P2 Digital Electronics

Part 2: Lectures E-H

Mark Cannon, Trinity Term 2026 mark.cannon@eng.ox.ac.uk

Notes originally prepared by Prof. Chris Stevens, revised by Prof. Martin Booth, Prof. Perla Maiolino and Prof. Mark Cannon

Contents

1 Lecture E – Multiplexers, ROMs and PLAs

1.1 Introduction to electronic integration

So far, we have looked at simple circuits with only a few gates, considering how they are connected together to perform a particular logical function. Modern-day digital electronics is not like this. Electronic engineers normally develop complex circuits using sophisticated design tools, and are increasingly moving away from schematic diagrams of interconnected gates towards high level software languages[1].

It is more cost-effective for circuit designers to rely on the preciseness of a computer language for describing their designs rather than drawing large numbers of wires and transistors. With this approach, entire sections of complex circuitry can be brought together into a large-scale integrated circuit.

Outside of the major manufacturers of large-scale integrated circuits such as Intel, there is still a need to go down the traditional route of designing a printed circuit board and building up the circuit complexity by connecting relatively simple integrated circuits together. These simple building blocks may be groups of NAND gates, flip-flops, simple arithmetic units, logic arrays, etc. The progression is from "small-scale integration (SSI)" to "medium scale integration (MSI)" and then onto "very large scale integration (VLSI)" (where the Intels of this world operate) and towards ULSI. In this lecture, we will consider medium scale integrated circuits. A rough definition of MSI chips would be a single integrated circuit containing up to a hundred gates or so, which might provide the basic building blocks for connecting large sub-units in a computer together.

1.2 Multiplexers (MUX)

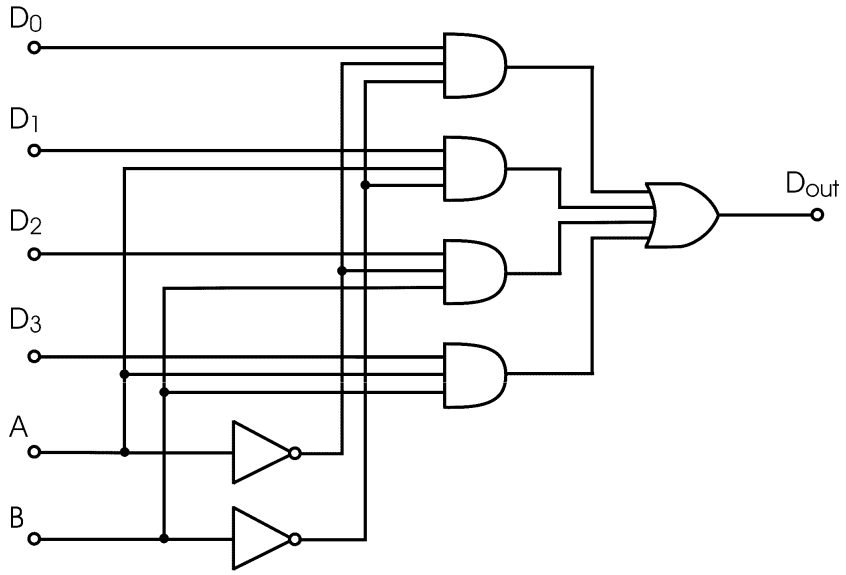
Very often in electronic circuits it is necessary to route just one particular data path onto the next stage in the circuit. A multiplexer is a circuit component that performs this task. It has a number of data input lines, usually a single

[1] An example is VHDL, which describes operations on data rather than the hardware to achieve them, then allows automated derivation of that hardware.

data output line and the appropriate number of bits for selecting which input is directed towards the output. It is analogous to a telephone exchange, whereby the data that gets through to a destination is determined by the switch settings at any of the multiplexers present in the data path.

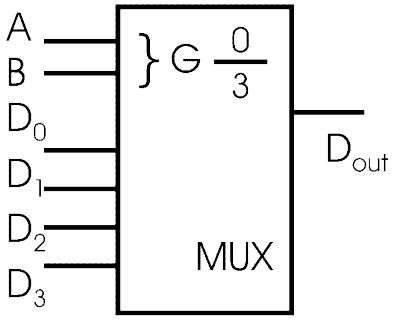
The following circuit is a typical multiplexer where one of four inputs, D_0, D_1, D_2, D_3 are chosen by setting the inputs A and B according to the logic

$$D = \overline{A}.\overline{B}.D_0 + A.\overline{B}.D_1 + \overline{A}.B.D_2 + A.B.D_3 \tag{1.1}$$



Four-input multiplexer

The standard way of representing such a multiplexer in a schematic circuit is shown below. Note that the lines A and B can be considered as address lines, represented collectively in this case as G. The 0/3 notation refers to the addressable space, i.e. 0-3 selectable lines. De-multiplexers are circuits which perform exactly the reverse function.



Schematic circuit for a four-input multiplexer

1.3 Read only memory (ROM)

A Read Only Memory is used for permanent data storage. The data to be stored in a ROM must be decided before it is manufactured as the binary data is encoded into the photolithographic masks that are used in semiconductor device fabrication. As this is an expensive process it is only really economical for mass production. It is also important that the stored data has been proven to be correct before mass production begins.

PROMs are programmable ROMs where the storage elements are fusible links that can be "blown" by applying electrical voltages to the device terminal (when in *write* mode). Once these links have been fused, the data is fixed and cannot be changed. These components are relatively cheap and ideal for designs with a smaller target market.

EPROMs, "Erasable PROMs", are a step further. They tend to rely on charge storage effects in MOSFETs which act in a similar fashion to the fused links in the ROM. When the EPROM is exposed to UV light, the data can be erased and the EPROM re-programmed. EEPROMs or Flash-EPROMs are electrically erasable devices and are ideal for prototyping software or retaining information that changes infrequently. All of these devices can be referred to as ROMs.

1.3.1 Structure of a ROM

A ROM is a look-up table, with every data entry considered to have a unique identifiable address. If the data held in the ROM corresponds to a 16-bit word, then the data output line is 16 bits wide. If the ROM contains a total of 1024 such data words then it must have sufficient input lines to be able uniquely to select any one out of the 1024, i.e. the ROM must have a 10-bit address. Thus, if a ROM has n input lines and m output lines, then it will have 2^n separate addresses, with an m-bit word stored at each address. Within the ROM, all possible address values must be decoded which, for some applications, might be an inefficient approach.

As a trivial example, consider a 3-input, 2-output ROM. The inputs are labelled

A_i, $i = 0, 1, 2$, and the outputs are labelled D_i, $i = 0, 1$. There are therefore 2^3 possible input addresses, each corresponding to a 2-bit output. The table here is an example in which the 2-bit data stored are 10, 11, 01, 01, 10, 00, 10 and 11, starting at address 0 and finishing at address 7.

A_2	A_1	A_0	D_1	D_0
0	0	0	1	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	1	0
1	1	1	1	1

Map of the contents of an 8 address, 2-bit ROM

The above is a map of the memory contents of the chip. The columns D_0 and D_1 may be considered either as a 2-bit stored word, or as a pair of logic functions:

$$D_0 = \overline{A_2 A_1} A_0 + \overline{A_2} A_1 \overline{A_0} + \overline{A_2} A_1 A_0 + A_2 A_1 A_0$$
$$D_1 = \overline{A_2 A_1 A_0} + \overline{A_2 A_1} A_0 + A_2 \overline{A_1 A_0} + A_2 A_1 \overline{A_0} + A_2 A_1 A_0$$

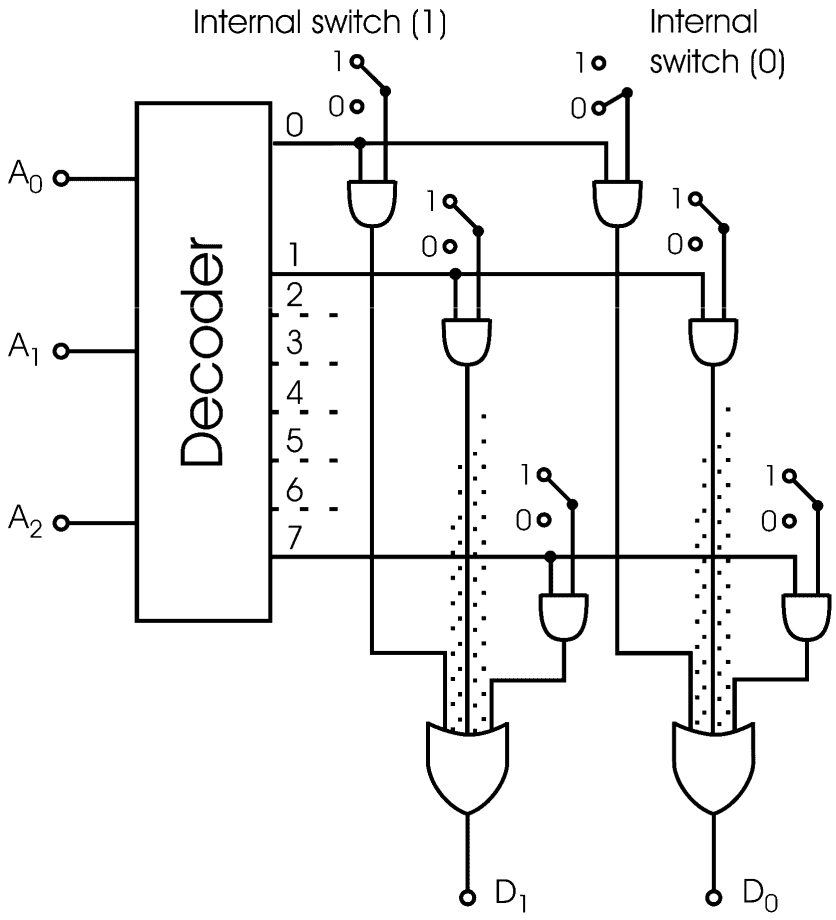
Here the logical expressions for D_0 and D_1 are given in their sum of products form (each product term being referred to as a minterm – see Lecture B). Each of the minterms must be decoded within the ROM and its contents configured to provide the relevant data. We can thus break down the internal structure of the ROM into three components:

1. A Decoder: This has n input lines and 2^n output lines. Decoder line i is set to logic 1 when the input lines encode the binary number i.

2. A $2^n \times m$ matrix of switches that store the individual bits (i.e. the ROM contents). Row i of the matrix is switched on by line i from the decoder.

3. An output stage which ORs the outputs from each column of the switch matrix to generate the required output functions.

An illustration of the workings of a ROM is given in the figure below.



Output matrix for a simple ROM or EEPROM

The figure shows a 3-to-8 line decoder which handles the addressing. Only the lines and switches corresponding to decoder lines 0, 1 and 7 are shown - the others are represented as dots. The details of how the internal switches and the output section are built are outside the scope of the course, but for those interested a possible realisation using a "wired-OR" implementation is shown in Figure 7.20 in Hill & Peterson and can also be found in other text books.

1.3.2 The decoder

It is possible to buy ROMs with a large number of inputs, say greater than 12. This results in a very complex decoder with a large number of gate inputs. The example given in Hill & Peterson describes a 12-input ROM – a device with 4096 addresses. The decoder would then require 4096 12-input AND gates to implement all the minterms.

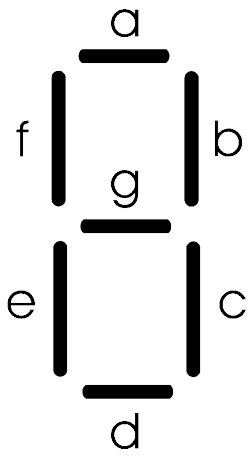
A more efficient decoding strategy is to break the input lines into smaller groups, say three inputs per group, and then further decode the resulting groups. (A detailed account of this is given e.g. in Hill & Peterson Section 7.3, particularly Figure 7.17).

1.3.3 Uses of ROMs

As ROMs store data in permanently configured hardware, the data is regarded as being *non-volatile*. This means that the data remains uncorrupted when power is lost. Most computers contain boot PROMs, which contain the initial instructions that a computer must carry out when switched on. The data contained in such chips corresponds to very simple instructions that usually allow the computer to load its operating system from disk or a network.

As a ROM fully decodes a logic function into its minterms, i.e. it implements a truth table by brute force, it is generally considered to be an overkill to use one to implement a relatively simple logic function. However, its ability to store a large number of words of data means that it is an efficient code converter.

A common example used is that of the truth table for a BCD to seven-segment conversion. A seven-segment display is the typical display seen on digital watches or cookers that display digits by lighting (or blocking out) the correct bars from the pattern of 8 shown in the figure below. A ROM is thus able to take the output from a simple counter and drive a complex display device directly.



7-segment display configuration

1.3.4 ROM programming example

A 3-bit counter cycles continuously through the sequence

$$0, 1, 2, 3, 4, 5, 0, 1, 2, \ldots$$

changing to the next number every 10 seconds. Clearly, the 3-bit counter can count from 0 to 7 (8 values encoded by 3 bits), but the numbers 6 and 7 are not used in this example. We will design a PROM program that has a 3-bit binary number as its input and drives a seven-segment output display. If the number 6 or 7 occurs, only the central segment lights up, indicating an error.

First, we need to work out which segment needs to be driven for each input count.

Count	Segments
0	a, b, c, d, e, f
1	b, c
2	a, b, g, e, d
3	a, b, c, d, g
4	f, g, b, c
5	a, f, g, c, d
6	g
7	g

We need a 3-input, 7-output PROM. Label the inputs as A_0, A_1, A_2 (address) and the outputs as a, b, c, d, e, f, and g (D_6 to D_0). Assume that the count is represented in straight binary. This then leads to the following truth table for the PROM:

A_2	A_1	A_0	a	b	c	d	e	f	g
0	0	0	1	1	1	1	1	1	0
0	0	1	0	1	1	0	0	0	0
0	1	0	1	1	0	1	1	0	1
0	1	1	1	1	1	1	0	0	1
1	0	0	0	1	1	0	0	1	1
1	0	1	1	0	1	1	0	1	1
1	1	0	0	0	0	0	0	0	1
1	1	1	0	0	0	0	0	0	1

1.4 Buses and tri-state outputs

A major factor in the cost of an integrated circuit is linked to the number of input and output pins that are required to achieve its functionality. Often the size of the silicon chip is dictated by the size of the peripheral frame that encloses the core circuitry. The peripheral frame contains the bond pads and protection circuitry for these interconnections to the outside world.

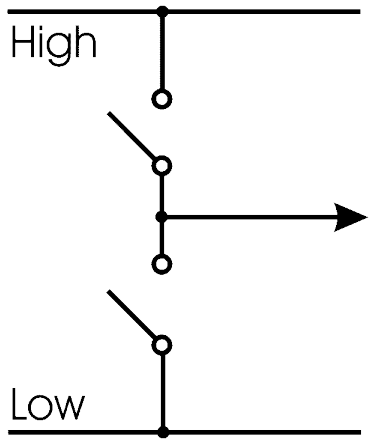
Similarly, in the design of a printed circuit board with many components, the routing of data and power lines can be an intractable problem. Even when utilising both sides of the printed circuit board for electrical connections, it is necessary for integrated circuits to share connections. These connections are called buses, and typically there are 8, 16, 32 or 64-bit buses for carrying data and address lines.

Bus conflicts will occur if several integrated circuits, connected to the same bus, try to change the logic values simultaneously. For example, a ROM chip and an ALU (Arithmetic Logic Unit) chip will probably share the data bus, but only one of these devices at any time should be allowed to output data to the bus. Think of what would happen if the ROM was driving all the data bus bits

to a logic 1, whilst the ALU was trying to put all logic 0's on the bus. The potential difference between these outputs would cause currents to flow and the actual logic values taken by the data bus would be undefined.

The simple solution to this problem is to have a third logic state, implemented in the hardware of the integrated circuit. Devices that use this approach are described as having Tri-state outputs. The output can then take one of the following states: logic 0, logic 1 or high impedance. When the output is in the high-impedance state, the integrated circuit is effectively isolated from the rest of the circuitry on the printed circuit board.

An illustration of how this functions is shown in the following figure. The actual implementation is achieved with MOSFETs if CMOS technology is being used (e.g. see Lecture A).



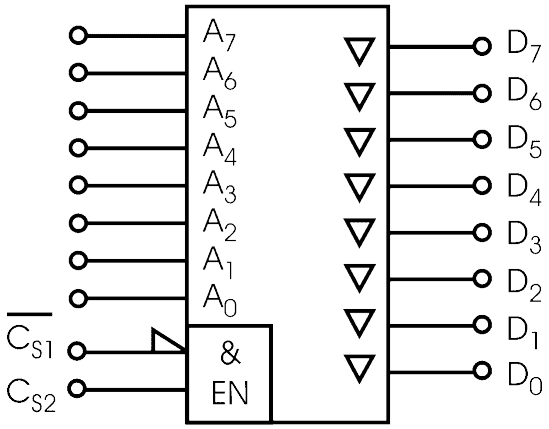
Diagrammatic version of a 3-state output – the third state is accessed by opening both switches

The figure above shows a representation of a tri-state logic output stage. The top switch being closed, with the bottom switch open, corresponds to a logic 1 at the output. The top switch open, with the bottom switch closed, corresponds to a logic 0 output. When both top and bottom switches are open, this corresponds to a floating output – the high- impedance state in which the gate does not affect the voltage of the output line.

1.5 Chip Select and Output Enable

Most complex logic chips such as ROMs or RAMs (Random Access Memory) have two control line inputs, one being labelled Chip Select, the other usually being labelled Output Enable. Chip Select can be thought of a turning on the input sections of the chip whereas Output Enable connects the output section to the external pins. Both Chip Select and Output Enable must be asserted for data to be read out of a memory chip.

The following figure shows a common circuit symbol for a 256 x 8-bit ROM.



A 256 address 8-bit ROM

The ROM symbol has small triangles drawn against its outputs – this indicates that these are tri-state outputs. The small square in the bottom left hand corner is the symbol for an AND function and indicates that the inputs at its side must both be true for the whole chip to function and for the tri-state outputs to exit the high impedance state.

The control input C_{S1} is active low, which means that it must be a low voltage to achieve its required function – this is indicated by the small triangle (sometimes a circle) drawn just above its input line and the bar above C_{S1}. Such active low signals are often used as chip selects. The other control input C_{S2} effectively acts as an output enable. If $\overline{C}_{\mathrm{S1}}$ has a low voltage applied to it and C_{S2} a high voltage applied to it, the ROM will turn on and drive the 8-bit data bus connected to outputs D_0 to D_7.

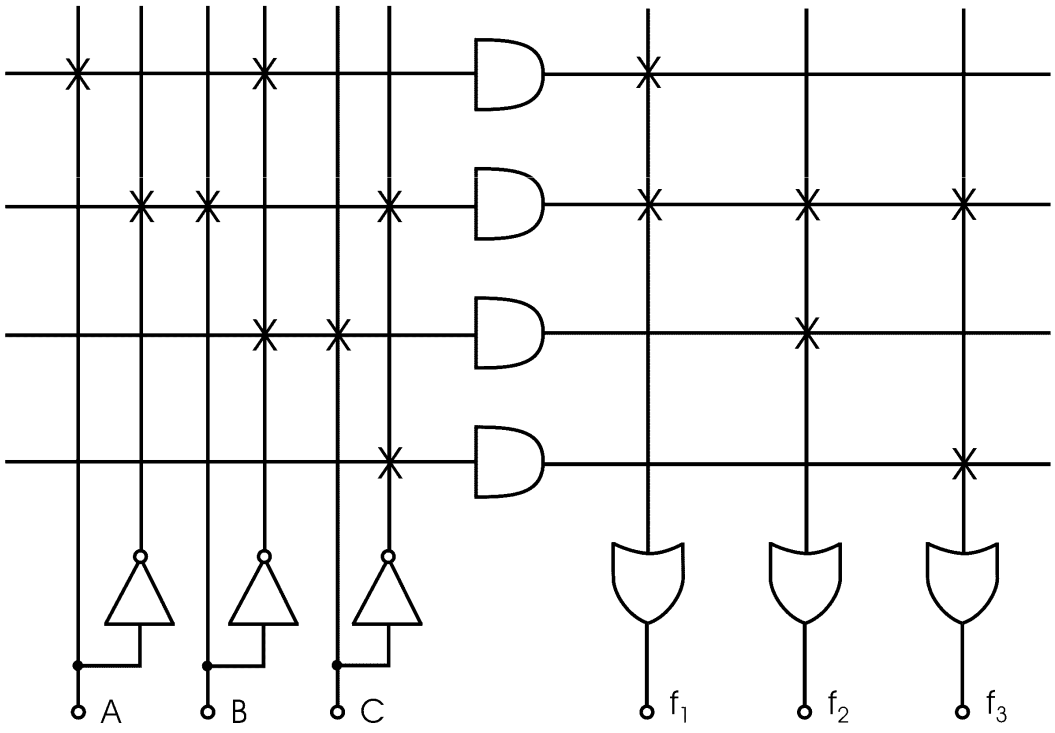
Chip Select is there to switch off the input decoding of the address lines (which

may be changing for other reasons). In CMOS circuitry, power is consumed only when the gates are changing state, so it is wasteful of power to allow the chip to be continually decoding the address inputs unnecessarily.

1.6 Programmable Logic Array (PLA)

It was stated earlier that a ROM is a highly inefficient implementation of a logic function. This is because a ROM has to decode all the address lines fully in order to map all the inputs to the outputs correctly. This can take several layers of gates, which is an inefficient use of silicon chip area. It adds to the heat dissipation and introduces unnecessary propagation delays.

A Programmable Logic Array differs from a ROM in that the input decoder is uncommitted and may be programmed using a matrix of fusible links. In order to program a PLA one must define both the decoder and output stage connections. The electrical connections are indicated by a set of X's placed on the input and output wire matrix. Only where an X occurs should it be assumed that an electrical connection exists. A simple decoder "program" is illustrated below.



Simple decoder implemented using a PLA

The above is a schematic representation of a programmed PLA. Each AND gate in the decoder may be driven by up to 6 inputs, formed from the inputs A, B, C, and their complements. A single line is drawn to represent each of these 6 inputs. Each OR gate has 4 inputs, each input corresponding to an AND gate in the decoder. The chip has 3 external inputs and 3 outputs. The chip thus provides three logic functions f_1, f_2 and f_3, each being functions of 3 external inputs. The number of AND gates in the decoder governs the complexity of the logic functions that can be obtained. If we needed to decode the input functions fully, the chip would need eight 6-input AND gates and would effectively be identical to a PROM (giving no saving).

The crosses indicate the connections chosen in this case, each cross denoting a connection. The logic functions implemented are thus:

$$\begin{aligned} f_1 &= A\overline{B} + \overline{A}B\overline{C} \\ f_2 &= \overline{A}B\overline{C} + \overline{B}C \\ f_3 &= \overline{A}B\overline{C} + \overline{C} \end{aligned} \tag{1.2}$$

Referring back to the figure, the region to the left of the AND gates is often referred to as the AND-plane, and that to the right as the OR-plane. The AND-plane essentially constructs all the necessary minterms and the OR-plane sums the required minterms to create the function.

The availability of cheap PLAs has given a huge boost to the production of electronic circuit boards, especially for small- to medium-sized companies. Essentially they allow large array circuits carrying a significant amount of discrete components to be shrunk into one or two chips, with all the logic functionality programmed into the array.

An extension of the PLA is the Field Programmable Gate Array (FPGA), the details of which are beyond the scope of this course. They are extremely popular as a circuit design approach that offers flexibility with a high level of sophistication. Normally a designer uses a computer package (XILINX is a key software supplier) to enter and check an electronic design, which is then programmed directly into the FPGA device using programming hardware connected to the computer's parallel port. Most modern FPGAs can be reprogrammed.

2 Lecture F – Memory, latches and clock mode circuits

2.1 Introduction

Up to this point all the circuits considered so far can be regarded as *combinatorial*, which means that the individual logic gates are not synchronised in any way. This can sometimes lead to glitches in the final output of the circuit if the paths to the final output have different numbers of layers of gates, and hence different delays (*race hazards*). This can be troublesome when designing large complex circuits that need to operate quickly.

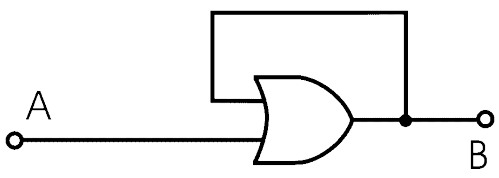
Control of the circuit *timing* is introduced through the use of a centralised *clock*, acting like a heartbeat for the whole of the circuitry. At first this might appear to slow everything down, but clock rates of several GHz are now realisable, which would have seemed unbelievable only a few years ago.

The starting point for designing circuitry that can be "clocked" is to introduce feedback, such that the output of a logic gate is connected to its inputs – which thus influences the output again. To analyse such a circuit's behaviour, the passage of time needs to be taken into consideration.

The idea of using feedback to ensure that the time evolution of a set of logic gates follows a desired sequence in response to external stimuli is fundamental to the design of digital computers. The analysis and design of such circuits will be the object of the rest of this course on sequential logic.

2.2 Memory – the SR latch

Consider an OR gate with feedback as shown below:



OR gate with feedback

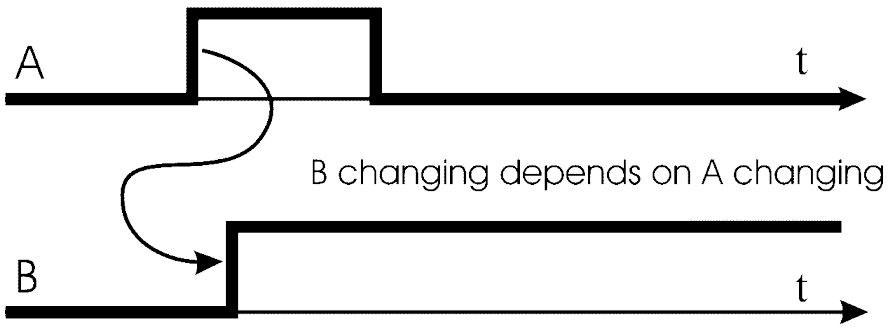
The equation governing the behaviour of this circuit is

$$B_{n+1} = B_n + A$$

where the $(n+1)$st point in time follows the nth point in time. The truth table associated with the circuit is as follows:

A	B_n	B_{n+1}
0	0	0
0	1	1
1	0	1
1	1	1

If it is assumed that the circuit has been powered up so that the output is 0 and the input is 0, then the output will stay at 0 for all time. If the input changes from 0 to 1 at any time, then the output will respond by changing from 0 to 1. However, once the output is 1, then the feedback ensures that one of the inputs to the OR gate is a 1. Thus, the output will stay at 1. There is thus an order of events that will lead to this circuit changing its output. The dependency on the order of changes in logic levels may be shown on a logic-timing diagram, which illustrates the edge dependencies of this circuit.



Timing diagram showing edge dependency – in a real circuit there will be a short delay between A and B changing

This circuit has a single binary output and therefore just two output states, i.e. 0 or 1. It is an example of a very simple *latch*, a circuit used to store an event. The output of the latch is said to be either NOT SET (for logic 0) or SET (for logic 1).

We can see from the truth table that once the input A has changed from logic 0 to logic 1 the output becomes SET and cannot be NOT SET by changing A. This is due to the effect of feedback. This sort of circuit is therefore useful for recording that a single event has occurred; it has "memory". However, it would be much more useful if it could be RESET in some way.

This can easily be achieved by introducing a mechanism to drive the feedback signal to logic 0 (thus removing its influence). Consider the logical expression for this, assuming that we have added another input called RESET:

$$\begin{aligned} B_{n+1} &= \big(B_n \ \text{ AND } \ \text{NOT}(\mathrm{RESET})\big) \ \text{ OR } \ \mathrm{SET} \\ &= B_n.\overline{\mathrm{RESET}} + \mathrm{SET} \end{aligned}$$

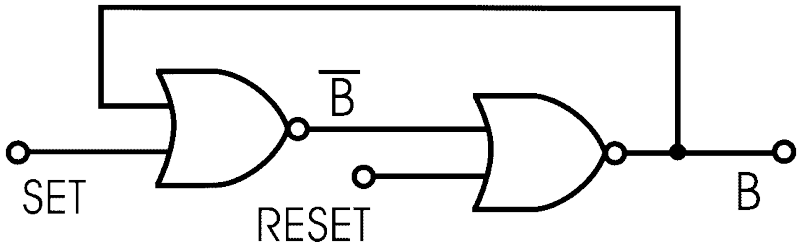
or, equivalently

$$B_{n+1} = (\overline{\overline{B}_n + \mathrm{RESET}}) + \mathrm{SET}$$

which implies

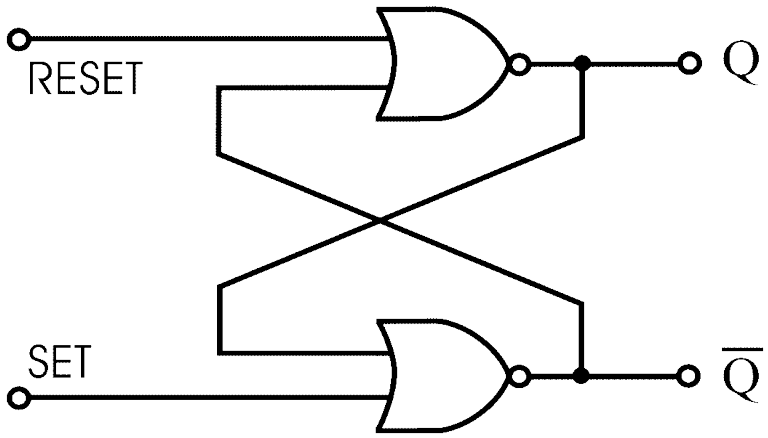
$$\overline{B}_{n+1} = \overline{(\overline{\overline{B}_n + \mathrm{RESET}}) + \mathrm{SET}} \tag{2.1}$$

where De Morgan's law has been used to convert ANDs into NORs. We now have a useful latch made from 2 NOR gates, shown below, that can be set and reset at will.



Modified latch

This circuit may be further re-drawn to emphasise the nature of the feedback circuitry. Output B is renamed Q (the standard letter used to label a state output). The "inverted" B signal is also generated as an output NOT(Q). This implementation of a latch is known as the SR latch; it is shown here with NOR gates, but it can just as well be implemented with NAND gates (although, of course, the latter implementation will have a different truth table).



The SR latch (NOR gates)

The SR latch (shown above for a latch implemented with NOR gates) has a truth table which is different from what we have seen up to now with combinatorial circuits. The feedback introduces time ordering, and so we need to consider the output Q in an ordered fashion. Q_n is considered to be the current value of Q, and Q_{n+1} represents the next state. We see that it is possible to SET and RESET the output by choosing correct values for both S and R. The X in the table, for $S = R = 1$ indicates a problem with this simple circuit. It is not possible simultaneously to SET and RESET the output, and if this were tried the output would be unpredictable (in fact it could end up oscillating between the two possibilities). This input option is considered to be *not allowed*.

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	X

The SR latch is a basic memory element and there is a whole family of latches based on this simple circuit.

2.3 Transparency

As the input values to the latch get modified, the output will change as quickly as the gates can respond. For such simple circuits this is typically a propaga-

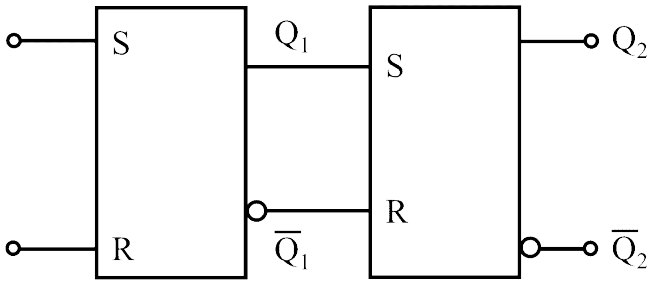
tion delay of the order of a few nanoseconds. This type of response to the input data, being independent of a system clock for example, is referred to as *transparency* and therefore this device is sometimes called a transparent latch.

On its own this delay isn't a huge problem but with a lot of latches and gates combined this slightly asynchronous behaviour can lead to problems. For this reason, many complex logic circuits are synchronised using clock signals (see the next section).

2.4 The clocked master-slave flip-flop

In the Introduction, we mentioned the problems that arise if the propagation delays in circuits are not accounted for. What is needed is synchronisation of the circuit(s) into discrete time slots, using a system clock. We will show how the SR latch of the previous section can be used to build clocked circuitry.

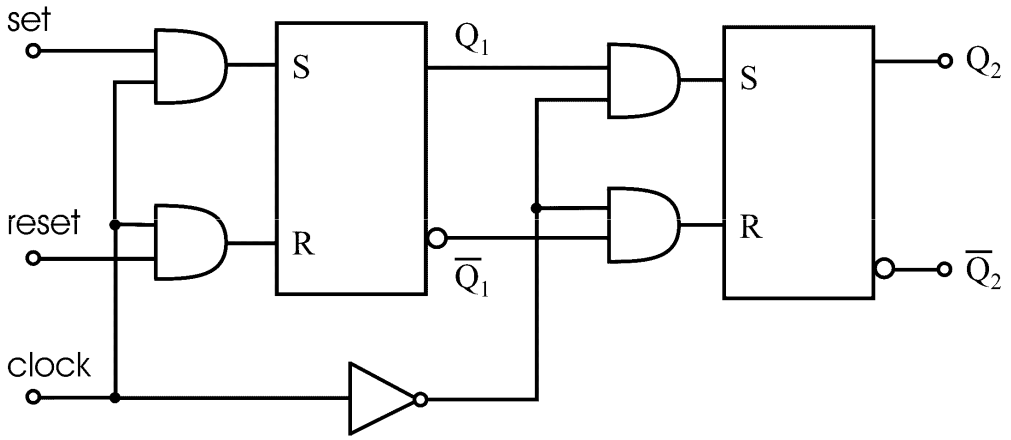
Suppose one wishes to transfer data between two latches in a controlled manner. As an example, suppose we wish to transfer data into SR latch number 1 and then into SR latch number 2, as illustrated below.



Two transparent SR latches connected together

The square boxes with S and R inside them are standard symbols for an SR Transparent Latch (yes – it's not very imaginative), the inverting output being denoted by a small circle. The input to latch 1 will appear at the output latch 2 after an uncertain set of gate delays inside the circuit. What is needed is some means to control the interaction between the two latches, and make their responses time ordered. Using a clock signal that repeats in an ordered fashion it is possible to synchronise the latches. The circuit below, a Master-

Slave SR flip-flop, achieves this objective by using AND gates to synchronise the changing inputs to the latches. Note that the term "flip-flop" (which was presumably chosen as the device can "flip" into one state and then "flop" back to another) is used to refer to circuits that change output state at a clock edge, whereas term "latch" is used to describe a transparent device.



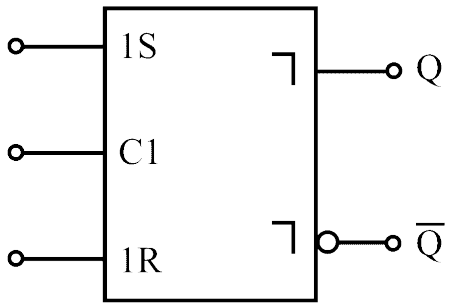
A Master-Slave SR flip-flop

In the above circuit, the Q_2 outputs only change when the clock makes a transition from *logic* 1 to *logic* 0. (Of course, the actual change occurs after the normal propagation delays). This means that a mechanism for synchronising the circuit with a clocking signal has been achieved.

We can see how this works by considering the operation of each transparent latch as the clock input takes different values. First let us assume that the clock is at a logic 1 level. For this case the latch on the left is exposed to the set and reset inputs (as the AND gates pass along the digital values). So, the first latch responds according to the earlier truth table.

However, because of the inverter in the clock path, the right latch is isolated from the changes to Q_1 and as $S = R = 0$ there is no change in Q_2. Only when the clock goes to logic 0 does the Q data pass along and Q_2 is affected in the normal manner. So, the overall output only responds to the input data once the clock signal has gone low. The first latch is considered to dictate the operation of the second latch and hence the term Master-Slave is commonly applied.

As Master-Slave devices are a standard building block, there is a standard diagrammatic representation as shown below.



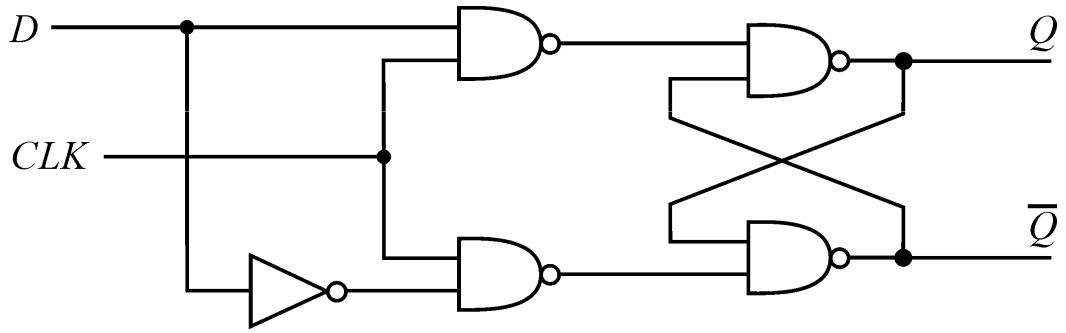
Master-Slave SR flip-flop symbol

The 1S and 1R denote that the device depends on a control input number 1. C1 denotes the connection for the control input (clock) number 1 and the ⌝ symbols on the outputs indicate that this is a Master-Slave device and that the outputs change on the falling clock edge.

You can build a lot of interesting logic using SR latches but more often than not one finds a different type being used for large scale computer memories – the D-type latch.

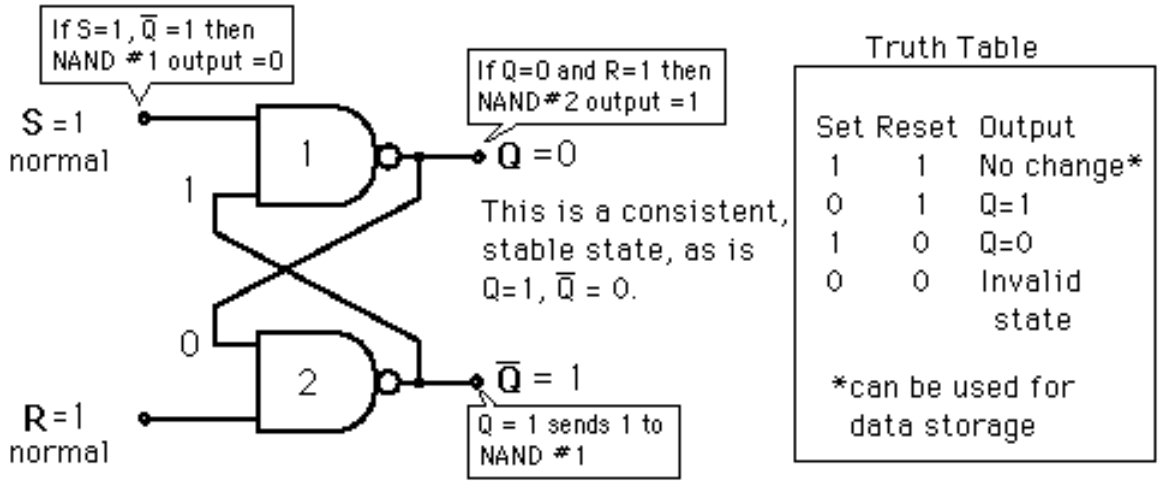
2.5 The D-type latch

An important type of latch is called the D-type latch, where the "D" stands for "data" (but a few people say "Delay"). It is central to most digital designs that rely on storage of data or states. The D latch can be constructed using a simple SR latch as in the following diagram. Here, the SR latch is given in its NAND equivalent form. This is an SR latch on the right hand side with some extra logic driving it. There are still two inputs, the D "Data" input and the Clock (CLK).



NAND SR latch implementation of a D-type Latch: the two NAND gates on the left generate the S and R inputs of the SR stage

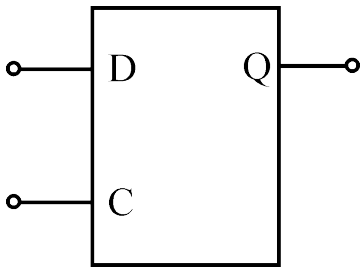
The NAND SR has a slightly different truth table from the NOR version with 0,0 being the invalid input state. The logic on the left hand side here is used to ensure that, should D be high when the clock pulse occurs, then the latch gets set high. Alternatively, if D is low it gets set to low (reset).

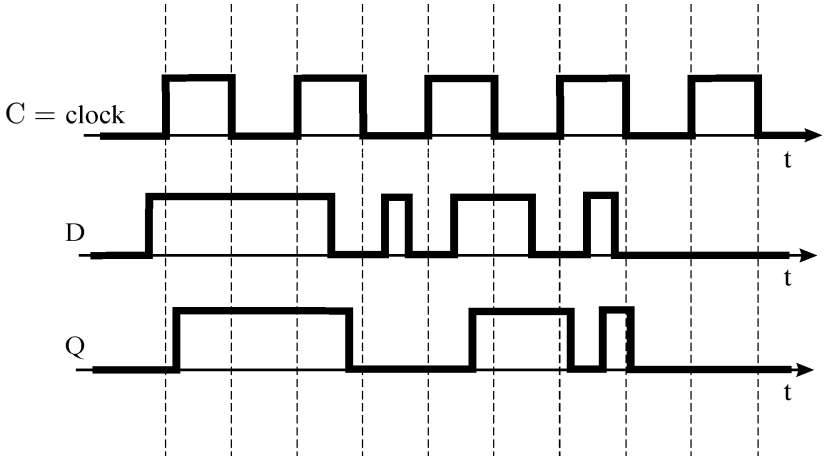


NAND SR latch with truth table
(http://hyperphysics.phy-astr.gsu.edu/hbase/electronic/nandlatch.html)

The truth table for the D latch becomes simple, with the Q output taking on the value of the D input whilst the clock is at logic 1. This is an important feature of a latch. Whilst the clock is high the output is free to change. When the clock goes low, the output remains constant and "holds" the last D value observable at its input. The truth table and symbol for the D latch is shown below and its behaviour is illustrated in the following timing diagram.

D	C	Q_{n+1}
X	0	Q_n
0	1	0
1	1	1

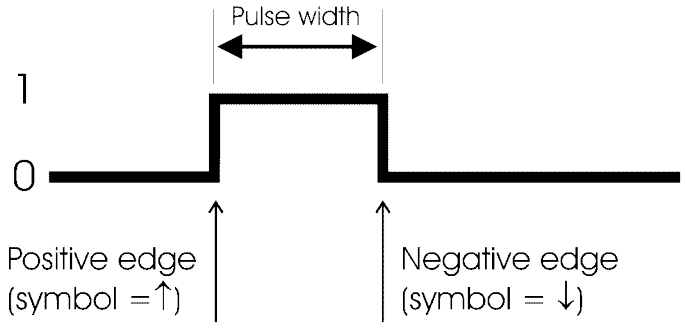




D latch truth table and timing diagram

2.6 The clock pulse

Circuits using just combinational logic and transparent latches are called *fundamental mode* circuits. The uncertainty in timing delays throughout various parts of such circuits can be a serious problem. Instead, whenever possible, a preferred option is to have the elements of the circuit responding to the edge of a system clock. In this lecture, we focus on edge-triggered components which are referred to as either *pulse mode* or *clock mode* circuits. Let us start by examining a clock pulse and defining various attributes.



Timing diagram for a single clock pulse

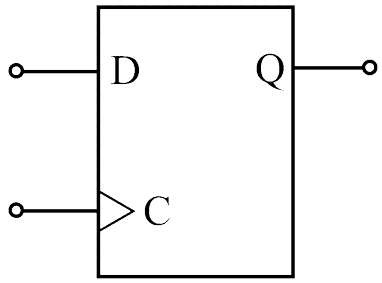
The clock pulse has two transitions, from logic 0 to 1 and from 1 to 0. These

are called the positive and negative edges of the pulse respectively (or sometimes the leading and trailing edges). A positive edge-triggered flip-flop responds to its current data inputs exactly at the 0 to 1 transition of the clock pulse, modifying its output a short propagation delay afterwards. Similarly, a negative edge-triggered flip-flop responds to its input data values on the 1 to 0 transition, producing a modified output shortly afterwards.

2.7 The edge-triggered D-type flip-flop

Remember: a Latch is asynchronous – outputs can change whenever inputs do; a Flip-Flop is synchronous – outputs can only change on a clock edge.

Data storage in pulse mode circuits is almost always achieved using a D-type flip-flop, for storing one bit of data. The circuit symbol for the positive edge-triggered variety is shown below. Like the D latch, it has a single D "data" input and a Q output, but often it is shown with a $\overline{Q}$ output as well. The only control signal it receives is the clock input.



Schematic circuit for an edge-triggered D-type flip-flop

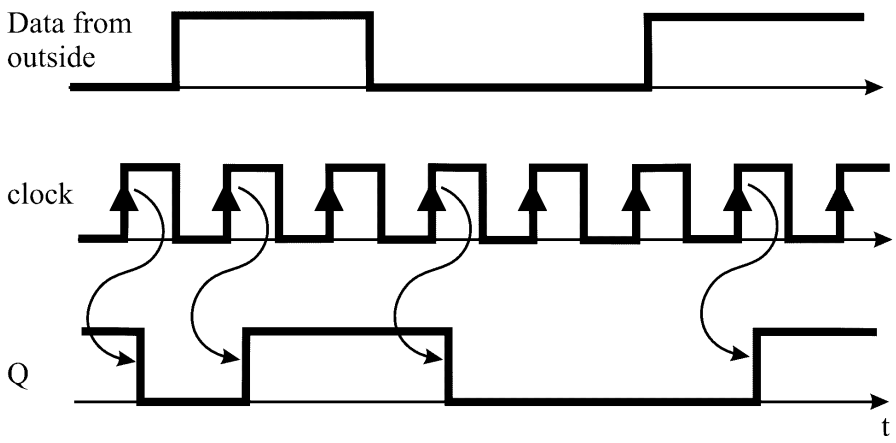
The operation of the edge-sensitive D-type flip-flop is best illustrated by its truth table. Remember that Q_{n+1} refers to the next changed state of the Q output.

D	C	Q_{n+1}
X	0	Q_n
X	1	Q_n
0	$\uparrow$	0
1	$\uparrow$	1

Notice that Q_{n+1} remains the same as Q_n in all cases except when the clock

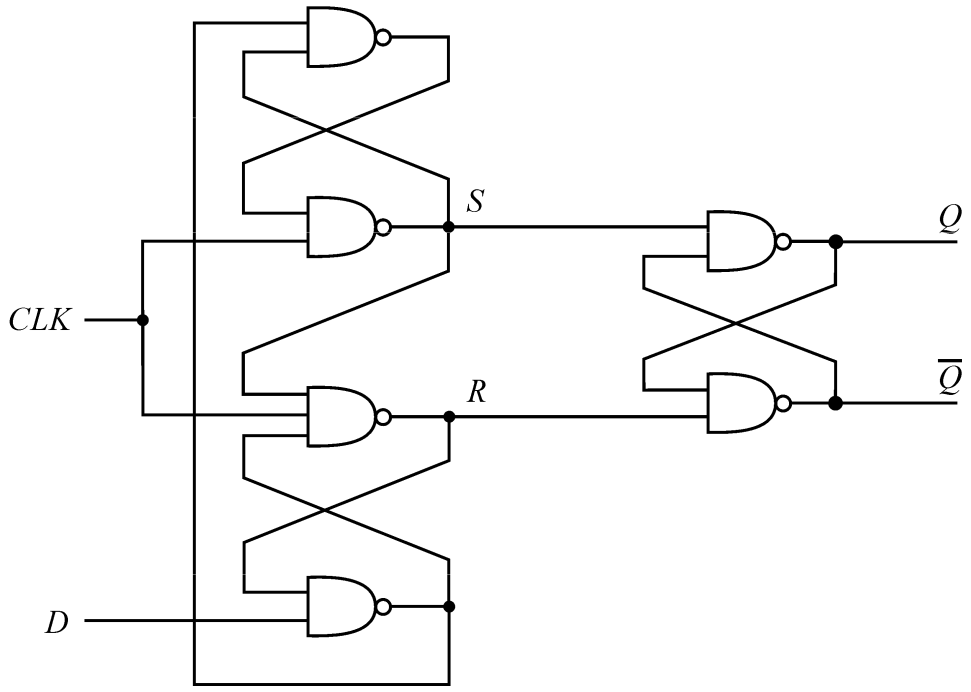
pulse has a positive transition (indicated by the up arrow). When this happens, the value of D is copied to Q_{n+1} (after a small propagation delay). The value of D copied to Q_{n+1} is the value exactly at the transition edge. Shortly after this transition the value of D can change but it will not alter the Q output. In this way the value of D has been stored until the next clock pulse comes along.

The effects of the clock edge transition can be further illustrated using a timing diagram as shown below. Note that the behaviour is different from the D latch, which is transparent when the clock is high. The curved arrows illustrate the dependencies between the clock edge and the changed output; note the slight delay between these effects.



Timing diagram for an edge-triggered D-type flip-flop

A D-type flip-flop circuit, based on the NAND implementation of the SR latch together with the master-slave arrangement, is shown below.

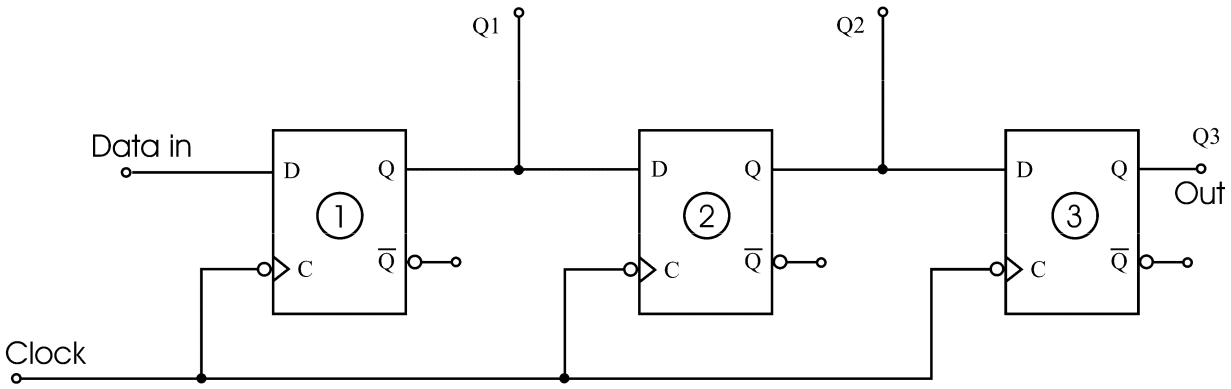


D-type flip-flop circuit (NAND implementation)

There is occasionally some confusion (particularly on various webpages) about the definitions of a latch and a flip-flop. A latch is defined by the fact that the output changes when an input changes (after a small delay of course), whereas in a flip-flop, the output only changes when the clock changes. SR flip-flops can be made either as level-triggered or edge-triggered devices. D-type flip-flops can only ever be edge-triggered.

2.8 Shift register

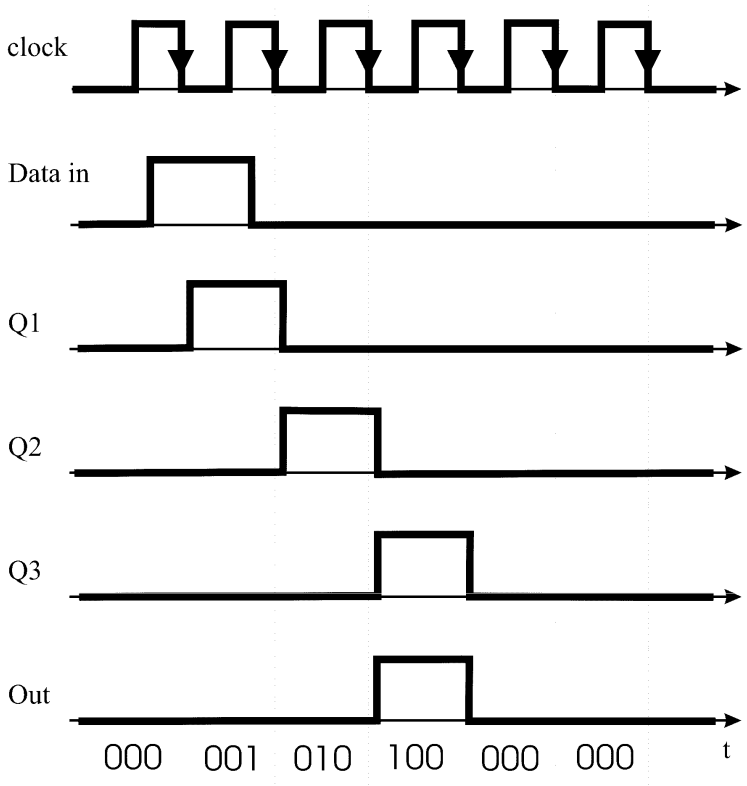
A shift register is a good illustration of how D-type flip-flops work in clock mode. In this example the flip-flops are connected in series and the serial input data is passed from one flip-flop to the other down the chain. The clock mode assumption is very important here. On every negative clock edge (note the small circle at each flip-flop's C input indicating that the clock input is inverted), data present on "Data in" is clocked into flip-flop number 1. At the *same* time, the content of flip-flop 1 is clocked into flip-flop 2. The data present at the input of each flip-flop is thus clocked at the same time and the "old" data shifts one place to the right at every negative clock edge.



A 3-bit shift register with D-type flip-flops

The data moves in discrete time steps, and the behaviour of the circuit is therefore very different from that of the ripple adder circuit discussed earlier, which had an unpredictable propagation delay before the outputs became valid. In this case the data will always be valid after a period equal to the propagation delay of one flip-flop, regardless of how many are cascaded together.

The timing diagram for the 3-stage shift register is shown below. Note how the outputs change (after the propagation delay) on the negative edge of the clock signal.

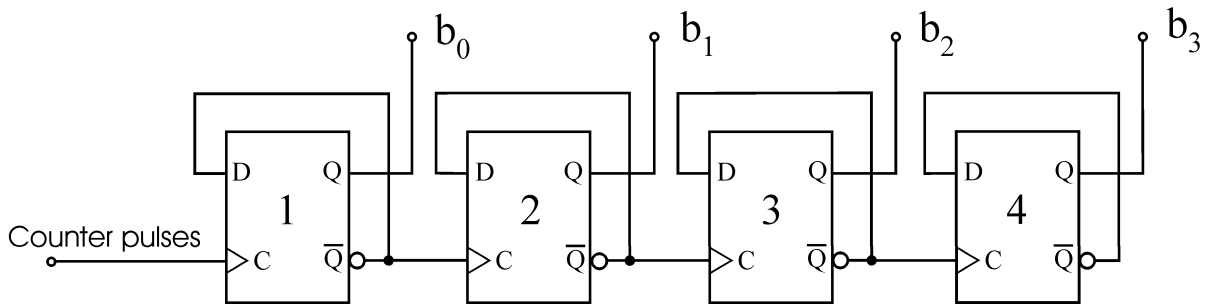


Timing diagram for the 3-bit shift register

2.9 Asynchronous (ripple) counter

D-type flip-flops can also be used to implement a ripple counter, which is an asynchronous circuit. The 4-bit ripple counter, shown below, consists of four D-type flip-flops, each flip-flop holding the value of one of the four bits.

The counter output $b_3\, b_2\, b_1\, b_0$ increases from 0000_2 to 1111_2 (15_{10}) in increments of one at each input pulse. The effect of the first flip-flop ripples along the cascade, so after an input pulse, the count does not settle until each flip-flop has been affected in turn. The propagation delay of each flip-flop therefore determines the overall counter speed. This is one of the reasons why synchronous counter designs are better, since their settling times are usually limited to the propagation delay of just one flip-flop.

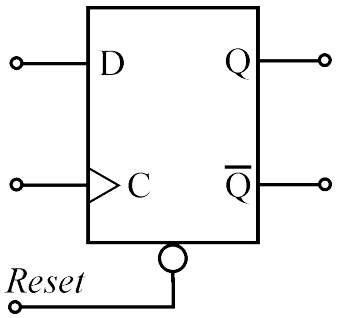


A 4-bit ripple counter implemented with D-type flip-flops

2.10 D-type flip-flop with reset

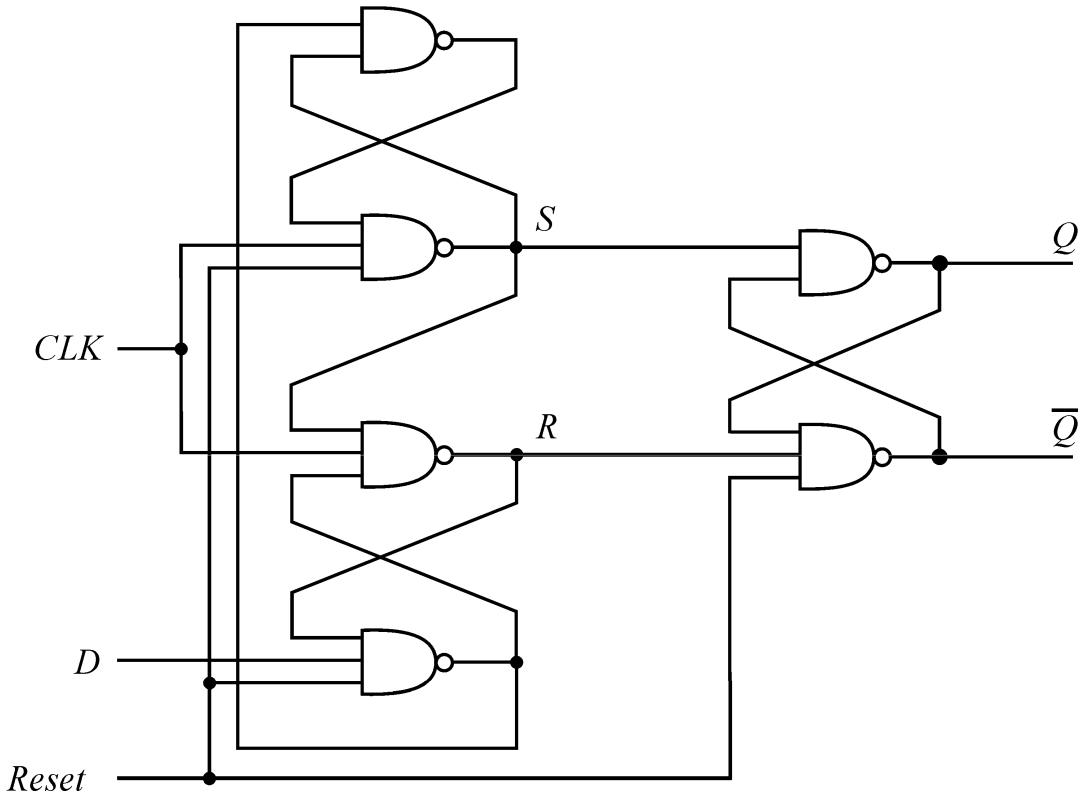
Clearly the counter needs to be reset to zero in some cases, and therefore it is necessary to have access to D-type flip-flops with a reset or clear line. Such flip-flops are useful in asynchronous counters that have a truncated count sequence, e.g. a modulo-6 counter, which counts from 0 to 5, then returns to 0. The symbol below is for a flip-flop with an active low reset line.

D	C	$Reset$	Q_{n+1}
X	0	1	Q_n
X	1	1	Q_n
0	$\uparrow$	1	0
1	$\uparrow$	1	1
X	X	0	0



D-type flip-flop with reset

A D-type flip-flop with reset constructed using NAND gates is shown below.



NAND gate D-type flip-flop with reset

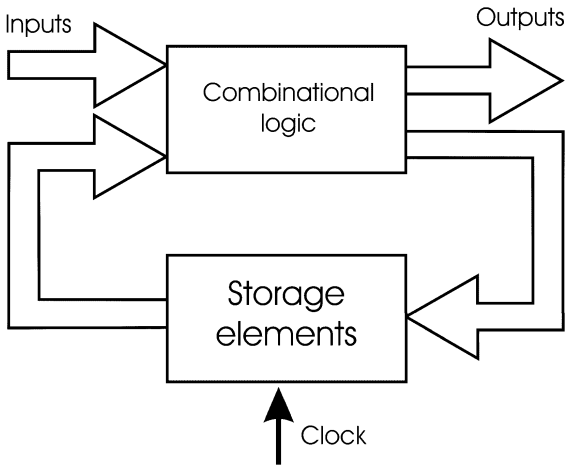
3 Lecture G – State machines and sequential logic

3.1 The digital state revisited

A disadvantage of asynchronous counters is that their feedback interconnections cause asynchronous interactions between the discrete states of the flip-flops, resulting in variable settling times as seen in the asynchronous ripple counter of Section 2.9. In the rest of this course we will concentrate on clock-mode designs, in which the state is fully defined by the outputs of the flip-flops within the circuit, and there is no feedback to the transparent inputs.

The circuit moves from its current state to its next state (which depends on the current state and external inputs) at each clock transition. Some combinational logic may be required to achieve this by decoding the current state and any inputs to the circuit, but essentially the progression through the states can only happen synchronised to the clock signal.

The combinational logic elements, as well as the system clock and the flip-flops which store the current state, are arranged in the configuration shown in the block diagram below. The current state of the system is defined by the outputs of the storage elements. The next state is defined by the inputs to the storage elements, but this next state will not be entered until the controlling clock pulse occurs. The outputs are defined as a logical function of the current inputs and the current state.

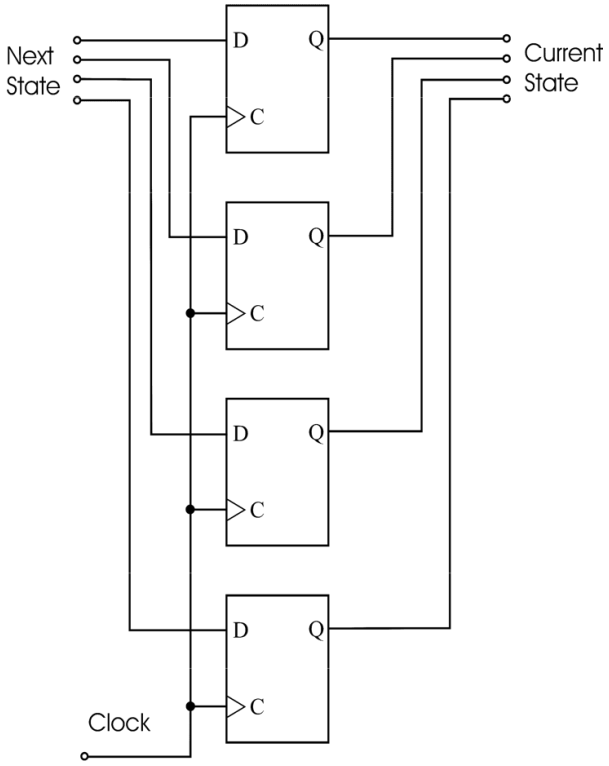


Block diagram of a synchronous finite-state machine

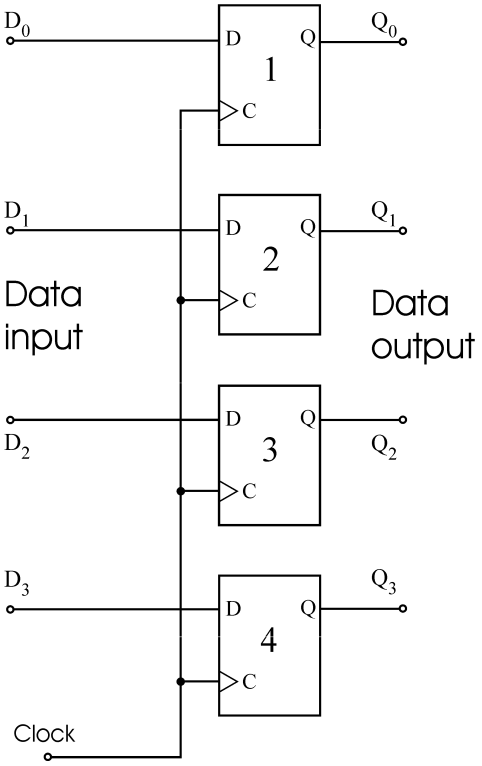
3.2 Storing the state

The data storage elements can be implemented most simply by using an array of D-type flip-flops, all driven by the system clock. Each flip-flop provides a single bit of storage. On each clock edge, the values provided by the combinational logic are recorded.

As an example, the following circuit shows a 4-bit register (on the left), and how it is used to store the system state $Q_3\,Q_2\,Q_1\,Q_0$ (on the right). This system can have 16 unique states, since $2^4 = 16$. The data is clocked through on the positive edge of the clock, and the flip-flop inputs $D_3\,D_2\,D_1\,D_0$ define the next state of the machine.



A 4-bit register



Using the register to store the state

3.3 State transition tables

We will next look at tools that will allow us to design the combinational logic that specifies state transitions. One way of describing how a state machine operates is to draw a table similar to a Karnaugh Map. The rows correspond to the current state and the columns to the values of the inputs. The entries

in the table correspond to the next state.

The table shown below is an example of such a transition table. Note that the entries for this table have no particular meaning and are just chosen for illustration.

		x_1x_2			
	Y_1Y_2	00	01	11	10
y_1y_2	00	00	00	10	01
	01	01	00	11	01
	11	01	10	11	11
	10	00	10	10	11

The leftmost column is labelled y_1y_2 and corresponds to all possible states for this system. We have four states defined by two state variables y_1y_2, giving rise to four rows. There are two inputs, x_1 and x_2, that give rise to the four columns. (Sometimes the internal variables y_1y_2 are called the secondary variables, whereas the inputs x_1x_2 are called the primary variables.) The whole table is labelled Y_1Y_2, which means that the entries of the table correspond to the next state (the current state being denoted by the lower case y's and the next state by the upper case Y's).

Consider the first row of the transition table. If the system is in state 00, then it remains in this state after the next clock edge if the input is 00 or 01. If the input is 11 or 10 there will be a state transition at the clock edge. Changing the input values does not change the state immediately, thus a change in input just corresponds to moving to another column in the same row. A row change only occurs at a clock edge, the next row being defined by the entry in the current table position. In order to design an arbitrary sequence generator we need to identify these state transitions and then work out the logic needed to achieve them. That is the subject of the next lecture.

3.4 Synchronous counters

With synchronous counters, all state transitions occur on a clock edge. Sophisticated designs can be obtained using a Programmable Read Only Memory

(PROM) instead of discrete gates in a combinational logic circuit, but both approaches will be explored in this lecture.

3.4.1 Generating the transition table

Consider first a simple counter example, say a modulo-5 or divide-by-6 counter (i.e. a counter with a count sequence $0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 0, 1, \ldots$). For such a counter, there are 6 states, which requires 3 bits.

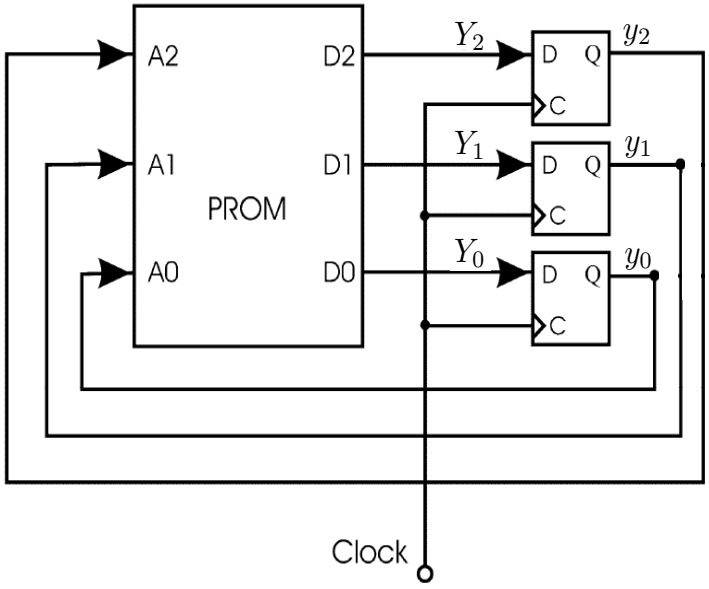
The transition table below shows the current state in the left column taking the lower case labels $y_2y_1y_0$, and the entries referring to the next state occurring on one of the clock edges, taking the upper case labels $Y_2Y_1Y_0$.

Current State $y_2y_1y_0$	Next State $Y_2Y_1Y_0$
000	001
001	010
010	011
011	100
100	101
101	000

3.4.2 Using registers to store the state: ROM-based counters

We now return to the state machine considered in Section 3.1, but we introduce a modification. Instead of using combinational logic, we will use a ROM to decode the next state.

Simple synchronous counters have no inputs, and the outputs are the states stored in the D-type flip-flops. Therefore, to implement the divide-by-6 counter, we can modify the generic state machine block diagram as shown below. Here a PROM replaces the combinational logic and there are no additional external inputs (because none are needed for this simple counter circuit).



PROM implementation of a divide-by-6 synchronous counter

In the ROM-based design the **next state** is the **data** output from the ROM in response to the **current state**, stored by the flip-flops. The current state can also be considered as the **address** input to the ROM. At every clock edge, the data output from the ROM is transferred to the D-type flip-flops. The clock-mode restriction ensures that the only data clocked into the flip-flops is that being generated by the ROM in response to the **current state**. It is the fact that the D-type flip-flops are edge-triggered that ensures that the above circuit works.

3.4.3 Data stored in the PROM

The PROM has a three-bit address ($A_2A_1A_0$) and provides a three-bit output ($D_2D_1D_0$). In this simple example the contents of the PROM, pointed to by the current address, must simply contain the next state. The necessary data content of the PROM to implement the divide-by-6 counter is given in the following table.

$A_2A_1A_0$	$D_2D_1D_0$
000	001
001	010
010	011
011	100
100	101
101	000
110	XXX
111	XXX

Note that there are two "don't care" lines in the truth table, since the circuit never reaches these states (remember that for a ROM all states have to be decoded, unlike a PLA design approach).

What should happen if the machine is powered up into one of these two unimplemented states? A good circuit designer would consider this possibility and probably decide to put the data values 000 into the PROM locations 110 and 111. In this way if any one of these unimplemented states are entered by mistake then the machine is steered to the initial state and counting would proceed as normal.

3.5 Sequencers: number of states required

The sequence for the modulo-6 or divide-by-6 counter is just the count from 0 to 5 repeated over. These 6 states could also refer to a different count sequence, for example, the repeated sequence $0, 1, 3, 5, 2, 4, 0, 1, 3, 5, 2 \ldots$. This is easy to achieve, as all that is required is for the PROM to be re-programmed as shown below.

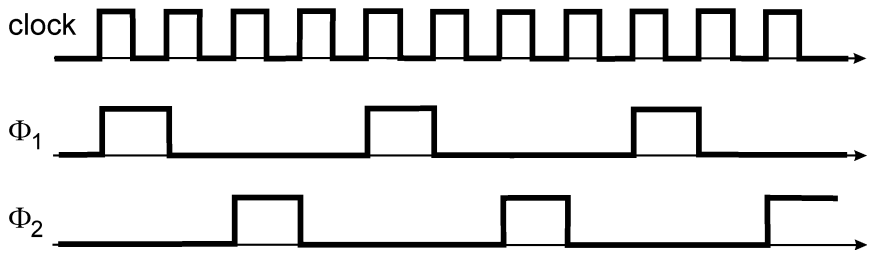
$A_2A_1A_0$	$D_2D_1D_0$
000	001
001	011
010	100
011	101
100	000
101	010
110	XXX
111	XXX

This illustrates the advantages of the PROM approach, compared with decoding using combinational logic (see later). With the latter, a simple change in the state sequence would require a complete re-design of the combinational logic circuitry.

It is not always possible to design a circuit in which the states are used as the outputs directly. This problem arises when there is not a unique "next state" corresponding to each output. As an example, consider the following problem.

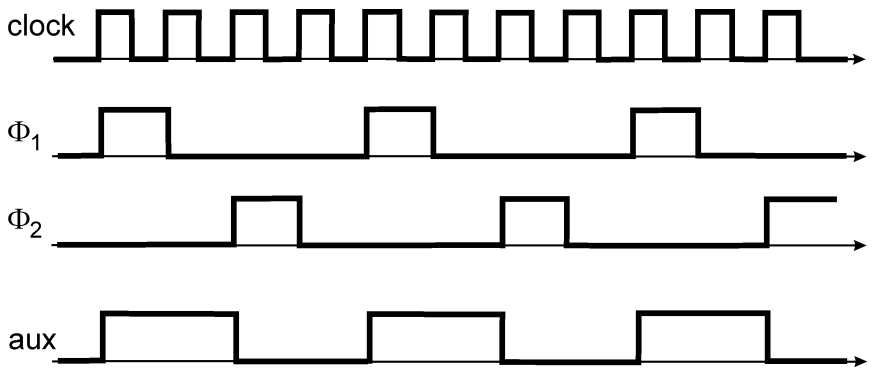
3.5.1 Example: a two-phase clock generator

In CMOS logic design, it is often necessary to generate a clocking signal consisting of two synchronised, non-overlapping clocks. The two-phase clock generator circuit needs to be driven by a single clock input and produce two non-overlapping clock outputs Φ_1 and Φ_2 as shown in the next figure. The preferred design approach is to use a ROM-based sequencer using D-type flip-flops as the storage elements, with the outputs being the states of the flip-flops.



Timing diagram for two-phase clock generator

The two non-overlapping clocks shown above change state on the positive, or leading, edge of the clock, and so positive edge-triggered devices are needed. However, since the outputs $\Phi_1\Phi_2$ follow the sequence 00, 01, 00, 10, 00, 01, ..., there is no unique state that follows the state 00 (it can be 01 or 10). This may be overcome by defining a third variable "aux", which is a logical 1 for one of the 00 states and logical 0 for the other, as shown in the revised timing diagram below:



Revised timing diagram for two-phase clock generator

The sequence is now 000, 101, 100, 010, 000, 101, ..., and there are two unique states associated with the non-overlapping clock outputs 00.

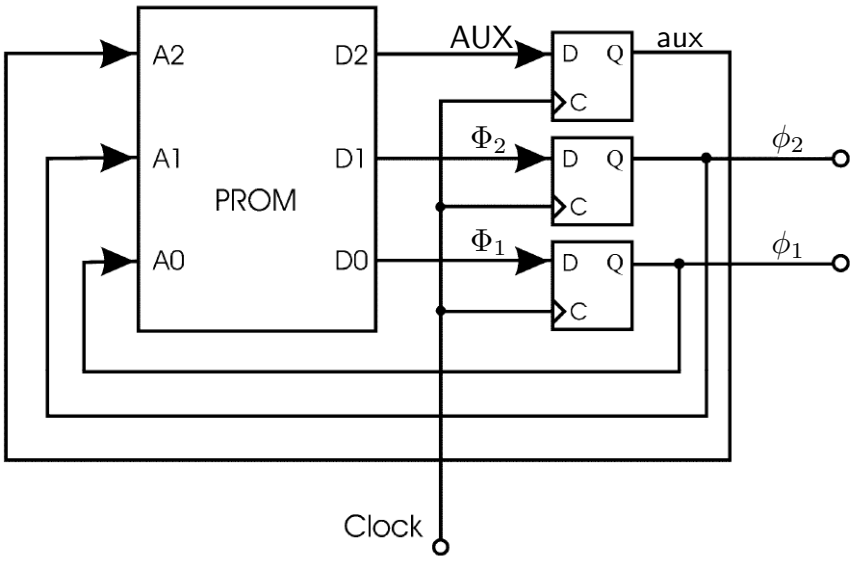
The transition table is:

Current state aux $\phi_2\phi_1$	Next state AUX $\Phi_2\Phi_1$
000	101
101	100
100	010
010	000

The transition table for the ROM is:

$A_2A_1A_0$	$D_2D_1D_0$
000	101
001	XXX
010	000
011	XXX
100	010
101	100
110	XXX
111	XXX

Finally, the circuit implementation for the non-overlapping clock generator is given below, where the PROM contains the data presented in the above table.



Two-phase clock generator

The above example demonstrates that the number of states required to implement a sequence depends on the length of the sequence, and not just the number of independent combinations of outputs required in the sequence.

3.6 Synchronous D-type flip-flop designs

The finite-state machines introduced above have used a combination of both D-type flip-flops and a PROM. Although this is a versatile and powerful design approach, it is not the only option and the need to fully encode all the memory locations of the PROM can be a disadvantage for simple designs. The latter can make use instead of combinational logic to drive the D-type flip-flops into the next state.

3.6.1 The steering table (or transition list)

When designing sequential circuits, it is useful to determine what value of D should be chosen to achieve the desired state transition at the Q output, assuming the current state is known. The steering table identifies what value is needed on the D input in order to obtain the desired transition on the clock edge. It is straightforward to derive the steering table (shown on the right below) from the truth table (left), because, for the D-type flip-flop, we need $D = 0$ whenever the transition is **0** or **L**, whereas we need $D = 1$ whenever the transition is **1** or **H**.

D	Q_n	Q_{n+1}	Transition
0	0	0	$0 \rightarrow 0 = \mathbf{0}$
1	0	1	$0 \rightarrow 1 = \mathbf{H}$
1	1	1	$1 \rightarrow 1 = \mathbf{1}$
0	1	0	$1 \rightarrow 0 = \mathbf{L}$

$\Longrightarrow$

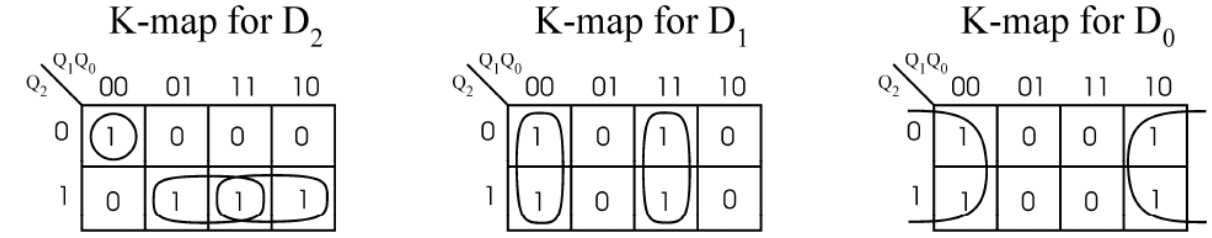
Transition	D
$0 \rightarrow 0 = \mathbf{0}$	0
$0 \rightarrow 1 = \mathbf{H}$	1
$0 \rightarrow 0 = \mathbf{1}$	1
$0 \rightarrow 0 = \mathbf{L}$	0

3.6.2 Example: synchronous counter

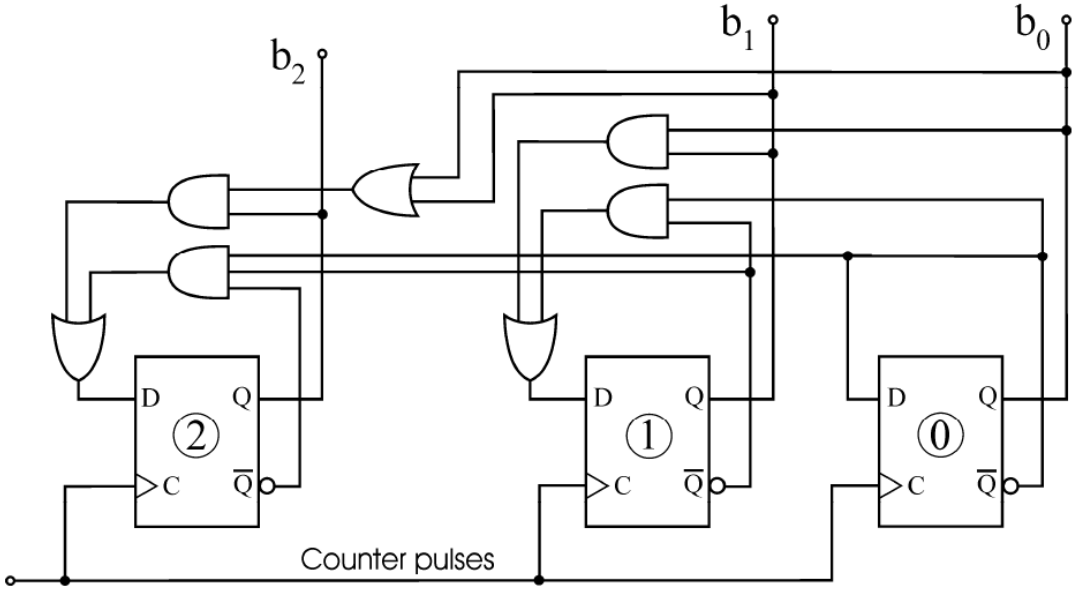
As an exercise in the design of a sequential logic circuit, consider a simple synchronous counter that counts down from 111 to 000 and back to 111, in a continuous sequence. The state transition table for a D-type implementation is given below, with the desired transitions on the right.

Current state				Next state				Transition		
$Q_{2,n}$	$Q_{1,n}$	$Q_{0,n}$		$Q_{2,n+1}$	$Q_{1,n+1}$	$Q_{0,n+1}$		D_2	D_1	D_0
1	1	1	(7)	1	1	0	(6)	**1**	**1**	**L**
1	1	0	(6)	1	0	1	(5)	**1**	**L**	**H**
1	0	1	(5)	1	0	0	(4)	**1**	**0**	**L**
1	0	0	(4)	0	1	1	(3)	**L**	**H**	**L**
0	1	1	(3)	0	1	0	(2)	**0**	**1**	**L**
0	1	0	(2)	0	0	1	(1)	**0**	**L**	**H**
0	0	1	(1)	0	0	0	(0)	**0**	**0**	**L**
0	0	0	(0)	1	1	1	(7)	**H**	**H**	**H**

Using K-maps, we can now determine the logic gates to implement the combinational logic to drive the 3 D-type flip-flops. The K-maps, logic functions and corresponding circuit are given below.



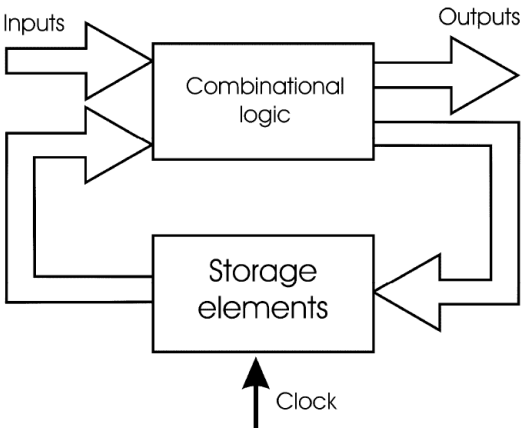
$D_2 = \overline{Q_2}\,\overline{Q_1}\,\overline{Q_0} + Q_2(Q_1+Q_0)$ $\quad D_1 = \overline{Q_1}\,\overline{Q_0} + Q_1Q_0$ $\quad D_0 = \overline{Q_0}$



A D-type implementation of a “down” synchronous counter

3.7 Conclusion

This lecture looked at how the finite-state machine architecture shown in the diagram below can be used to implement synchronous counters. These are simple examples of finite-state machines, which are defined by their states (of which there is a finite number), the transitions between these states, and the actions or outputs associated with these states.



The concept of a finite-state machine is at the heart of the theory of computing, as it allows for a description of how bits of information can be handled by a sequential machine. If we introduce a branching capability which allows the sequence of states to change according to an input condition, we have most of the elements required for understanding how a computer work

4 Lecture H – Data converters: ADC and DAC

4.1 Introduction

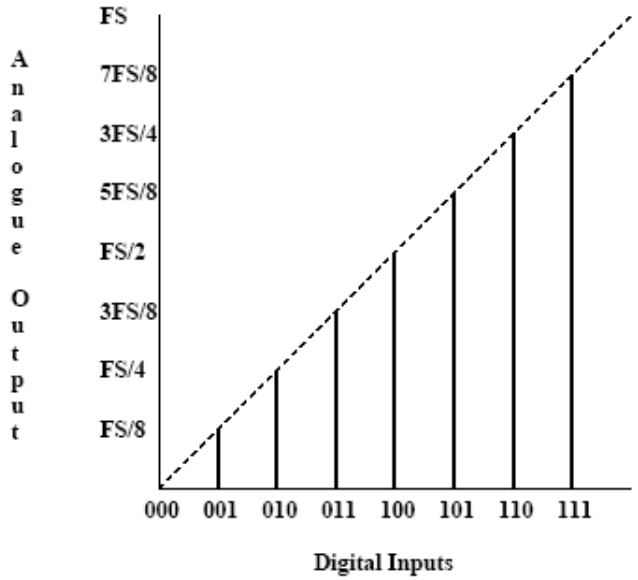
The outputs from sensors and communications receivers are analogue signals that have continuously varying amplitudes. In many systems it is convenient to record and/or process these signals within a digital circuit, which may be a microcontroller, microprocessor or a computer. In a digital circuit the signal will be represented as a list of binary numbers, with each number representing the amplitude of the signal at a specific time.

4.2 Data converters

Conversion from an analogue signal to a digital number is performed by an *analogue-to-digital converter* (ADC). There are several different types of ADCs, some of which contain a digital-to-analogue converter (DAC) that converts a digital number to the equivalent analogue signal. When taken together with their independent role in creating analogue output signals to drive parts such as heaters and motors, this makes DACs a critical part of many systems. It is therefore important to understand the operation of both DACs and ADCs.

4.3 Specification of D/A converters (DACs)

A digital to analogue converter (DAC) converters a digital input represented as a binary number to an analogue voltage (or current) that is proportional to the value of this input. The *ideal* relationship between the analogue output and digital input for a 3-bit converter is shown below.



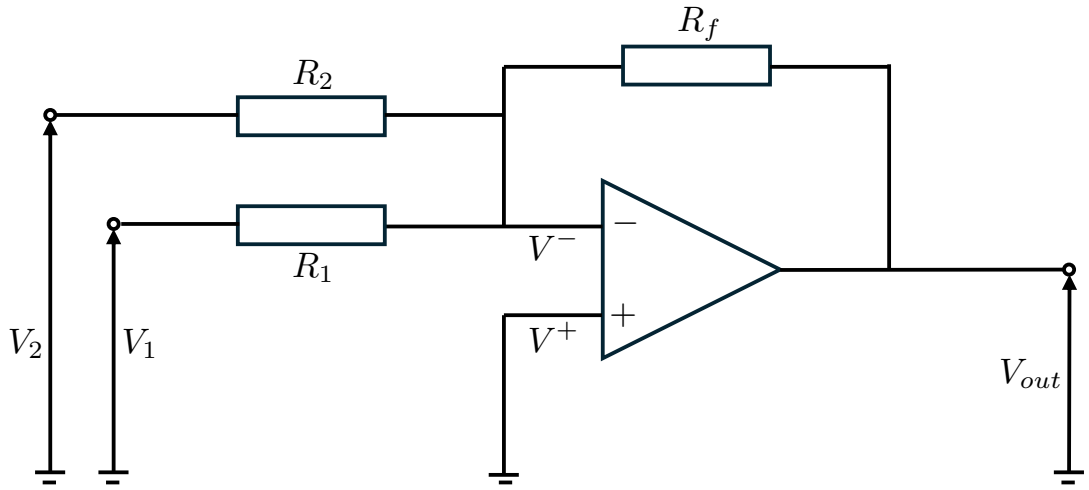
The ideal response of a 3-bit DAC

The graph shows the analogue output voltage as a fraction of the full scale output, denoted by FS. Each bar represents the output for a particular input and the dashed line shows the line connecting the ideal outputs. In this diagram 3 bits have been shown for clarity. However, in real instrumentation systems, DACs with 8, 10, 12 or 14 bits are often used.

4.4 D/A converter architectures (DAC architectures)

4.4.1 The summing amplifier

The basis operation required to create a DAC is the ability to add inputs that will eventually correspond to the contributions of the various bits of the digital input. In the voltage domain, that is if the input signals are voltages, addition can be achieved using the inverting summing amplifier shown in below.



An inverting summing amplifier

To understand how this circuit operates assume that the op-amp is ideal. Since the op-amp is ideal $V^- = V^+$, but, in this circuit $V^+ = 0$, so the current flowing into this node from the two inputs is

$$I_{in} = \frac{V_1}{R_1} + \frac{V_2}{R_2}$$

Since no current flows into the inverting input of the ideal op-amp, all this current must flow around the feedback loop through resistor R_f. This will only happen when the op-amp output voltage is

$$V_{out} = -I_{in} R_f$$

which becomes

$$V_{out} = -\frac{V_1 R_f}{R_1} - \frac{V_2 R_f}{R_2}$$

Now if we assume that

$$R_f = R_2 = 2R_1$$

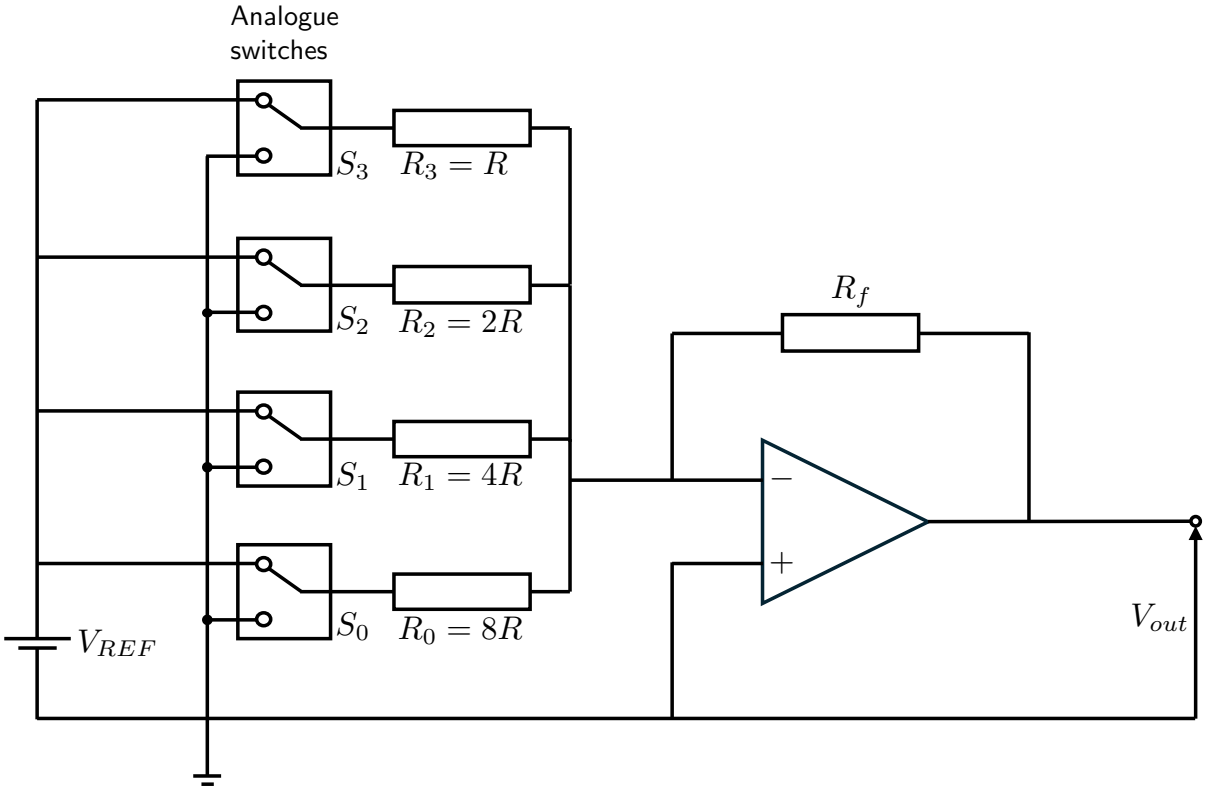
then

$$V_{out} = -(2V_1 + V_2).$$

This weighted combination of inputs is the principle behind the operation of many digital-to-analogue converters.

4.4.2 D/A Converters

The simplest way of convert a digital input word into a corresponding analogue voltage is to use an op-amp as a summing amplifier with a weighted resistor "ladder", as shown below.



A 4-bit DAC with weighting resistors and summing amplifier

At the start of the conversion process, a 4-bit input code, $B_3B_2B_1B_0$, is applied to control the corresponding switches $S_3S_2S_1S_0$. Each switch S_n connects the resistor R_n to the voltage source V_{REF} when the corresponding bit B_n is high, whereas if B_n is low the resistor R_n is grounded. The other end of each resistor is connected to the summing junction of the op-amp. For a 4-bit converter, in which the resistors are in the ratio $8:4:2:1$, as shown in the figure above,

the total current flowing onto the inverting input of the op-amp is

$$I_{in} = V_{REF}\left(\frac{B_3}{R} + \frac{B_2}{2R} + \frac{B_1}{4R} + \frac{B_0}{8R}\right)$$

this current then flows through R_f to generate the output voltage and hence

$$V_{out} = -\frac{R_f}{R}V_{REF}\left(B_3 + \frac{B_2}{2} + \frac{B_1}{4} + \frac{B_0}{8}\right)$$

The output voltage V_{out} is therefore proportional to a weighted sum of the input bits. If $R = 2R_f$, then the relationship between digital inputs and the analogue output voltage V_{out} will be as shown in the following table.

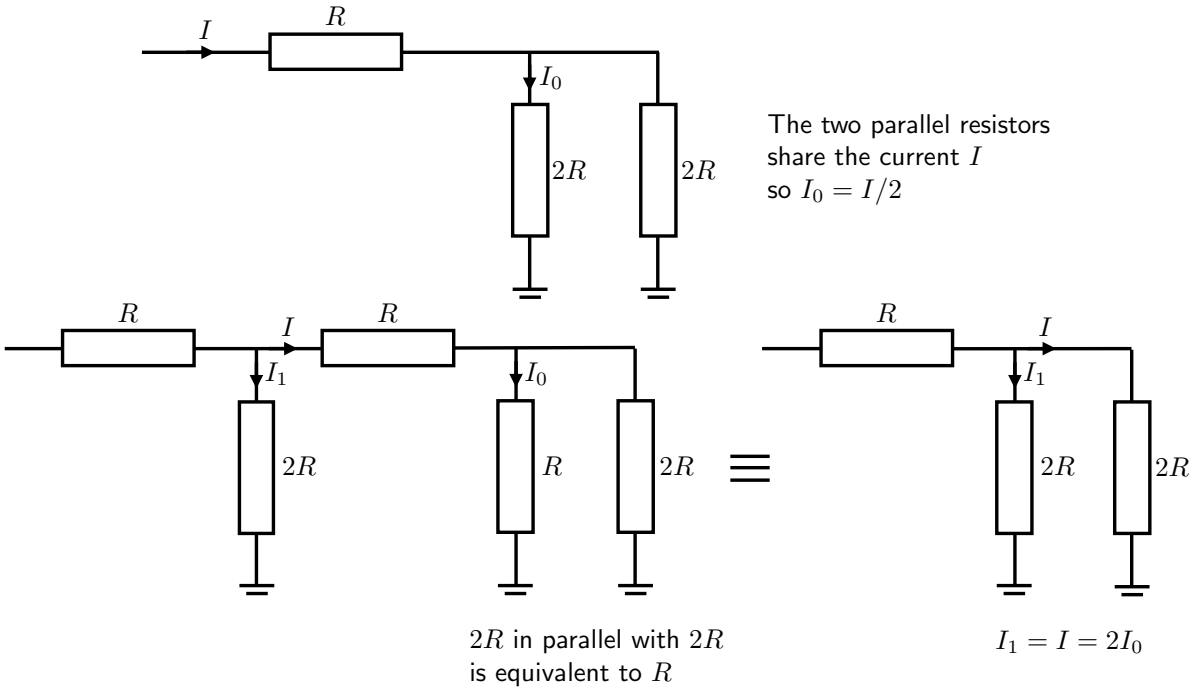
Digital input	V_{out}
0000	0
0001	$-\frac{1}{16}V_{REF}$
0010	$-\frac{2}{16}V_{REF}$
$\vdots$	$\vdots$
1110	$-\frac{14}{16}V_{REF}$
1111	$-\frac{15}{16}V_{REF}$

The circuit therefore achieves the desired function of producing a voltage proportional to the value of the binary number represented by $B_3B_2B_1B_0$. However, there are two main problems with this circuit:

(i). The output voltage from the reference voltage source must stay constant even when its output current is changing, i.e. its source resistance must be zero.

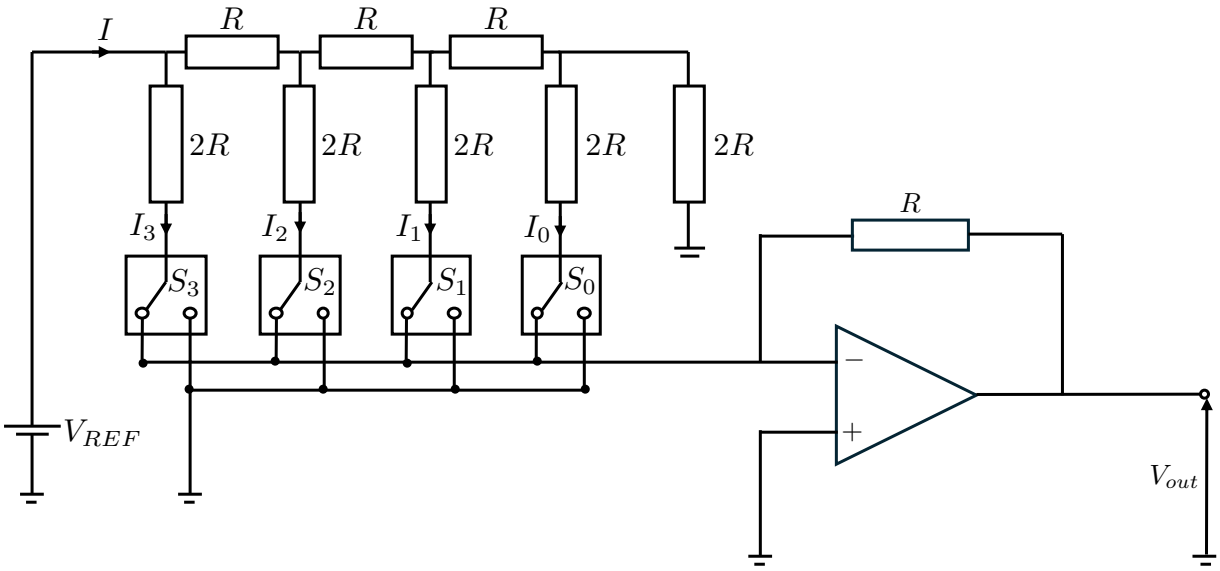
(ii). The resistor values must be very accurate and in the correct ratio to one another. Although this requirement can be achieved in an integrated circuit, the range of values required for, say, a 12-bit D/A converter (for example, $10\,\text{k}\Omega$ to $20.48\,\text{M}\Omega$) makes it impractical.

For these reasons, a different type of resistor network is normally used, the R-$2R$ *ladder*, which can be constructed, as its name indicates, using just two values of resistors. This network, shown in the figures below, therefore avoids the need to create different resistance values.



Analysis of the R-$2R$ ladder network

The trick to analysing the R-$2R$ ladder network is to start from the right-hand end. As shown in the diagram below, there are two $2R$ resistors in parallel at this end of the ladder, which are therefore equivalent to a single series resistor R. This resistance is in series with resistance R, forming a total resistance of $2R$. But this $2R$ is in parallel with another resistance $2R$, and so on. Thus, all elements to the right of a particular node in the ladder network are equivalent to a single resistance of $2R$.



An R-$2R$ ladder 4-bit D/A converter

The analysis of the ladder network means that for the network in the figure above the incoming current splits into two at each node and thus

$$I_2 = 2I_1 = 4I_0$$

and

$$I_3 = V_{REF}/2R = 2I_2 = 4I_1 = 8I_0.$$

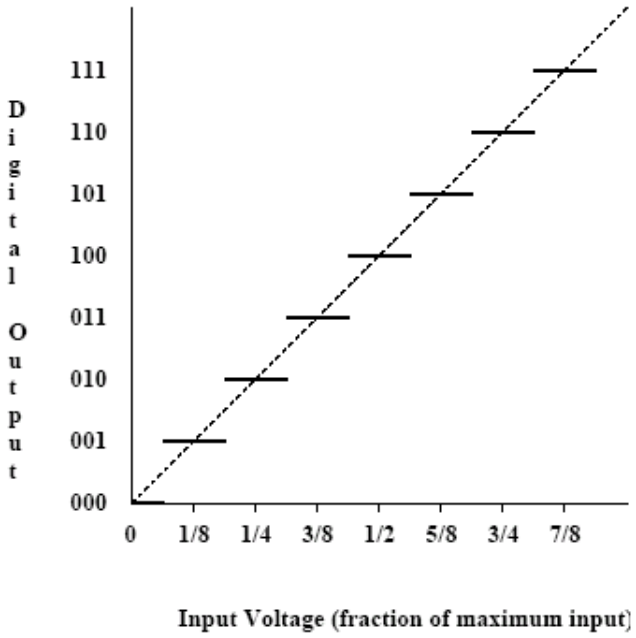
As with the previous circuit, each bit B_n of the digital code controls a switch S_n. When $B_n = 1$, the switch S_n directs current I_n towards the summing junction; otherwise the current flows straight to ground. The DAC output voltage is therefore determined by a current that is proportional to the weighted sum of the input bits as required.

The other advantage of this architecture is that the inverting input of the op-amp is a virtual earth and hence one end of each the $2R$ resistors is always connected to ground. This means that the current flowing through each branch of the ladder network is independent of the switch conditions and hence the digital input. The significance of this is that it means that the total current supplied by the voltage source is constant and the circuit performance is independent of the output impedance of the voltage source.

One potentially useful modification to this basic architecture is to use a variable voltage source, possibly formed by a second DAC, to create a variable reference voltage. The analogue output signal is then proportional to the product of the variable reference voltage and the input binary number; this type of device is usually known as a *multiplying DAC* or MDAC.

4.5 A/D converters (ADCs)

Analogue to Digital (A/D) conversion is the process whereby an analogue signal is converted into a corresponding binary number, the digital output. The ideal relationship between the analogue input and the digital output for a 3-bit A/D converter is shown below. The input analogue values are *quantised* by dividing the continuous analogue input range into 8 discrete steps or code ranges.



The ideal response of a 3-bit ADC

Since the ADC is unable to distinguish among different values in the same code range the output can have an error as large as $\pm\frac{1}{2}$ LSB. This *quantisation error* is an intrinsic limitation of representing a continuous input by a finite set of output numbers. The first approach to minimising the effects of quantisation errors is to ensure that the maximum expected amplitude of the input signal matches the input range of the ADC. This usually means that amplifiers are needed between the signal source and the ADC. By using an active low-pass filter this amplification function can be performed by the anti-aliasing filter.

The other method of reducing quantisation error is to increase the number of output bits. For example $2^{12} = 4096$ and hence a 12-bit A/D converter can resolve a signal to 1 part in 4096, or 0.024% of the maximum input.

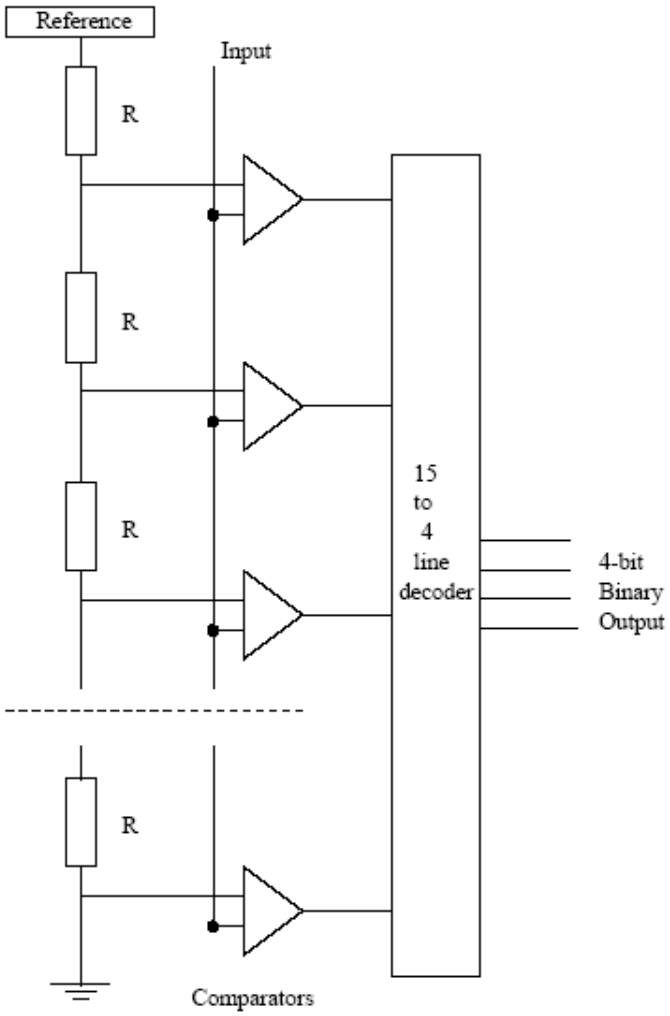
The two types of A/D converter that we will discuss are:

- Parallel converters
- Successive approximation converters

The choice between the types of converter is made on the grounds of the cost, resolution and speed required for a particular application.

4.5.1 Parallel ADCs

Parallel encoding (sometimes known as "flash" encoding) is the fastest (and most expensive) method of A/D conversion. This architecture, which is shown in the figure below, achieves an n-bit conversion by simultaneously comparing the analogue input with $2^n - 1$ voltage reference levels. The reference levels are usually generated by a chain of identical resistors connected in series. Each of these references is compared to the input by a device known as a comparator. This is a circuit with a very high differential gain so that its output saturates to a maximum value when the voltage on the non-inverting input is higher than the voltage on the inverting input. Otherwise the comparator output voltage saturates to a minimum value.



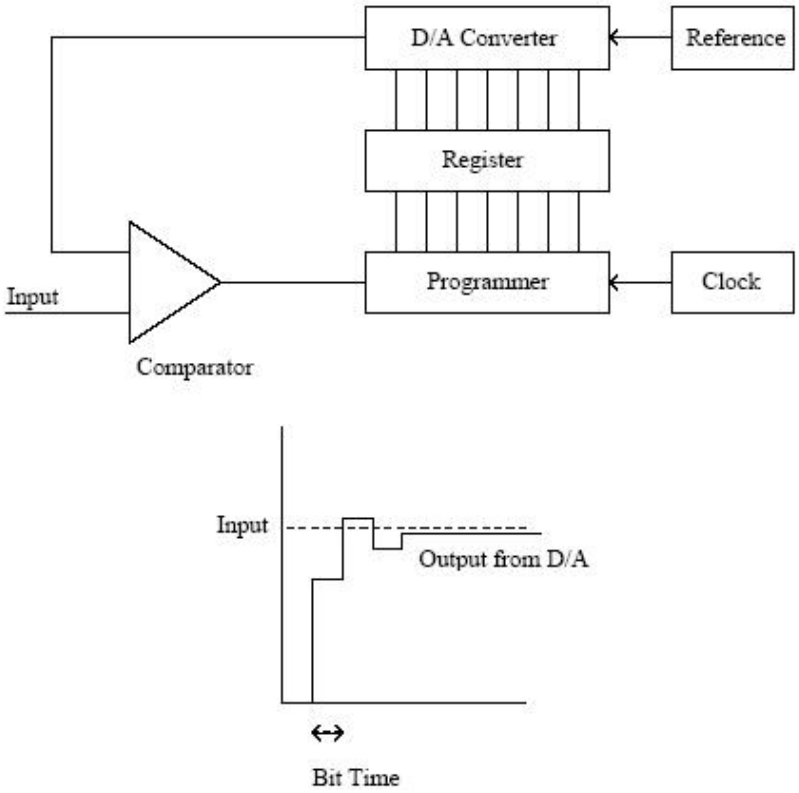
Schematic of a flash A/D converter

A comparator array can therefore be designed so that the outputs from all comparators whose reference voltage is below the common input saturate to a maximum output value, while the output of all the other comparators in the array saturate at the minimum output voltage. The maximum comparator output voltage is interpreted as a logical 1, whilst the low output is interpreted as logical 0. Therefore the $2^n - 1$ outputs from the array represent the analogue input value by the position of the transition between ones (from the comparators with reference voltages below the input voltage) and zeros (from those with references above the input). The position of the transition between the ones and the zeros moves as the analogue input voltage changes. This representation is therefore referred to as a *thermometer code*. The final stage of the conversion process is to use a digital circuit, known as an encoder, to convert this representation of the input to a conventional n-bit binary output.

The flash converter is a simple concept, but its main advantage is the speed at which conversion can be achieved. Since the input is compared to all the reference values simultaneously the time required to perform a conversion is simply the response time for the comparators and the encoder. This *conversion time* is significantly shorter than the fastest alternative architectures. The disadvantage of the flash converter is the large number of comparators and resistors required, which means that these converters are relatively expensive. Furthermore, as the number of comparators increases, the voltage difference between the reference inputs of two adjacent comparators reduces, and the errors between reference levels caused by variations between the values of individual resistors must therefore be reduced. Since these variations are caused by slight differences in the sizes of different resistors any reduction in errors will only be achieved by using larger area resistors. Unfortunately, this simply further increases the cost of the final component. Overall, flash converters are therefore fast, but, expensive.

4.5.2 Successive-Approximation ADCs

The successive-approximation converter shown below operates by approximating the analogue input signal with a binary code. This binary code is successively revised by changing each bit in the code until the best approximation is achieved. At each step in the approximation, the present estimate of the binary value corresponding to the analogue input signal is saved in the successive approximation register. The contents of this register are converted to an analogue signal by a DAC so that a single comparator can determine whether the approximation is larger or smaller than the input signal.



A schematic diagram of a successive-approximation ADC and its internally generated analogue signal (solid line), and input signal (dashed).

As shown at the bottom of the figure, the first approximation sets the most significant bit (the MSB), of the successive approximation register to 1 and resets all the other bits (i.e. sets them to 0). If the DAC output (which is therefore equal, at this point, to half full-scale) is smaller than the analogue input, the MSB is left at 1; if the DAC output is too large, then the MSB is set to 0. In the next clock cycle, the *next* most significant bit is set (i.e. at

the DAC output is now equal to either $3/4$ or $1/4$ of full-scale, depending on whether the most significant bit was left on or not), and this new approximation is compared with the analogue input. Each successive bit is similarly tested. After the least significant bit has been tested, the conversion is complete and the output register contains the binary code.

If the accuracy of conversion is to equal the resolution of the converter, the input signal must remain constant within the analogue value of $1/2$ LSB during the conversion time

To quantify the limitation this places on the input signals that can be converted accurately assume that the input signal is a sinusoidal wave of frequency f Hz and peak-to-peak amplitude V_{REF}, i.e.

$$V_{in} = \frac{1}{2} V_{REF} \sin 2\pi f t.$$

For an n-bit converter, $1/2$ LSB (a simple estimate of the quantisation error) is equivalent to a voltage of

$$\frac{1}{2} V_{REF}/2^n.$$

The rate of change of the input signal is

$$\frac{\mathrm{d}V_{in}}{\mathrm{d}t} = \pi f V_{REF} \cos 2\pi f t.$$

The maximum rate of change occurs when the input is zero and is given by

$$\left|\frac{\mathrm{d}V_{in}}{\mathrm{d}t}\right|_{\max} = \pi f V_{REF}.$$

If the conversion time is t_c, then we must have

$$\left|\frac{\mathrm{d}V_{in}}{\mathrm{d}t}\right|_{\max} t_c \leq \frac{1}{2}\frac{V_{REF}}{2^n},$$

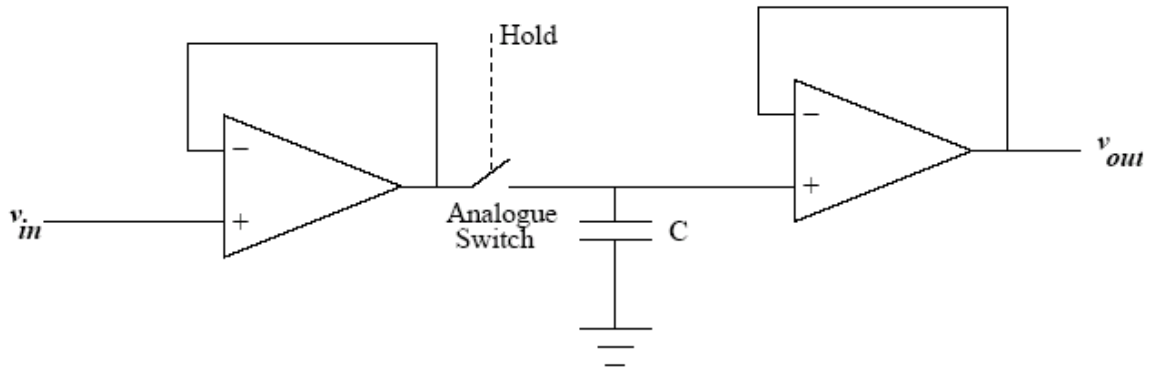
which can be re-written as

$$\pi f V_{REF} t_c \leq \frac{V_{REF}}{2^{n+1}},$$

and therefore requires that the input frequency is limited to

$$f_{\max} = \frac{1}{\pi t_c 2^{n+1}}.$$

For an 8-bit ADC with conversion time of $10\,\mu$s, this gives a maximum frequency of $62\,$Hz. This is obviously much too low for most applications. The problem that limits the maximum frequency that can be converted arises from the changes in the input signal during the conversion process. These changes can be avoided by using a *sample-and-hold* circuit just before the ADC input. As its name suggests this type of circuit *samples* the signal and then *holds* the sampled value until the conversion process is completed and a new sample is acquired.



A sample-and-hold circuit

The basic sample-and-hold circuit consists of an analogue switch and a storage capacitor, as shown above. The analogue switch is controlled by a signal, labelled Hold, which allows the input signal to pass through to the capacitor during the *aperture time* and disconnects it during the *hold time*. The value of the input signal v_{in} is therefore stored on the capacitor during the hold time. The choice of a value for this capacitor is a compromise between the need to minimise voltage changes caused by leakage currents during the hold interval (i.e. make C as large as possible) and the need to follow high-frequency input signals without them being low-pass filtered by the combination of the capacitor and the finite on-resistance of the switch (i.e. make C as small as possible). In order to reduce leakage currents during the hold time, to prevent voltage changes, the voltage on the capacitor is sensed using an op-amp configured as a voltage follower. Similarly, the speed of the circuit is increased by detecting the input signal via a second op-amp acting as unity gain buffer that reduces the source impedance driving the capacitor during the aperture time.

With a sample-and-hold circuit on the input to a successive-approximation A/D converter the maximum operating frequency of the converter is now given by

$$f_{\max} = \frac{1}{\pi t_a 2^{n+1}}$$

where t_a is the *aperture time* which can be just a few tens of nanoseconds. Hence input signals whose frequency is several tens of kHz can now be converted to binary format with this type of A/D converter.

4.6 Summary

The outputs from sensors and communications receivers are typically analogue signals that have continuously varying amplitudes. In many systems it is convenient to record and/or process these signals using a digital circuit, which may be a microcontroller, a microprocessor or computer. In a digital circuit the signal is represented as a sequence of binary numbers, with each number representing the amplitude of the signal at a specific sampling instant.

A digital to analogue converter (DAC) converters a digital input signal, represented as a binary number, to an analogue voltage (or current) that is proportional to the value of this input. A DAC can be created using an R-$2R$ ladder and an op-amp.

Conversion from an analogue signal to a digital (binary) number is performed by an analogue-to-digital converter (ADC). The analogue signal input to an ADC is *quantised* by dividing the continuous input range into 2^n discrete steps or code ranges (for an n-bit ADC). This results in rounding errors called quantisation noise, which has a maximum value of $V_{max}/2^{n+1}$.

In a flash ADC the input voltage is compared in parallel with many different reference voltages. The resulting system is conceptually simple, fast but expensive.

A successive-approximation ADC operates by comparing the analogue input signal to the output of a DAC, and thus determines successive bits of the output, starting with the MSB, until all bits including the LSB of the output

are determined. The result is only valid if the input remains approximately constant during the time taken to perform the conversion. Therefore a sample and hold circuit is used to sample the input voltage and hold it constant during conversion.