machines
  - Outputs are generated by ORing together appropriate FF outputs.

Elimination of Redundant States

When designing state machines, it is possible that unnecessary states may be introduced. By reducing the number of states, it is likely you reduce the number of FFs required, thereby reducing the complexity of the next state logic owing to the presence of more unused states.

The states should be listed out with the location of the next state given a particular input and the output given a particular input, hence:

Current State	Next State X=0	Next State X=1	Output (Z) X=0	Output (Z) X=1
A	B	C	0	0
B	D	E	0	0
C	F	G	0	0
D	H	I	0	0
E	J	K	0	0
F	L	M	0	0
G	N	P	0	0
H	A	A	0	0
I	A	A	0	0
J	A	A	0	1
K	A	A	0	0
L	A	A	0	1
M	A	A	0	0
N	A	A	0	0
P	A	A	0	0

The first thing to check for is identical states – ie states where the next state and outputs are the same – here they are H and I. Therefore, one can be removed with all references to I being replaced with a reference to H.

Then, you can check for states which you can't get to – ie do not appear in the next state columns. You should do this recursively, doing both steps and removing items.

Current State	Next State X=0	Next State X=1	Output (Z) X=0	Output (Z) X=1
A	B	C	0	0
B	D	E	0	0
C	E	D	0	0
D	H	H	0	0
E	J	H	0	0
H	A	A	0	0
J	A	A	0	1

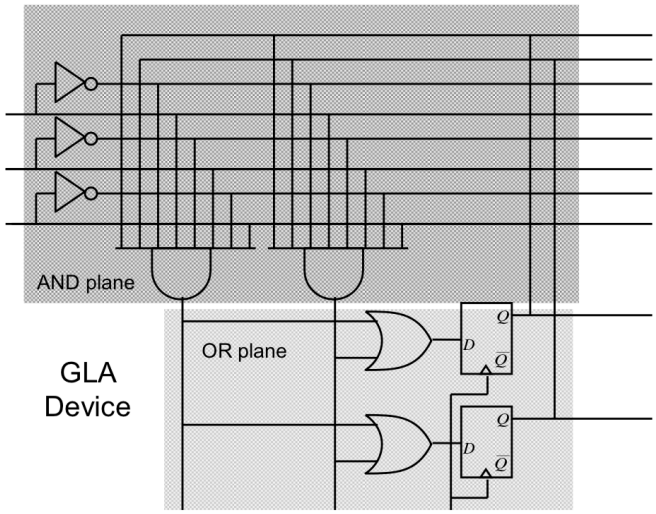
This procedure is known as **ROW MATCHING** and it is highly important to note that row matching is not sufficient to find all the equivalent states except for in special circumstances.

Implementation of FSMs

In the same way that programmable logic can be used to implement combinational logic circuits, using PLAs and PALs, they have been modified to use D-type FFs to permit FSMs to be implemented using programmable logic.

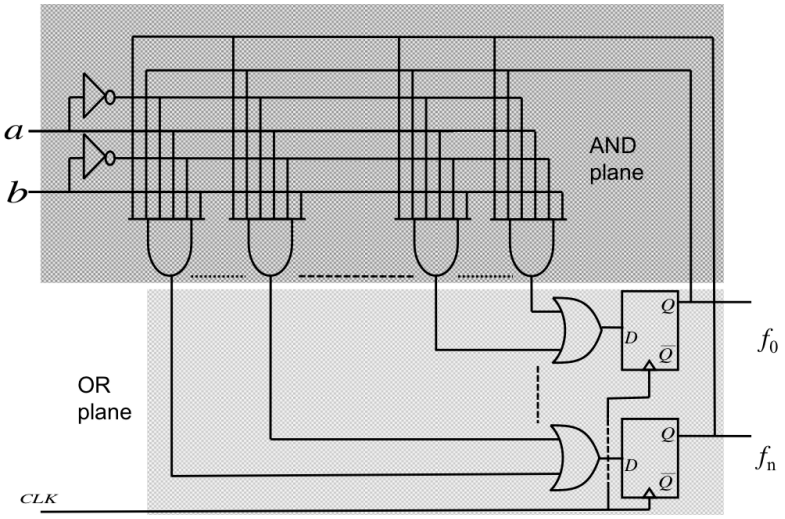
Generic Logic Array (GLA)

GLAs are similar to PLAs, but they have the option to make use of a D-type FF in the OR plane (one following each OR gate). In addition, the outputs from the D-types are made available to the AND plane (in addition to the usual inputs). This means it is possible to build programmable sequential logic circuits.



Generic Array Logic

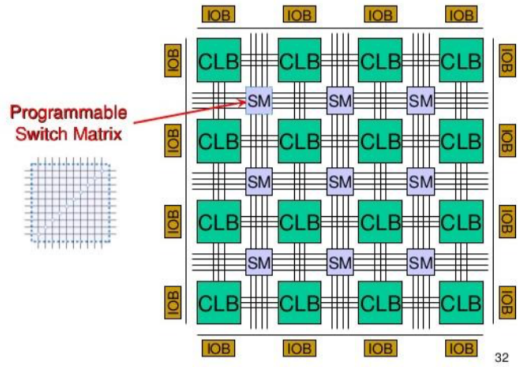
A GAL is a modification of a GLA where the OR Plane is not programmable.



Field Programmable Gate Arrays (FPGAs)

They are an array of configurable logic blocks (CLBs) surrounded by Input Output Blocks (IOBs).

- CLBs contain look up tables (LUTs), multiplexers and D-type FFs.
- Programmable routing channels permit CLBs to be connected to other CLBs and to IOBs
    - **Programming Routing Channels work using wires plus programmable switch matrices.**



- The FPGA is configured by specifying the contents of the LUTs and select signals for the MUXs

FPGA – Spartan CLB

- Has 2, 4-input LUTs and 1, 3 input LUT.
- Has two combinational outputs and 2 registered outputs (from D-type FFs)
- Depending on the MUX configuration, the outputs are either given from specific LUTs.
- The D-type FF inputs come from one of the LUTs or a specific Din input.
- Therefore, each CLB can perform up to 2 combination and / or 2 registered functions
- **All functions can involve at least 4 input variables (but maybe upto 9)**
- **Synthesis Tool**
    - The synthesis tool determines how the LUTs, MUXs and routing channels are configured
    - This configuration information is then downloaded to the FPGA.
    - The Xilinx devices (the ones being described) store their information in SRAM so it is easy to reprogram.

FPGAs Differences Between Manufacturers

- Many other manufacturers are available – Altera is one example
- Main differences will be
    1. Number of CLBs
    2. Structure of CLBs (number of LUTs, MUXs, etc)
    3. Internal or External ROM
    4. Additional features such as specialised arithmetic blocks
    5. User RAM

Electronics, Devices and Circuits

Basic Electricity

- An electric current is produces when charged particles move in a definite direction.
- In metals, the outer electrons are held loosely by their atoms and are free to move around the fixed positive metal ions.
- This free electron motion is random and so there is no net flow of charge in any direction.
- However, if a metal wire is connected across the terminals of a battery, the battery acts as an ‘electron pump’ and forces the free electrons to drift towards the +ve terminal and in effect flow through the battery
    - This annoys me. In reality, the electrons are drawn by a electrostatic potential between the positive anode of the positive terminal of the battery and the negatively charged electron.

    - **The drift speed of the free electrons is low, 1mm/s due to frequent collisions with the metal ions.**
    - However, electrons all flow together when the potential is applied (therefore there is a current).
- The flow of electrons in one direction is known as an electric current and reveals itself by making the metal warmer and by deflecting a magnetic compass.
    - Conventional current is +ve to –ve
- Current is when charges moves past a point.
    - 1A = 1C of charge moving passed a point in a circuit per second
    - 1C = 6.25 x 10^{18} electrons
    - Q = It
- **EMF:** The electromotive force of a battery is said to be 1V if it gives 1 Joule of electrical energy to each coulomb of charge passing through it
    - **E=VQ**
- **Potential Difference is** defined to be 1V if it changes 1 Joule of electrical energy into other forms of energy when 1C of charge passes through it.
- *PD and EMF are both voltages since both are measured in volts.*
- **Power = VI**

Basic Materials

- The electrical properties of materials are central to understanding the operation of electronic devices.
- Their functionality depends on our ability to control properties such as their IV characteristics.
- Whether a material is a conductor or insulator depends on how strongly the outer valence electrons are bound to their atomic cores.

Insulators

On a crystalline insulator, electrons are strongly bound and unable to move. When a voltage difference is applied, the crystal will distort a little, but no charge will flow until the material breaks down.

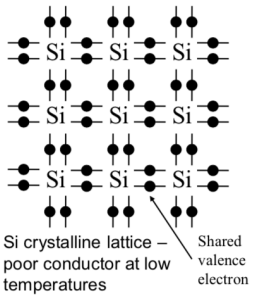
Conductors

In conductors, the electrons are weakly bound and free to move. When a voltage difference is applied, the crystal will distort a bit, but more importantly, charge will flow through the movement of electrons.

Semiconductors

Since there are a lot of free electrons in a metal it is difficult to control its properties. Therefore, using a material with a low electron density is better (ie a semiconductor). By carefully controlling the electron density, we can create a whole range of electronic devices.

Silicon is a poor conductor of electricity:



Si crystalline lattice – poor conductor at low temperatures

Si is *tetravalent*, i.e., it has 4 electrons in its *valance* band

Si crystals held together by *'covalent'* bonding

Recall that 8 valence electrons yield a stable state – each Si atom now appears to have 8 electrons, though in fact each atom only has a half share in them. Note this is a much more stable state than is the exclusive possession of 4 valence electrons

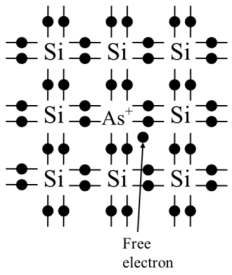
However, as the temperature rises, thermal vibration of the atoms causes bonds to break. This means electrons are free to move around the crystal. When an electron breaks free (moves into the conduction band), it leaves behind a hole (or absence of negative charge). The hole can appear to move if it is filled by an electron from an adjacent atom.

n-type Si

n-type Silicon is doped with arsenic that has an additional electron that is not involved in the bonds to the neighbouring Silicon atoms. The additional electron needs only a little energy to move into the conduction band.

Owing to its negative charge, the resulting semiconductor is known as n-type.

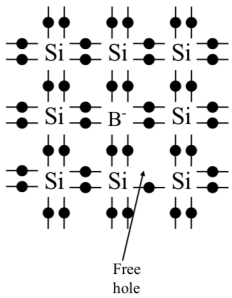
Arsenic is known as a donor since it donates an electron.



p-type Si

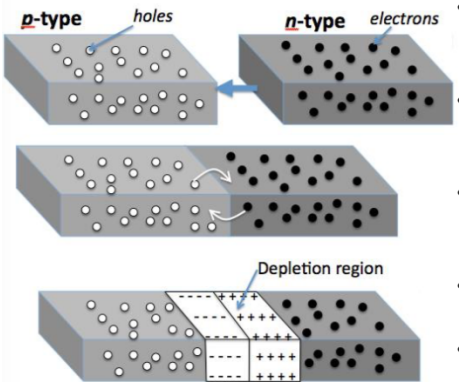
p-type Silicon is doped with Boron. The Boron atom has only got 3 valence electons – accepts an extra electron from an adjacent silicon atom to complete its covalent bonding. This leaves a hole in the lattice which is 'free to move'. This free hole has a positive charge.

Boron is known as an acceptor



p-n junctions

- The key to building useful devices is combining p and n type semiconductors.
- Here, electrons and holes diffuse across the junction due to large concentration gradients – electrons move to fill holes.
- On the n-side, diffusion out of electrons leaves +ve charge donor atoms.
- On the p-side, diffusion out of holes leave –ve charged acceptor atoms.
- This leaves a space-charge (depletion) region with no free charges.
- The space charge **gives rise to an electric field that opposes diffusion.**
- Equilibrium is reached when no more charges move across the junction.
- The PD associated with the field is known as the **contact potential**



Reverse Biased p-n Junction

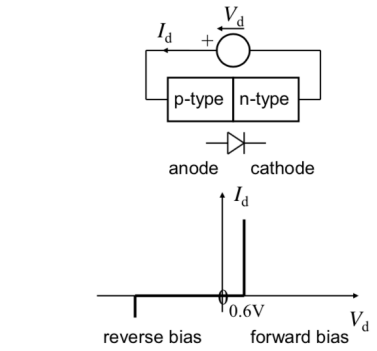
By applying a voltage across the junction, making the n-type +ve, electrons are removed from it, increasing the size of the space charge region. Similarly, holes are removed from the p-type region. Therefore, the space charge region and the associated field are increased.

The net current is known as the **reverse saturation current.**

Forward Bias

- With forward bias, the holes on the p-side are pushed towards the junction where they neutralise some of the –ve space charge.
- Similarly on the n-side, the electrons are pushed towards the junction and neutralises some of the +ve space charge.
- So depletion region and associated field are reduced.
- **Therefore, the diffusion current increases significantly and has the order of mA.**
- **Therefore, the p-n junction allows significant current flow in only one direction.**
- **This device is known as a diode.**

Diode – Ideal Characteristic



Circuit Theory

The opposition to current is called resistance and is measured in ohms. The larger the resistance, the larger the potential difference measured across the conductor (for a given current).

Ohm's Law

- R = V/I
- Only true for Ohmic resistors where eventually plotting I against V yields a straight line through the origin.

Kirchhoff's Current Law

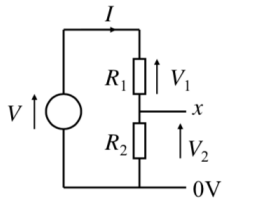
The sum of currents entering a junction (or node) is zero.

Kirchhoff's Voltage Law

In a closed loop of an electric circuit, the sum of all the voltages in that loop is zero.

Potential Divider

Finding the voltage at point x:



Note: circle represents an ideal voltage source, i.e., a perfect battery

$$V = V_1 + V_2$$

$$V_1 = IR_1 \qquad V_2 = IR_2$$

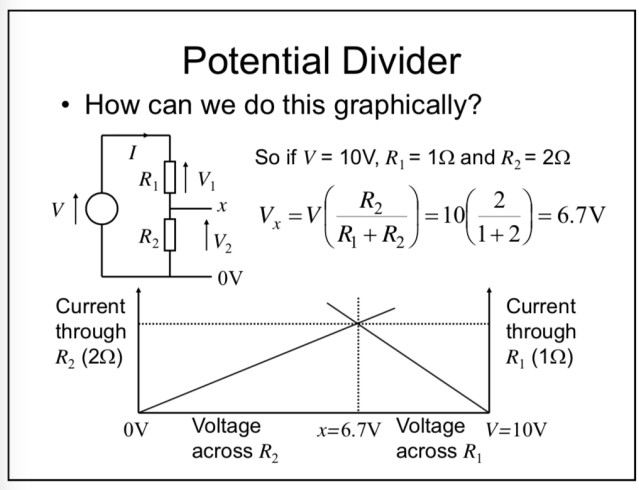
$$V = IR_1 + IR_2 = I(R_1 + R_2)$$

$$I = \frac{V}{(R_1 + R_2)}$$

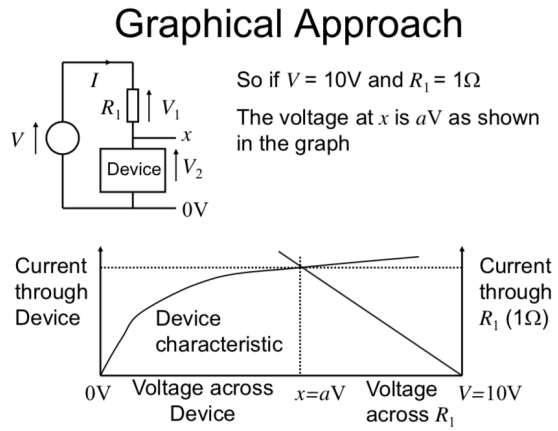
$$V_x = V_2 = \frac{V}{(R_1 + R_2)} R_2 = V\left(\frac{R_2}{R_1 + R_2}\right)$$

Solving Non-Linear Circuits

Not all devices have linear I-V characteristics, importantly this includes the FETs used to build logic circuits. Therefore we cannot use the algebraic approach of the potential divider, therefore we must look at the graphical approach.



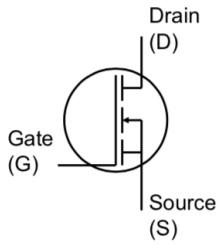
In the case of a non-linear device, we simply substitute the V-I characteristic of the non-linear device in place of the linear characteristics used previously for R2.



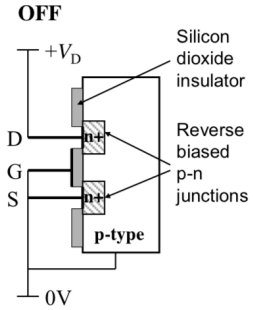
MOSFETs

n-channel MOSFETs

A MOSFET is a Metal Oxide Semiconductor Field Effect Transistor controls the current flow from a **drain** to a **source,** controlled by the voltage applied between **the gate and the source.**



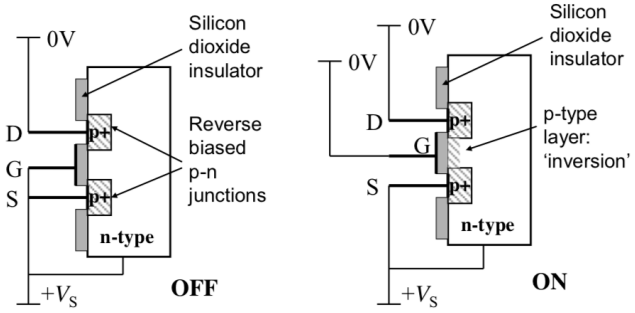
The structure of a n-Channel MOSFET is like this:



Initially, the Drain (and Source) diode is reverse biased, so there is no path for current to flow from S to D, therefore the transistor is off. However, when the gate voltage is raised to a positive value, electrons are attracted to the underside of the G plate, so this region is 'inverted' and becomes n-type A **channel** is found between the junctions. There is now a continuous flow between the channels so then transistor is formed. The Gate Voltage needed for the channel to be created is known as the **threshold voltage.** Typically, it is around 0.3 to 0.7V.

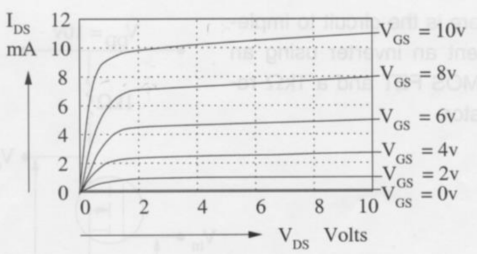
p-channel MOSFETs

A p-channel has the opposite construction, ie n-type substrate and p-type S and D regions.

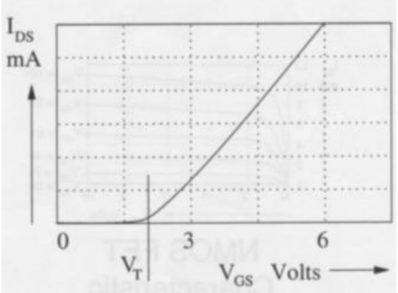


n-MOS Logic

Plot of V-I characteristics of the n-MOSFET for various Gate Voltages:

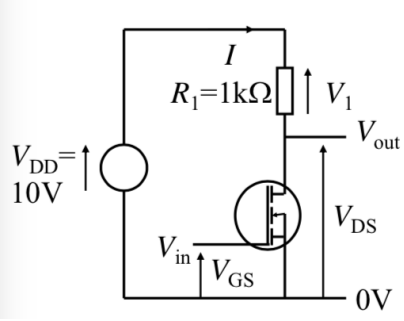


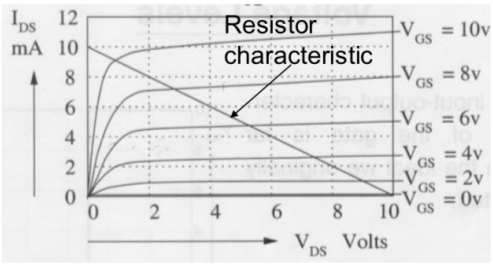
At a constant value of Vds, we can also see that Ids is a function of the gate voltage. The transistor also only begins to conduct when the gate voltage reaches the threshold voltage.

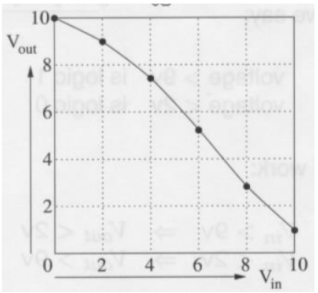


n-MOS Inverter

We can define a n-MOS inverter using the following circuit, using the graphical approach to find the relationship between Vin and Vout.







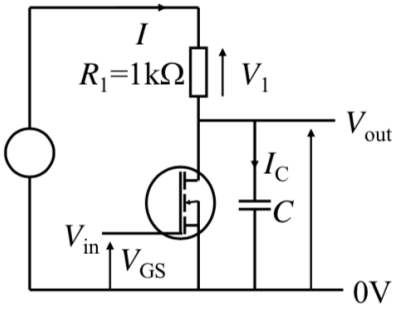
Note $V_{in}=V_{GS}$ and $V_{out}=V_{DS}$

While this would work by specifying suitable voltage thresholds, it does not have the ideal 'inverter' characteristic.

Using n-MOS logic, it is possible (and was done) to build other logic functions (NOR and NAND) using n-MOS transistors.

However, n-MOS logic has problems:
1. Speed of operation
2. Power consumption

One of the major speed limitations is **due to the stray capacitance, ie an inverter becomes something like this:**



Capacitors

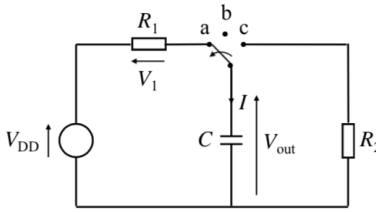
A Capacitor is a device that stores electrical charge using 2 conductors, separated by a non-conductor (or dielectric). Therefore, any parallel conductors brought close together (1/n relationship) will form a capacitor. These parallel conductors often appear on circuit boards, therefore creating parasitic capacitors. These can have a negative impact on the switching characteristic of digital logic circuits.

Q = VC

Or

$$V = \frac{1}{C}\int Idt$$

Charging Capacitor



$$V_{DD} - V_1 - V_{out} = 0$$

$$V_{DD} = V_1 + V_{out}$$

Sub for V_{out} in terms of I and C (from earlier eqn.)

$$V_{DD} = IR_1 + \frac{1}{C}\int Idt$$

Differentiate wrt t gives

$$0 = R_1\frac{dI}{dt} + \frac{I}{C}$$ Then rearranging gives $$-\frac{dt}{CR_1} = \frac{dI}{I}$$

Integrating both sides of the previous equation gives

$$-\frac{t}{CR_1} + a = \ln I$$

We now need to find the integration constant a .

To do this we look at the initial conditions at t = 0, i.e., V_{out}=0. This gives an initial current I_0=V_{DD}/R_1

$$a = \ln I_0 = \ln\left(\frac{V_{DD}}{R_1}\right)$$

So,

$$-\frac{t}{CR_1} + \ln I_0 = \ln I$$

$$-\frac{t}{CR_1} = \ln\frac{I}{I_0}$$

Antilog both sides,

$$e^{-t/CR_1} = \frac{I}{I_0}$$

$$I = I_0 e^{-t/CR_1}$$

Now,

$$V_{out} = V_{DD} - V_1$$

and,

$$V_1 = IR_1$$

Substituting for V_1 gives,

$$V_{out} = V_{DD} - IR_1$$

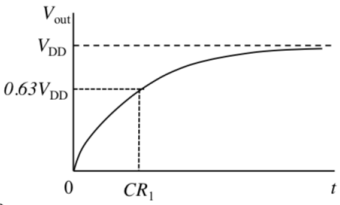
$$V_{out} = V_{DD} - R_1 I_0 e^{-t/CR_1}$$

Substituting for I_0 gives,

$$V_{out} = V_{DD} - R_1\frac{V_{DD}}{R_1}e^{-t/CR_1}$$

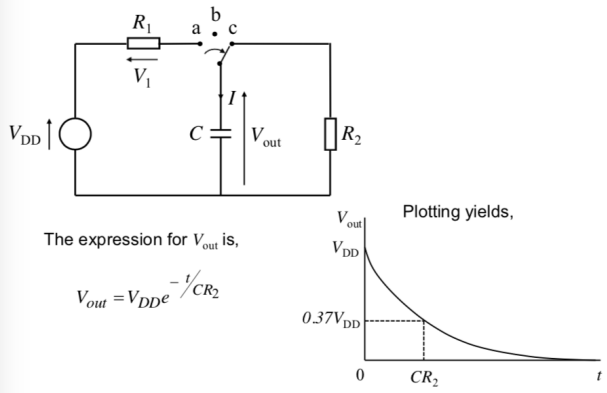
$$V_{out} = V_{DD}\left(1 - e^{-t/CR_1}\right)$$

Plotting yields,



CR_1 is knows as the time constant – has units of seconds

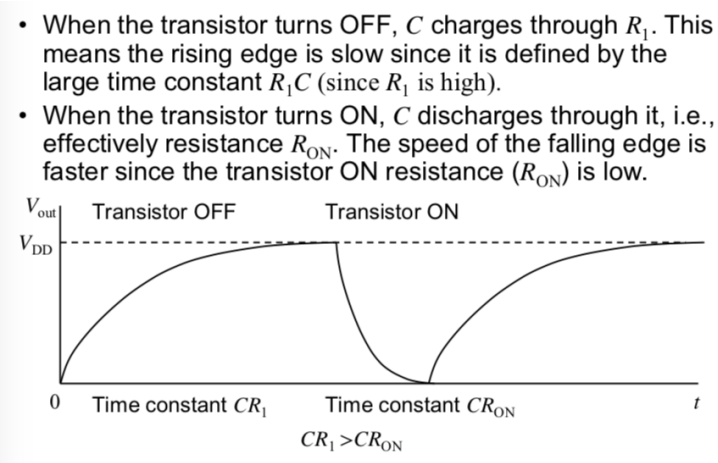
Discharging a Capacitor



n-MOS Inverter with Stray Capacitance

When the transistor is turned from OFF to ON, the circuit is effectively an open RC circuit, therefore the capacitor gets charged. (The general problem with the capacitor is that the voltage across it cannot change instantaneously).

Then when the transistor is turned on, it is effectively a low value resistor. Here therefore, the capacitor discharges through the resistor.



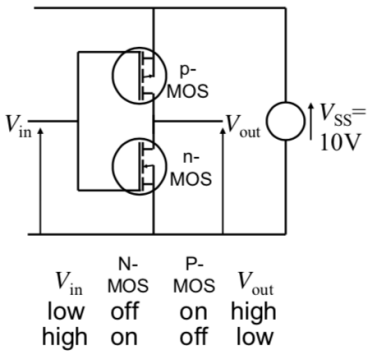
Power Consumption

When the transistor is on, there is always current flowing through the resistor, therefore there is power dissipated in the resistor.

CMOS Logic

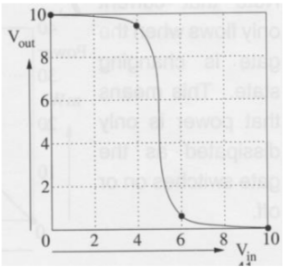
Complementary Metal Oxide Semiconductor Logic: Utilises both p-channel and n-channel MOSFETs.

CMOS Inverter

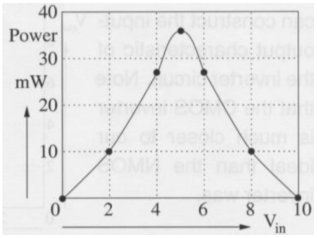


V_{in}	N-MOS	P-MOS	V_{out}
low	off	on	high
high	on	off	low

Using the graphical approach, we can show that the switching characteristic are much better than for the n-MOS inverter.



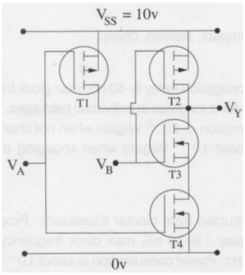
It can also be shown that the transistors only dissipate power while they are switching. This is when both transistors are on. When one or the other is off, the power dissipation is zero. CMOS is also better at driving capacitive loads, since it has active transistors on both rising and falling edges.



CMOS Gates

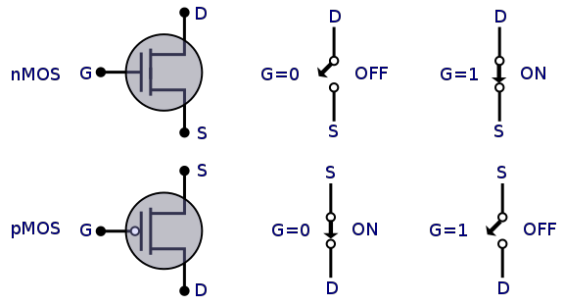
CMOS can also be used to build NAND and NOR gates. NAND gates use p-channel MOSFETs in parallel and n-channel MOSFETs in series. NOR gates use them in the other way (p-channel MOSFETs in series and n-channel MOSFETs in parallel).

NAND Gate

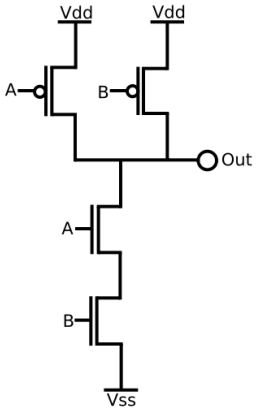


V_A	V_B	T1	T2	T3	T4	V_Y
low	low	on	on	off	off	high
low	high	on	off	on	off	high
high	low	off	on	off	on	high
high	high	off	off	on	on	low

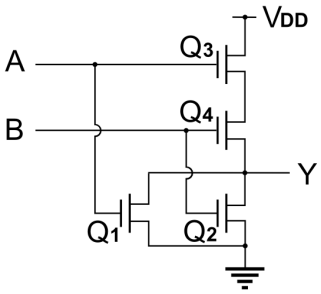
(Another way of drawing p-channel and n-channel MOSFETs was in this way – this is arguably much easier to draw)



A NAND gate drawn with these symbols looks like this:



NOR Gate



Logic Families

NMOS – Compact, slow, cheap and obsolete
CMOS – Older families are slow (4000 series are about 60ns), but new ones (74AC) are much faster (3ns). The new series are very popular.
TTL – Uses bipolar transistors (2 PN junctions back to back, with 2 consecutive P junctions or 2 consecutive N junctions). They are known as 74 series – most of which are available in CMOS. The new versions are slow (16ns) but the newer ones are faster (2ns).
ECL – High speed, but high-power consumption.

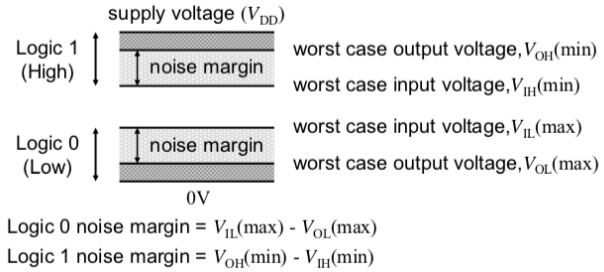
It is generally best to stick with the particular family which has the best **(1) performance, (2) power consumption & (3) cost** trade-off for the required purpose. While it is possible to mix logic families and sub-families, but care is required regarding the acceptable logic voltage levels and gate current handling capabilities.

Voltage Levels

The relationship between the input voltages to a gate and the output voltage depends upon the particular implementation technology.

The signals between outputs and inputs are **analogue** and so are susceptible to corruption by additive noise (due to cross talk from signals in adjacent wires).

Noise Margin



Logic 0 noise margin = V_{IL}(max) - V_{OL}(max)

Logic 1 noise margin = V_{OH}(min) - V_{IH}(min)

Example of Noise Margin for the 74 series High Speed CMOS (HCMOS)

Logic 0 noise margin = V_{IL}(max) - V_{OL}(max)

Logic 0 noise margin = 1.35 – 0.1 = 1.25 V

Logic 1 noise margin = V_{OH}(min) - V_{IH}(min)

Logic 1 noise margin = 4.4 – 3.15 = 1.25 V

See the worst case noise margin = 1.25V, which is much greater than the 0.4 V typical of TTL series devices. Consequently HCMOS devices can tolerate more noise pick-up before performance becomes compromised

ADC

Digital Electronic Systems often need to interface to the analogue real world. For example:

- To convert an analogue audio signal to a digital format we need an ADC.
- Similarly, to convert a digitally represented signal we need to use a DAC.

ADC is a 2-stage process:

1. Regular sampling to convert the continuous time analogue signal into a signal that is discrete in time.
2. These sample values can still take continuous amplitude values; hence the next step is to represent them using only discrete values in the amplitude domain. To do this, the samples are quantised in amplitude (they are constrained to take one of only M possible amplitude values).
    a. Each of these discrete amplitude values is represented by an n-bit binary code.

Thus, the ADC process introduces **quantisation error** owing to the finite number of possible amplitude levels that can be represented. The greater the number of quantisation levels, the lower the quantisation error, but at the cost of a higher bit rate.

In addition, the sample rate (1/T) must be at least twice the highest frequency in the analogue signal being sample – **known as the Nyquist rate**. **To ensure this happens, the analogue**

signal is passed through a low pass filter that will remove frequencies above a specified maximum. If the Nyquist rate is not satisfied, **the sampled rate will be subject to alias distortion that cannot be removed and will be present** in the reconstructed analogue signal.

The ADC also requires that the input signal is in amplitude range. Usually, ADC has a range covering several Volts, while the signal from the transducer can be of the order of mV. Therefore, amplification is usually required before being inputted into the ADC. If not, the digitised signal will have poor quality (a low signal to quantisation noise ratio).

Operation amplifier based 'Gain block' in front of the ADC are often used since they have predictable performance and are easy to use.

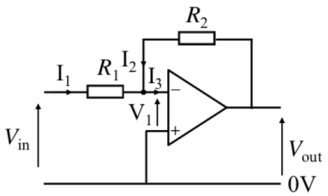
DAC

- A DAC is used to convert the digitised sample values back to an analogue signal.
- A low pass filter usually follows the DAC to yield a smooth continuous time signal.
- Operational Amplifier based buffer amplifiers are used after the DAC to prevent the load (transducers such as headphones) affecting the operation of the DAC.

Op-Amps

- Op-Amps have 2 inputs and a single output. They can be configured to implement gain blocks, filters, summing blocks etc.
- When looking at op-amps, we always assume there is an ideal op-amp, which has an infinite input resistance (zero input current) and infinite gain.

Inverting Amplifier



$I_1 + I_2 - I_3 = 0$

Now, $I_3 = 0$ since the input resistance of the op amp is ∞, so

$I_1 = -I_2$

$V_{in} - I_1 R_1 - V_1 = 0$ and

$V_1 + I_2 R_2 - V_{out} = 0$

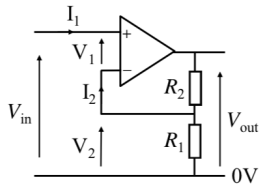
Now, $V_1 = 0$ since the op - amp has ∞ gain (virtual earth assumption)

$V_{in} = I_1 R_1$ and $V_{out} = I_2 R_2 = -I_1 R_2$

So, $I_1 = -\frac{V_{out}}{R_2}$

Yielding, $V_{in} = -\frac{V_{out} R_1}{R_2}$ Voltage gain, $\frac{V_{out}}{V_{in}} = -\frac{R_2}{R_1}$

Non-Inverting Amplifier



Now, $I_1 = I_2 = 0$ since the input resistance of the op amp is ∞, so

$V_2 = V_{out} \frac{R_1}{R_1 + R_2}$

$V_{in} - V_1 - V_2 = 0$

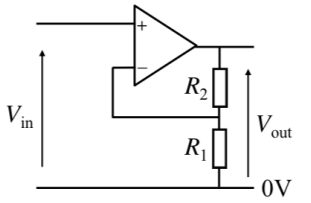
Now, $V_1 = 0$ since the op - amp has ∞ gain (virtual earth assumption)

$V_2 = V_{in}$

So, $V_{in} = V_{out} \frac{R_1}{R_1 + R_2}$

Yielding, $\frac{V_{out}}{V_{in}} = \frac{R_1 + R_2}{R_1}$ Voltage gain, $\frac{V_{out}}{V_{in}} = 1 + \frac{R_2}{R_1}$

Buffer Amplifier (Unity Gain)

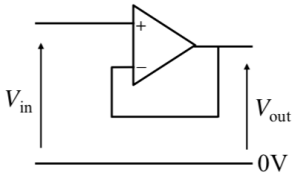


We know the voltage gain for the non-inverting amplifier is,

$$\frac{V_{out}}{V_{in}} = 1 + \frac{R_2}{R_1}$$

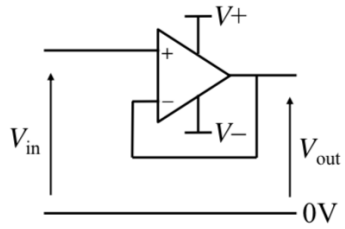
Now, if we let $R_2 = 0$ (a short circuit) and $R_1 = \infty$ (open circuit) then

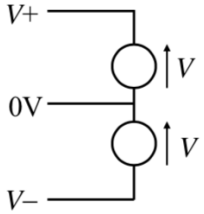
$$\frac{V_{out}}{V_{in}} = 1$$



The purpose of a buffer amplifier is to isolate one circuit from the load of another circuit – they should not affect one another.

Power Supply for Op-Amps





In order to allow both +ve and –ve signals to be amplified, op-amps generally use a split power supply.
However, this can be inconvenient for battery powered equipment. If the input signal is set to be always +ve then the V- rail can be set to 0V.

Applications

1. Filters
    a. Circuits that manipulate the frequency content of signals.
2. Mathematical functions
    a. Integrators and differentiators.
3. Comparators and triggers
    a. Thresholding devices