Foreword

I'm going to skip a lot of detail simply because i'm trying to fast forward you to the useful stuff.
I'm assuming you're not trying to build a transistor, design anything requiring difficult mathematics/physics, design a complex circuit or even connect up anything affected by the more complex aspects of current/voltage/electrons.

- I am writing this in case you want to do more than just randomly plug wires into modules, (specifically logic, sequencing and switching modules).
- I am also writing this for those who may want to make modifications to RYO modules or to build expanders to RYO modules with the expansion headers/pads for adding expansion options provided.

As such there's a mixture of basic information on
- how conceptual logic works,
- how electronic logic works,
- how more complex components built of logic work, and,
-how more complex components built of those components work.
This goes right up to near the level of how full blown computers work (not in extreme detail) so as to cover how the most complex RYO modules like the VC Sequencer work, both theoretically and practically.

This also might provide inspiration for some of the more adventurous wigglers of modular synths to build patches involving whole serge style patch programmable, complex, multiple module assemblies that have higher order functions similar to some of the computer parts or even simple computers described later in this document.

You may not need to know all of this or certain specific parts may only be of interest so you can read this from anything as an interesting look up to a single specific 'how does this work' to a full blown book on how to build a computer. Do feel free therefore to skip anything, but, you may find skipping stuff you don't already know leaves you confused later on, because near everything in here is in here for a reason that will make sense by the time you've reached later parts, if not the end.

Contents and Why

- **Transistors** *(only a little info)* because RYO logic modules, the more complex RYO modules, and computers are built of or based on parts built of them.
- **Logic** Theory *(and a little of the electronics - no math here)* and because RYO's simplest modules are the logic modules and the concepts allow grasping of the more complex part of sequencers and computers.
- **Combinational Logic** because it explains specifically the operation and building of encoders/decoders, multiplexers and some other stuff used in RYO modules and computers.
- **Sequential logic** and multivibrators, as above because both go to explain how the stuff used in some RYO modules and computers work, specifically counters and shift registers.
- some types of **DAC's and ADC's** because they are used in the RYO VC SEQ and PATHS, some other parts of RYO sequencing, logic and switching modules and also in certain computing and audio applications.
- the rest is mostly just for pure interest purposes - it doesn't specifically apply to RYO mdoules as such.

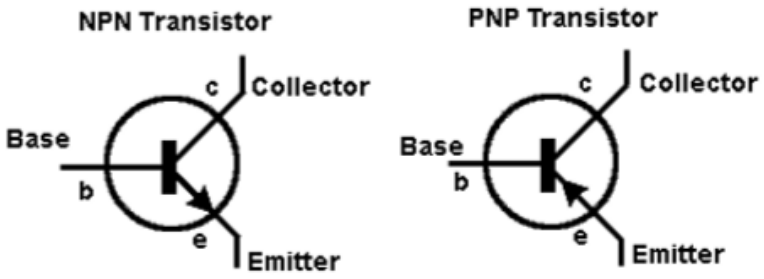
Bipolar transistors

The simplest form of transistor to understand is the bipolar junction transistor (**BJT**). *For our purposes, all you really need to be aware of (vaguely), is the following:*

- there's 3 parts to a typical transistor: **collector**, **emitter**, and **base**.

- signal typically flows **from** collector **to** emitter and is **altered** by the base, such as in a voltage controlled resistor or switch circuit.

The three leads of a bipolar transistor are called the Emitter, Base, and Collector:



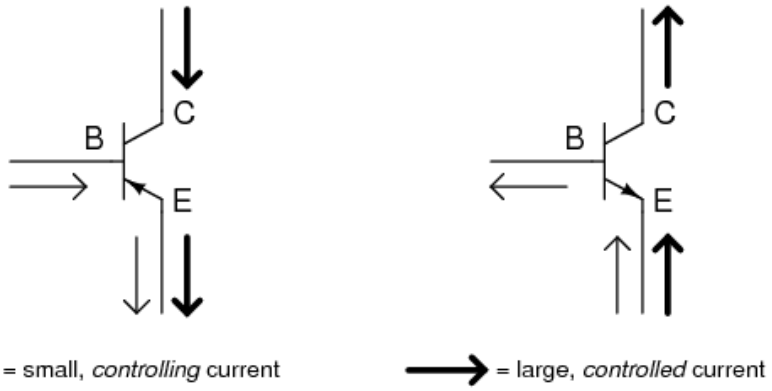
- Transistors function as current regulators by allowing a small current to control a larger current.

- The amount of current allowed between collector and emitter is primarily determined by the amount of current moving between base and emitter.

Note, currents go **against the emitter arrow symbol**.

the base is generally 'opposite' the collector and emitter:
- in a typical BJT it'll be **positive collector and emitter** and negative base (called a **PNP** transistor because of physics stuff), or;
- vice versa, **negative collector and emitter** and positive base (called an **NPN** transistor).

- When a transistor has **zero current** through it, it is said to be in a state of **cutoff** (fully nonconducting).
- When a transistor has **maximum current** through it, it is said to be in a state of **saturation** (fully conducting).



At most basic, as a switch:

- the **base** is the **input** of the switch;
- the **emitter** is the **output** of the switch;
- the **control signal** for the switchgoes into the **collector**;
putting a binary signal into the base will turn it on and off.

(in fact, there is a 'transient area' between on and off which can be *predictably* controlled.)
- however, it's important to note that even as a switch, it works like a resistor, i.e.,
- when **on** it's **low resistance**, and when **off** it's **high resistance**, (signal flow often being diverted to the base, where low is ground).

- when in this transient area, (called the saturation zone), between on and off, resistance is variable.
- but, although it doesn't work like a perfect resistor, it can be used as a variable resistor for voltage control purposes.

(this aside is just some further detail on transistors that isn't really relevant to logic so feel free to ignore. But, it is useful for module design/modification in general since it is how they are used for voltage control):

- in transistors, signal can also flow from the base to both the collector and emitter.
- (as in amplifiers, collector acting as non inverting, emitter as inverting].)

as an amplifier:
- signal goes into the base
- amount of current flowing into the collector controls gain.
- the emitter is typically grounded,

however, if the output is taken from the emitter, it will be inverted.
[as it is the opposite of the collector – which, due to a 'virtual ground' being in place is going to be 'negative'];

> (when creating a bipolar (+ve/-ve) supply from a unipolar source, a point between negative and positive - a zero 'midpoint' will exist, even though not connected to earth. This is a 'virtual' ground.)

note;
- the input signal itself is typically 'biased' somewhat. as as transistors are unipolar devices, when used alone, a resistor is used to set the midpoint between cutoff and saturation to allow linear operation.
- then, the output is then usually sent through a capacitor 'in series' with (in between) the signal to remove this 'bias' after the amplifying 'gain' stage.

> using a capacitor this way is what in modular synthesis is often heard referred to as called 'ac coupling':
>
> – this coupling or 'blocking' is just a removal of static/slow changing voltages.
> - a capacitors property of taking to charge/discharge can be used since static current will not pass the 'gap' in the two insulators inside the capacitor, whereas a moving one will.
>
> [see diagram for explanation]
>
> effectively, although no current is actually flowing in or out, change of current is still able to occur:
>
> - This is because, as the forwards-backwards movement of the pistons represents, the changing ac movement can happen, but,
> - the pistons cant move in one direction forever - as would be required for continuous, dc flow.

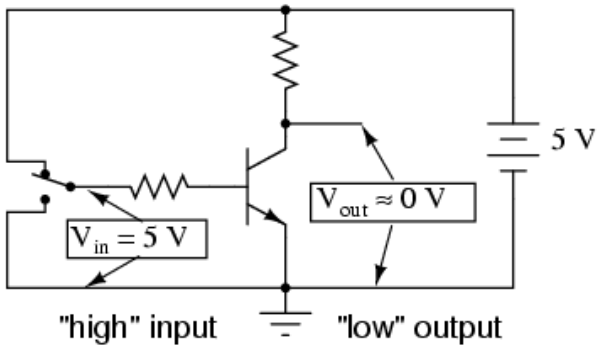
So know you know!

---

logic

In this circuit, the transistor is in a state of saturation due to the applied input voltage (5 volts) through the two-position switch

Transistor in saturation



0 V = "low" logic level (0)

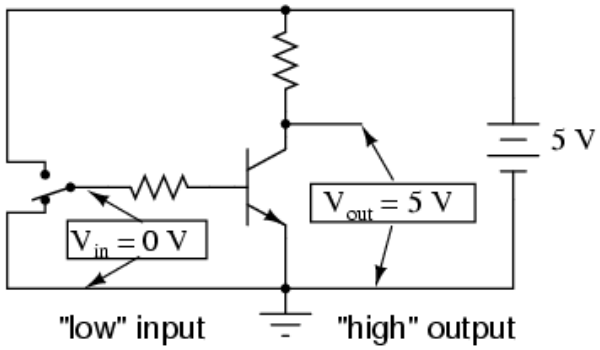
5 V = "high" logic level (1)

- the **saturated transistor drops very little voltage between collector and emitter**, resulting in an **output voltage of** (practically) **0 volts**.

- using this circuit to represent binary bits, it would be said that the **input signal is a binary "1"** and that the **output signal is a binary "0"**.
- Any voltage close to **full supply voltage** (*measured in reference to ground*) **is considered a "1"** and a **lack of voltage is considered a "0"**.
- Alternative terms for these voltage levels are **high (same as a binary "1")** and **low (same as a binary "0")**.
- A general term for the representation of **a binary bit by a circuit voltage is 'logic level'**.

- Moving the switch to the other position, **applying a binary "0" to the input we receive a binary "1" at the output:**

Transistor in cutoff



0 V = "low" logic level (0)

5 V = "high" logic level (1)

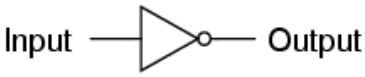
- **What we've created here** with a single transistor is a circuit generally **known as a 'logic gate'**, or simply 'gate'.

- The gate shown here with the single transistor **is known as an 'inverter', or NOT gate,** because it outputs the exact opposite digital signal as what is input.

- For convenience, gate circuits are generally represented by their own symbols rather than by their constituent transistors and resistors;

- The following is the symbol for an inverter:

Inverter, or NOT gate



- One common way to express the particular function of a gate circuit is called a truth table.

- Truth tables show all combinations of input conditions in terms of logic level states (either "high" or "low," "1" or "0," for each input terminal of the gate), along with the corresponding output logic level, either "high" or "low".

- For the inverter, or NOT, circuit just illustrated, the truth table is very simple indeed:

NOT gate truth table

Input	Output
0	1
1	0

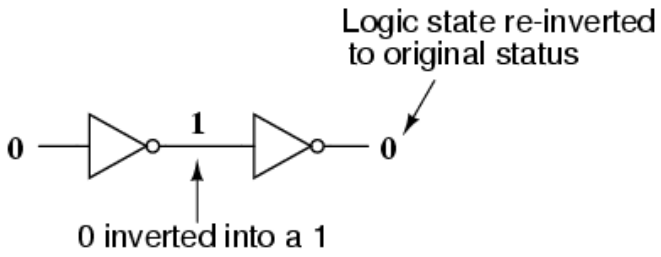
Other Gates

- **Two inverter, or NOT, gates connected in "series"** so as to invert, then re-invert, a binary bit **perform the function of a buffer**.

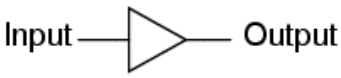
- **Buffer gates** merely **serve the purpose of signal amplification**: taking a **"weak"** signal source, that isn't capable of sourcing or sinking much current, and **boosting** the current 'capacity' of the signal so as to be able to drive a 'load'.

- Buffer circuits are symbolized by a triangle symbol with no inverter "bubble.

Double inversion



"Buffer" gate



Input	Output
0	0
1	1

- **Adding more input terminals to the circuit(s) gives more logic gate possibilities:**

- **With a single-input gate such as the inverter or buffer, there can only be two possible input states: either the input is "high" (1) or it is "low" (0).**

- **A two input gate has four possibilities (00, 01, 10, and 11).**

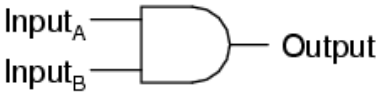
The AND gate

- One of the easiest multiple-input gates to understand is the AND gate,

- **The output of the AND gate will be "high" (1) if, and only if, all inputs (first input and the second input) are "high" (1).**

- **If any input(s) are "low" (0), the output is guaranteed to be in a "low" state as well:**

2-input AND gate



A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

- *(note, AND gates can be made with more than two inputs)*

<u>The NAND gate</u>

- The NAND gate is a variation on the idea of the AND gate.

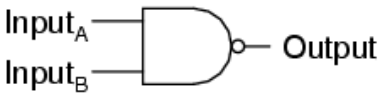
- The word "NAND" is a verbal contraction of the words NOT and AND.

- Essentially, a NAND gate behaves the same as an AND gate with a NOT (inverter) gate connected to the output terminal.

- To symbolize this output signal inversion the NAND gate symbol has a bubble on the output line.

- The truth table for a NAND gate is as one might expect, exactly opposite as that of an AND gate:

2-input NAND gate



A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Equivalent gate circuit

- (*As with AND gates, NAND gates can be made with more than two inputs, but the same general principle applies*):

- The o**utput will be "low" (0) if and only if all inputs are "high" (1)**.

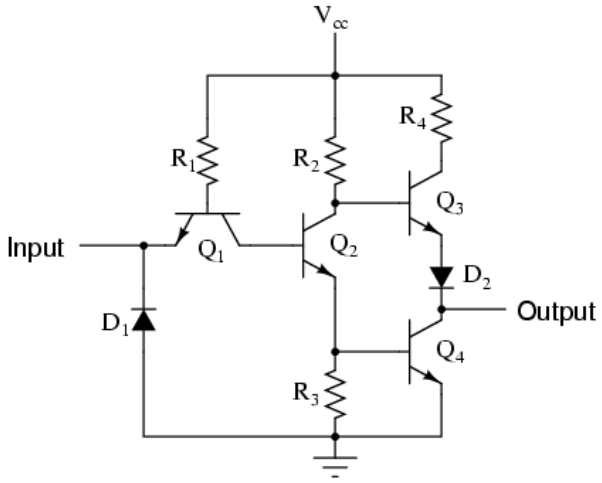
- **If any input is "low" (0), the output will go "high" (1).**

TTL

- Real inverter circuits contain more than one transistor to maximize voltage gain (i.e. to ensure that the final output transistor is either in full cutoff or full saturation), and other components designed to reduce the chance of accidental damage.

- Shown here is a schematic diagram for a real inverter circuit, complete with all necessary components for efficient and reliable operation:

Practical inverter (NOT) circuit

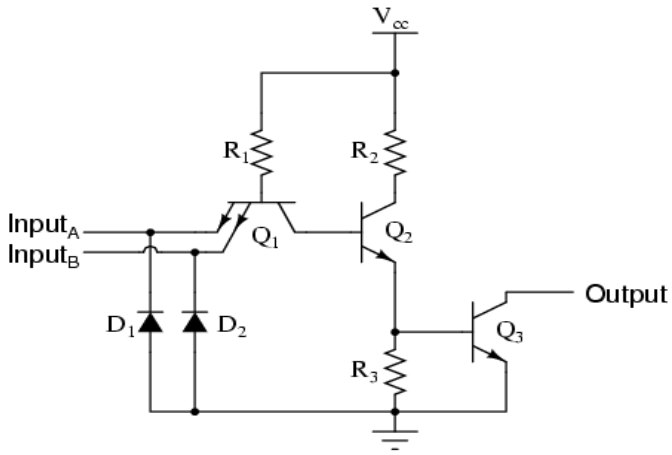


- Gate circuits constructed of resistors, diodes and bipolar transistors as illustrated in this section are called TTL.

- TTL is an acronym standing for Transistor-to-Transistor Logic.

- Suppose we altered our basic inverter circuit, adding a second input terminal just like the first:

(A simple transistor structure is inadequate to simulate the three junctions necessary in this network, so a different transistor (and symbol) is needed); This transistor has one collector, one base, and two emitters like this:



- **The only difference to the simple inverter circuit is a second input terminal** connected in the same way to the base of transistor Q2.

- **We can say that each of the inputs will have the same effect on the output,**

- i.e. if either of the inputs are grounded, transistor Q2 will be forced into a condition of cutoff, thus turning Q3 off and the output goes "high".

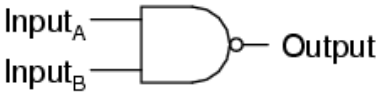
- In any case where there is a grounded ("low") input, the output is guaranteed to be "high".

- Conversely, the only time the output will ever go "low" is if transistor Q3turns on, which means transistor Q2 must be turned on (saturated).

- This means neither input can be diverting R1 current away from the base of Q2. The only condition that will satisfy this requirement is when both inputs are "high".

- **Collecting and tabulating these results into a truth table, we see that the pattern matches that of the NAND gate:**

NAND gate



A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

- **In other words, a TTL NAND gate can be made by taking a TTL inverter circuit and adding another input.**

- As a result, An AND gate may be created by adding an inverter stage to the output of the NAND gate circuit.

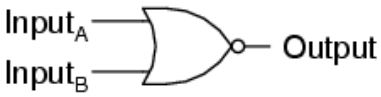
OR Gates

- The OR gate is so-called because the output of this gate will be "high" (1) if *any* of the inputs (first input ***OR*** the second input) are "high" (1).

- The output of an OR gate goes "low" (0) if and only if all inputs are "low" (0).

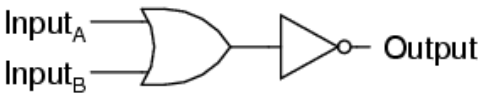
- Therefore, **the NOR gate is an OR gate with its output inverted,** just like a NAND gate is an AND gate with an inverted output.

2-input NOR gate



A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

Equivalent gate circuit



- Exclusive-OR gates output a "high" (1) logic level if the inputs are at *different* logic levels, either 0 and 1 or 1 and 0. (Exclusively one OR, Exclusively the other).

- Conversely, they output a "low" (0) logic level if the inputs are at the *same* logic levels.

- The Exclusive-OR (sometimes called XOR) gate has both a symbol and a truth table pattern that is unique:

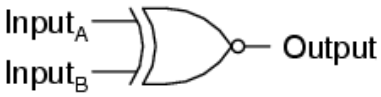
Exclusive-OR gate

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

- **The Exclusive-NOR gate, otherwise known as the XNOR gate is equivalent to an Exclusive-OR gate with an inverted output.**

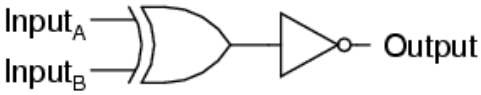
- The truth table for this gate is exactly opposite as for the Exclusive-OR gate:

Exclusive-NOR gate



A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

Equivalent gate circuit



To sum up, the six most common and useful logic gates:

- An **AND** gate: output is **"high" only if** first input *and* second input are **both "high"**
- A **NAND** gate: output is ***not*** **"high" if both** the first input *and* the second input are **"high"**

- An **OR** gate: output is **"high" if** input **A** ***or*** input **B are "high"**
- A **NOR** gate: output is ***not*** **"high"** if **either** the first input *or* the second input **are "high"**

- An **XOR** gate: output **is "high"** if the **input** logic levels are ***different***
- An **XNOR** gate: output **is "high"** if the **input** logic levels are the ***same***

Combinational Logic

- In mathematics a **combination** is an unordered set, which is a formal way to say that **nobody cares which order the items came in**.

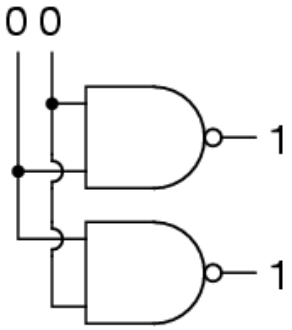
- With combinational logic, the circuit produces the same output regardless of the order the inputs are changed.

- Combinational logic perform **functions like arithmetic, logic, or conversion**.

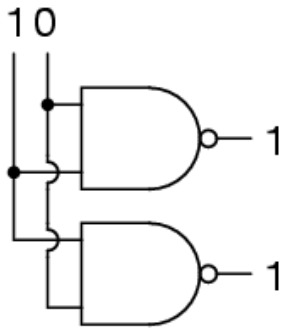
- Each logic gate discussed previously is a combinational logic function.

- See how two NAND gates work if provided with inputs in different orders.

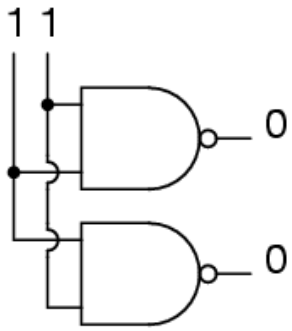
Beginning with both inputs being 0:



Then setting one input high:



Then setting the other input high:



So **NAND gates do not care about the order of the inputs**, and **the same is true of** all the other gates covered up to this point (**AND, XOR, OR, NOR, XNOR, and NOT**).

A Half-Adder

(please note: the mathematics isn't fully essential in detail to understand this section – i've tried to ensure this is graspable from just the 'symbolic' representations of how-to-build useful logic devices/building blocks)

a quick note [or perhaps refresher] on binary mathematics:

- Instead of ten different cipher symbols **we only have *two* cipher symbols**,
- **The two allowable cipher symbols** for the binary system of numeration **are "1" and "0".**
- These ciphers are arranged **right-to-left in doubling values of weight**.
- **The rightmost place is the *ones* place**, just as with decimal notation.
- **We refer to each cipher position in binary as a *bit.***

- The bit on the far right side is called the Least Significant Bit (LSB)`
- The bit on the far left side is called the Most Significant Bit (MSB)

- **A bit value of "1"** means that the respective place weight **gets added to the total value**, and,
- **A bit value of "0"** means that the respective place weight **does *not* get added to the total value**.

Adding binary numbers is very similar to the longhand addition of decimal numbers:

- As with decimal numbers, start by adding the bits (digits) one column at a time, from right to left.

```
  0     1     0
+ 0   + 0   + 1
  -     -     -
  0     1     1
```

- Also as with decimal addition, when the sum in one column is a two-bit (two-digit) number:
- The least significant figure is written as part of the total sum.
- The most significant figure is written as "carried" to the next left column.

```
  1       1
+ 1     + 1
  --    + 1
  10      --
          11
```

- These bits that are carried to the next column are called carry bits.
- Consider the following more complex examples:

```
          <-- No              11  1 <---These bits -----> 11
  1001101   carry           1001001    are 'carry'     1000111
+ 0010010   bits:        + 0011001    bits:         + 0010110
---------   follow        ---------    follow them   ---------
  1011111   down/along     1100010    down/along      1011101
```

- The addition problem on the left did not require any bits to be carried since the sum of bits in each column was either 1 or 0, not 10 or 11.
- In the other two problems, there definitely were bits to be carried - where

columns added resulting in 10 or 11 then a 1 was carried to the next column and then the process was repeated in turn for that next column.

(As you may be now beginning to see, there are ways that electronic circuits can be built to perform this very task of addition, by representing each bit of each binary number as a either high [for a 1], or low [for a 0] voltage signal. This is the foundation of all the arithmetic which modern digital computers perform.)

- An example of useful combinational logic is a device that can add two binary digits together.

- calculating what the answers should be:

0 + 0 = 0 0 + 1 = 1 1 + 0 = 1 1 + 1 = 10

- So two inputs (a and b) and two outputs will be needed.

- The low output will be called Σ because it represents the sum, and the high output will be called C_{out} because it represents the carry out.

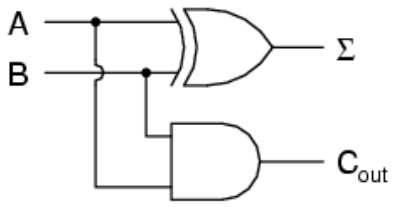
- The truth table is

A	B	Σ	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- looking at the results:

- **The Σ column is an XOR gate**, and,
- **The C_{out} column is an AND gate.**

- Therefore we can construct the device like so:



- **This device is called a half-adder** (for reasons that will make sense shortly).

A Full Adder

The half-adder is extremely useful until you want to add more that one binary digit quantities.

The slow way to develop a two binary digit adders would be to make a truth table and reduce it.
Then when you decide to make a three binary digit adder, do it again.
Then when you decide to make a four digit adder, do it again.
Then when ...

The circuits would be fast, but development time would be slow.

Looking at a two binary digit sum shows what we need to extend addition to multiple binary digits:

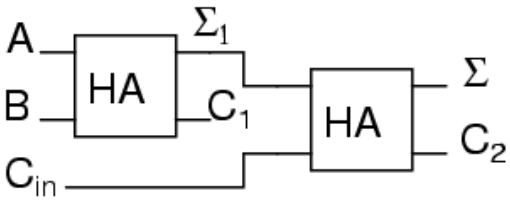
```
  11
   11
   11
  ---
  110
```

- Note how many inputs the middle column uses:

- The adder needs three inputs; a for one row, b for the second row, and the carry from the previous sum.

- Using our two-input adder to build a three input adder:

Σ (the total sum) is the easy part.
Normal arithmetic tells us that if $\Sigma = a + b + C_{in}$ and $\Sigma_1 = a + b$, then $\Sigma = \Sigma + C_{in:}$



- But where do with C_1 and $C_{2\ go}$?

- Looking at three input sums its possible to calculate:

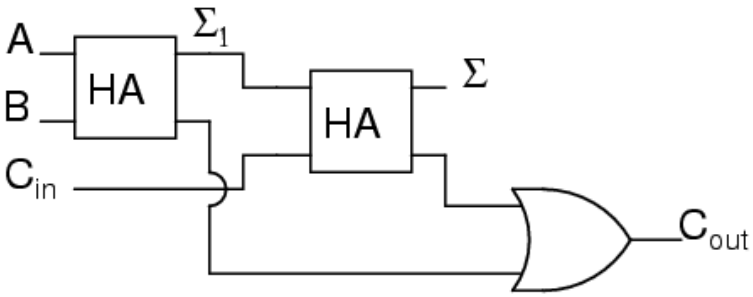
```
 a + b + Cin=  ?
 0 + 0 + 0 =  0
 1 + 0 + 0 =  1
 0 + 1 + 0 =  1
 0 + 0 + 1 =  1
 1 + 0 + 1 =  1
 1 + 1 + 0 = 10
 0 + 1 + 1 = 10
 1 + 1 + 1 = 11
```

- In order to calculate the MSB bit, note that it is 1 in both cases when a + b produces a C_1.
- Also, the MSB bit is 1 when a + b produces a Σ_1 and C_{in} is a 1.

- To achieve this it requires a carry on occasions when when either C_1 **OR** (Σ_1 AND C_{in}) are 1 (high).

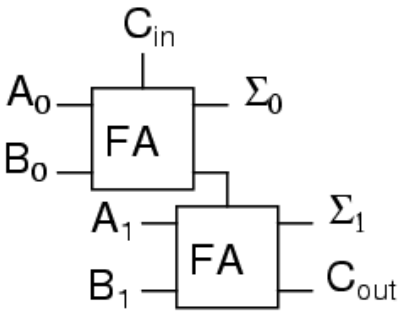
- This means to complete the adder such that we can process C_1 and C_2 we require an OR gate connected to the two carry outs of the half adders.

- The complete three input adder is:



- Note, for some designs, being able to eliminate one or more types of gates can be important, and you can replace the final OR gate with an XOR gate without changing the results.

- Connecting two adders to add 2 bit quantities:



- There are several reasons for using a FA for the LSB:

- One being that we can then allow a circuit to determine whether the lowest order carry should be included in the sum.

- This allows for the chaining of even larger sums.

- Consider two different ways to look at a four bit sum.

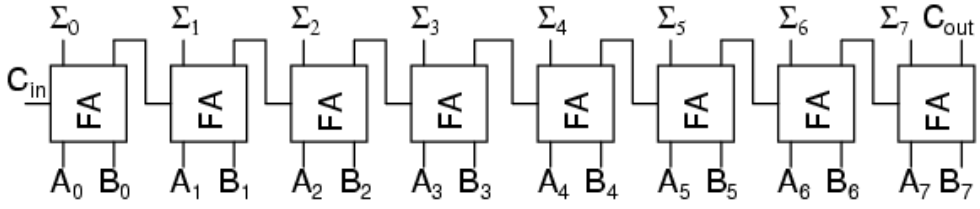
```
111            1<-+  11<+-
 0110             |  01  |  10
 1011             |  10  |  11
-----          -  | ---- | ---
10001          1  +-100  +-101
```

- If the program is allowed to add a two bit number and remember the carry for later, and then use that carry in the next sum, then the program can add any number of bits the user wants even though it only has a two-bit adder.

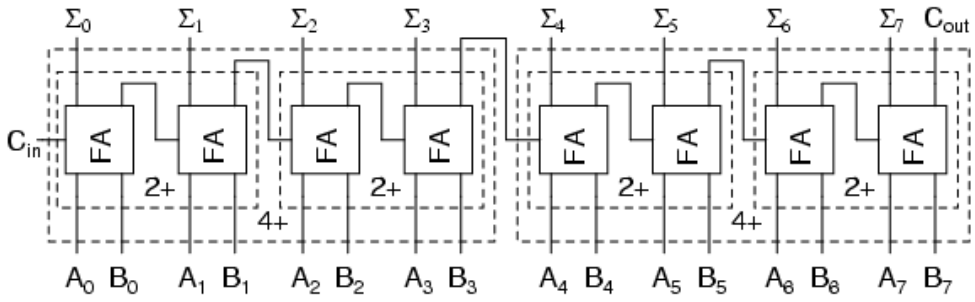
- Small PLCs can also be chained together for larger numbers.

- These full adders can also can be expanded to any number of bits space allows.

- As an example here's how to do an 8 bit adder:



- This is the same result as using the two 2-bit adders to make a 4-bit adder and then using two 4-bit adders to make an 8-bit adder:



- Each "2+" is a 2-bit adder and made of two full adders.
- Each "4+" is a 4-bit adder and made of two 2-bit adders.
- And the result of two 4-bit adders is the same 8-bit adder previously built of full adders.

- For any large combinational circuit there are generally two approaches to design:
- you can take simpler circuits and replicate them; or,
- you can design the complex circuit as a complete device.

- Using simpler circuits to build complex circuits allows a you to spend less time designing but then requires more time for signals to propagate through the transistors.
- The 8-bit adder design above has to wait for all the C_{xout} signals to move from $A_0 + B_0$ up to the inputs of Σ_7.

- If a designer builds an 8-bit adder as a complete device simplified to a sum of products, then each signal just travels through one NOT gate, one AND gate and one OR gate.
- A seventeen input device has a truth table with 131,072 entries, and reducing 131,072 entries to a sum of products will take some time.

- When designing for systems that have a maximum allowed response time to provide the final result, you can begin by using simpler circuits and then attempt to replace portions of the circuit that are too slow.
- That way you spend most of your time on the portions of a circuit that matter.

Decoders

- A decoder is a circuit that changes a code into a set of signals.
- It is called a decoder because it does the reverse of encoding, but this will study encoders and decoders beginning with decoders because they are simpler to design.
- A common type of decoder is the line decoder which takes an n-digit binary number and decodes it into 2^n data lines

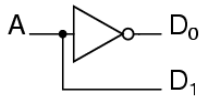
(*dont panic at the mathematics – the truth tables and logic gate diagrams are still ok to follow*).

- The simplest is the 1-to-2 line decoder.
- The truth table is:

A	D_1	D_0
0	0	1
1	1	0

- A is the address and D is the dataline.

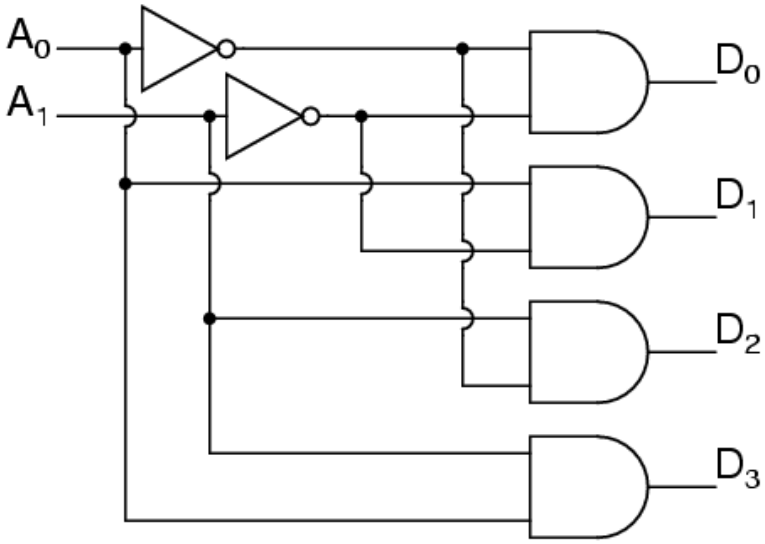
- D_0 is NOT A and D_1 is A. The circuit looks like



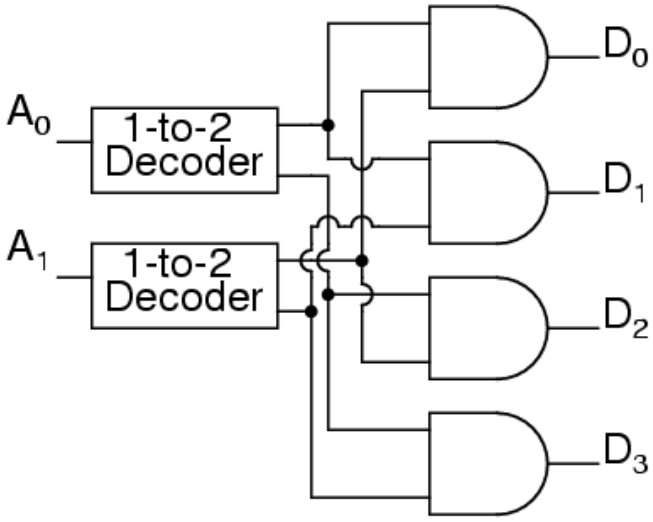
Only slightly more complex is the 2-to-4 line decoder. The truth table is

A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

- Developed into a circuit it looks like:



- Larger line decoders can be designed in a similar fashion;
- but, just like with the binary adder there is a way to make larger decoders by combining smaller decoders.
- An alternate circuit for the 2-to-4 line decoder is:



- Replacing the 1-to-2 Decoders with their circuits will show that both circuits are equivalent. - In a similar fashion a 3-to-8 line decoder can be made from a

1-to-2 line decoder and a 2-to-4 line decoder;
- And, likewise, a 4-to-16 line decoder can be made from two 2-to-4 line decoders.

- A typical application of a line decoder circuit is to select among multiple devices.
- A circuit needing to select among sixteen devices could have sixteen control lines to select which device should "listen".
- With a decoder only four control lines are needed.

Encoder

- An encoder is a circuit that changes a set of signals into a code.
- Beginning with making a 2-to-1 line encoder truth table by reversing the 1-to-2 decoder truth table:

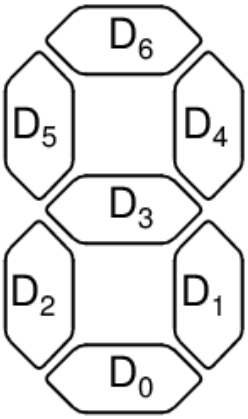
D_1	D_0	A
0	1	0
1	0	1

- This truth table is a little short, because a complete truth table would be:

D_1	D_0	A
0	0	
0	1	0
1	0	1
1	1	

- One question to be answered is what to do with those other inputs;
- Should they be ignored or should they generate an additional error output?
- In many circuits this problem is solved by adding sequential logic in order to know not just what input is active but also which order the inputs became active.

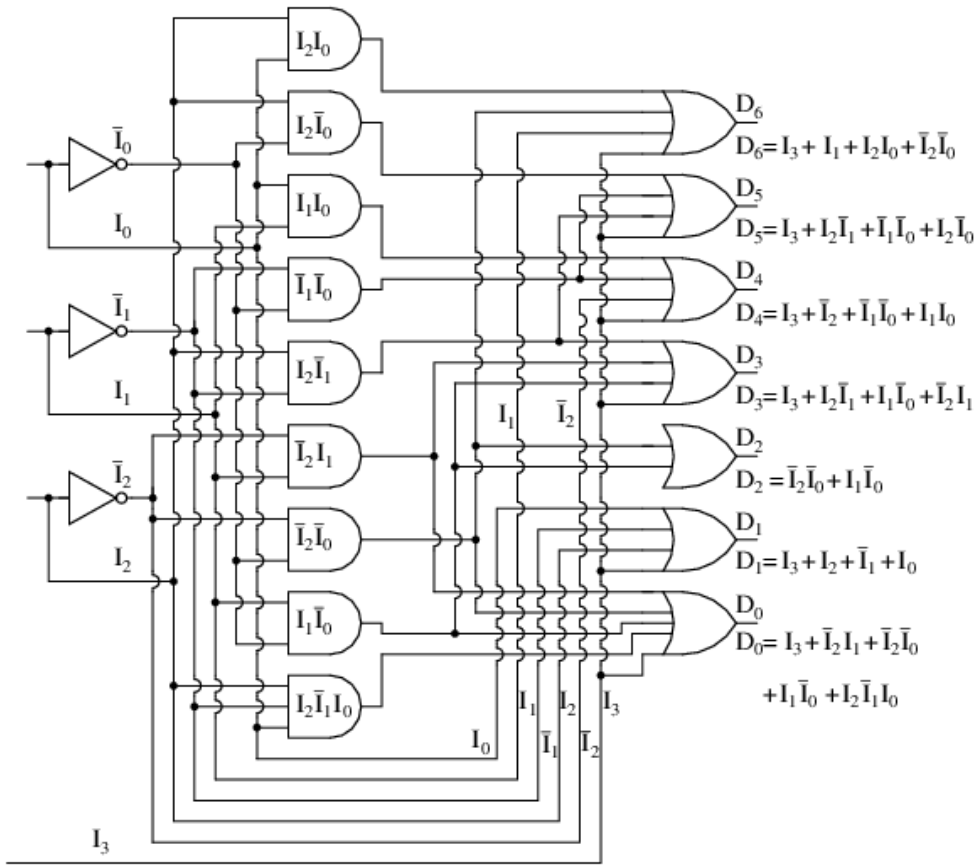
- A specific useful application of combinational encoder design is a binary to 7-segment encoder;
- The seven segments is generally familiar as the digit on a 'digital clock' type display:



- The truth table is:

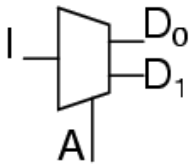
I_3	I_2	I_1	I_0	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	1	1	1	0	1	1	1
0	0	0	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0	1
0	0	1	1	1	0	1	1	0	1	1
0	1	0	0	0	1	1	1	0	1	0
0	1	0	1	1	1	0	1	0	1	1
0	1	1	0	1	1	0	1	1	1	1
0	1	1	1	1	0	1	0	0	1	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

- Deciding what to do with the remaining six entries of the truth table is easier with this circuit.
- This circuit should not be expected to encode an undefined combination of inputs, so we can leave them as "don't care" when we design the circuit:



Demultiplexers

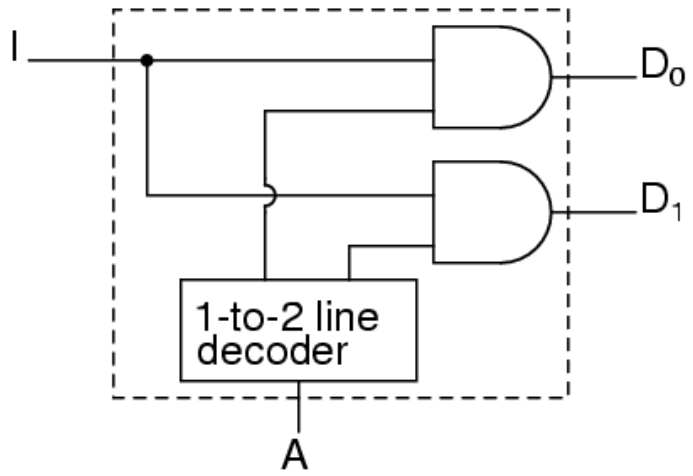
- A demultiplexer, sometimes abbreviated dmux, is a circuit that has one input and more than one output.
- It is used when a circuit wishes to send a signal to one of many devices.
- This description sounds similar to the description given for a decoder, but a decoder is used to select among many devices while a demultiplexer is used to send a signal among many devices.
- A demultiplexer is used often enough that it has its own schematic symbol



- The truth table for a 1-to-2 demultiplexer is

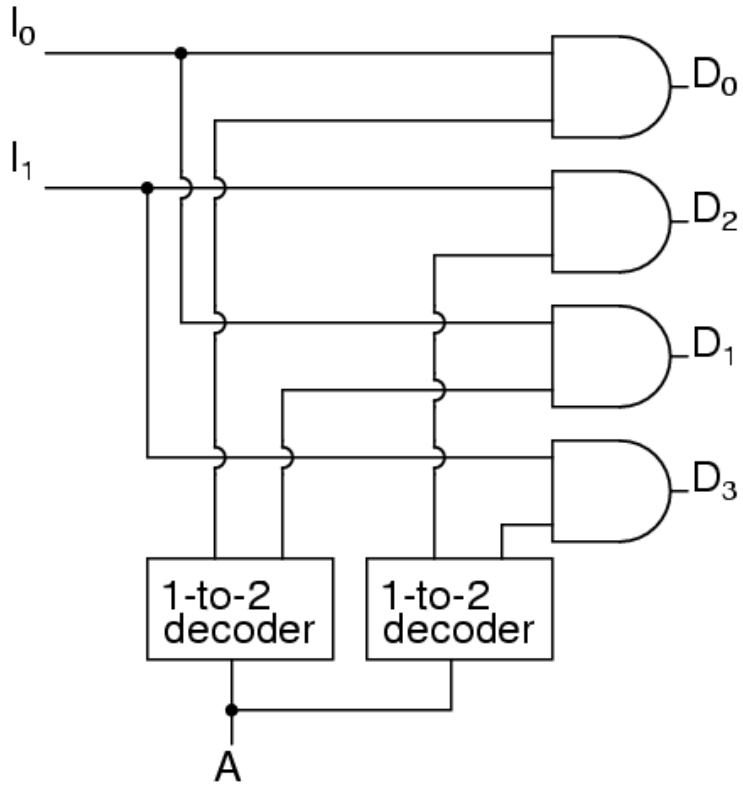
I	A	D_0	D_1
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

- Using the 1-to-2 decoder as part of the circuit, the circuit can be expressed as:

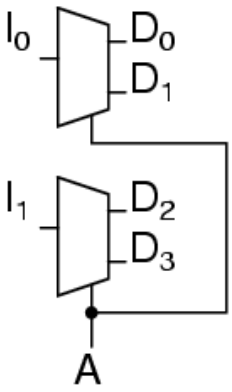


- This circuit can be expanded two different ways.
- You can increase the number of signals that get transmitted, or you can increase the number of inputs that get passed through.
- To increase the number of inputs that get passed through just requires a larger line decoder. - Increasing the number of signals that get transmitted is simpler.

- As an example, a device that passes one set of two signals among four signals is a "two-bit 1-to-2 demultiplexer".
- Its circuit is:

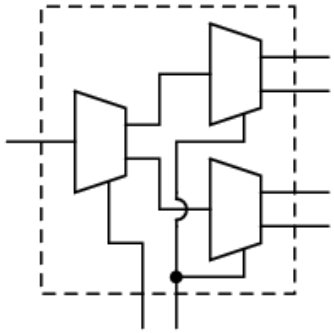


- or the circuit can be expressed as:



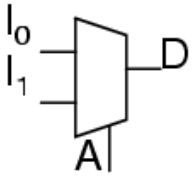
- this shows that it could be two one-bit 1-to-2 demultiplexers without changing its expected behavior.

- likewise a 1-to-4 demultiplexer can be built from 1-to-2 demultiplexers as follows:



Multiplexers

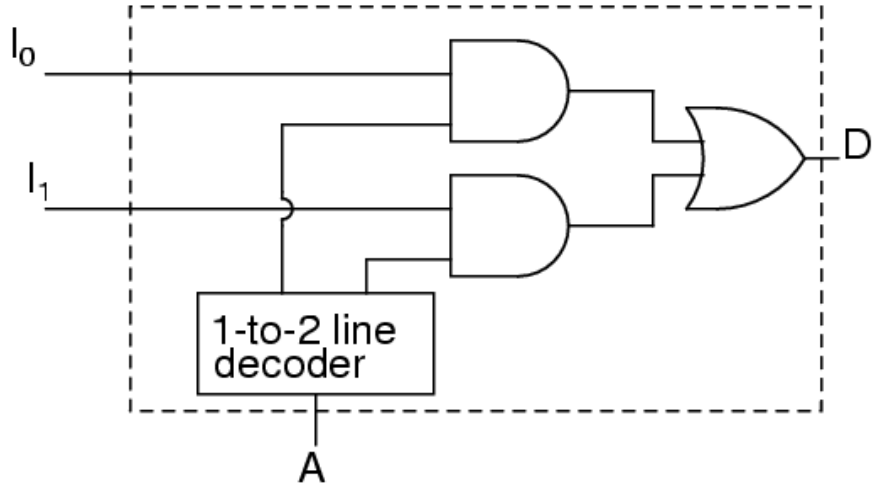
- A multiplexer, abbreviated mux, is a device that has multiple inputs and one output.
- The schematic symbol for multiplexers is:



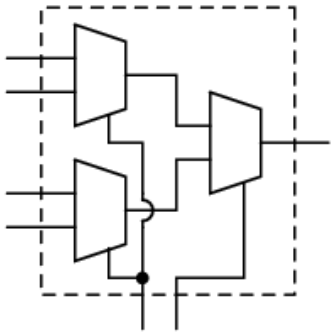
- The truth table for a 2-to-1 multiplexer is

I_1	I_0	A	D
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Using a 1-to-2 decoder as part of the circuit, the circuit can be expressed as:



- Multiplexers can also be expanded with the same naming conventions as demultiplexers.
- A 4-to-1 multiplexer circuit is:

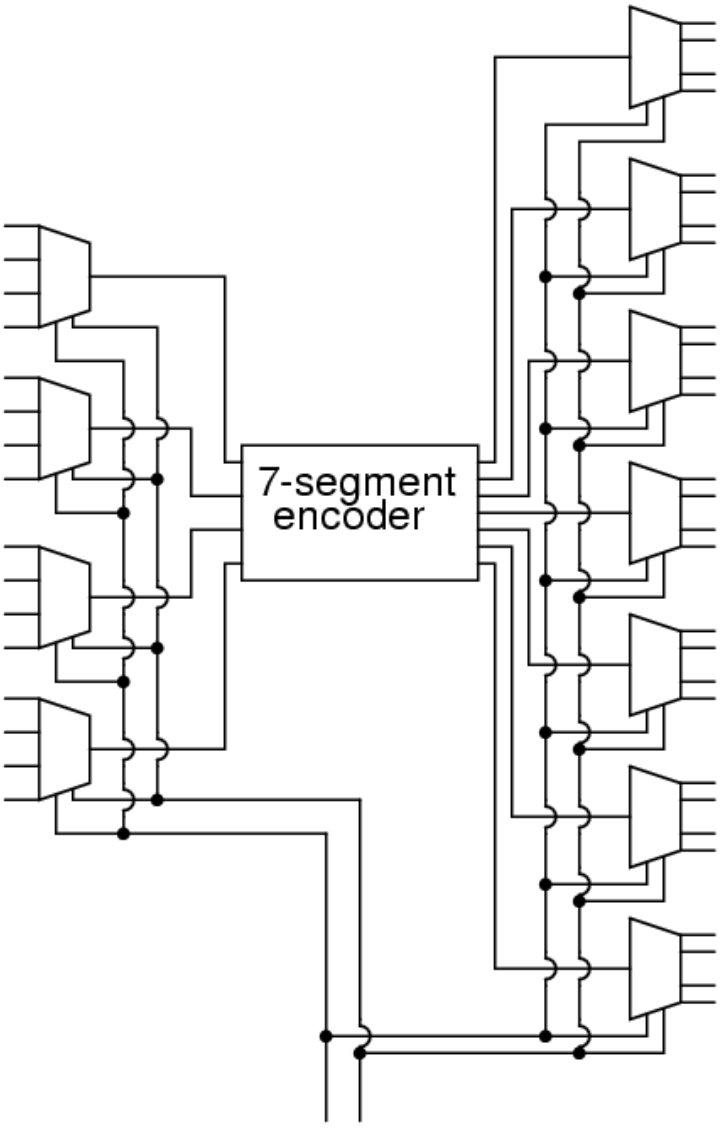


- That is the formal definition of a multiplexer.
- Informally, there is a lot of confusion.
- Both demultiplexers and multiplexers have similar names, abbreviations, schematic symbols and circuits, so confusion is easy.
- The term multiplexer, and the abbreviation mux, are often used to also mean a demultiplexer, or a multiplexer and a demultiplexer working together.
- So when you hear about a multiplexer, it may mean something quite different.

Using multiple combinational circuits

- As an example of using several circuits together, a device that will have 16 inputs, representing a four digit number, to a four digit 7-segment display can be made using just one binary-to-7-segment encoder.

- First, the overall architecture of our circuit provides looks like the description provided:



- Follow this circuit through, it can be confirmed that it matches the description given above. - There are 16 primary inputs.
- There are two more inputs used to select which digit will be displayed.
- There are 28 outputs to control the four digit 7-segment display.
- Only four of the primary inputs are encoded at a time.
- there is a potential question though:

- When one of the digits are selected, what do the other three digits display?
- Reviewing the circuit for the demultiplexers and noticing that any line not selected by the A input is zero. the other three digits are blank.
- So, there actually isn't a problem, only one digit displays at a time.

- To gain some perspective on the complexities of large circuits:

- Note how quickly this large circuit is developed from smaller parts.
- This is true of most complex circuits: they are composed of smaller parts allowing a designer to abstract away some complexity and understand the circuit as a whole.
- Sometimes a designer can even take components that others have designed and remove the detail design work.

- In addition to the added quantity of gates, this design suffers from one additional weakness. You can only see one display one digit at a time.
- If there was some way to rotate through the four digits quickly, you could have the appearance of all four digits being displayed at the same time.
- That is a job for a sequential circuit.

Sequential Logic

- There are circuits which depend on the when the inputs change, these circuits are called sequential logic.

- Sequential logic circuits make sure everything happens in order.

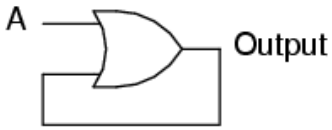
MULTIVIBRATORS

Digital logic with feedback

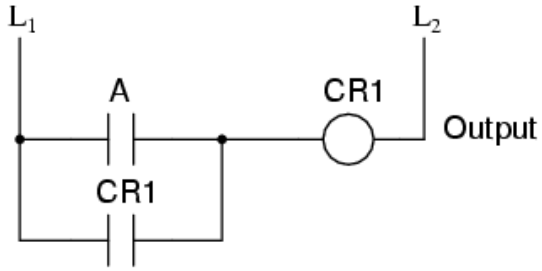
- With simple gate and combinational logic circuits, there is a definite output state for any given input state.
- Take the truth table of an OR gate, for instance:

- For each of the four possible combinations of input states (0-0, 0-1, 1-0, and 1-1), there is one, definite, unambiguous output state.
- Whether it's a multitude of cascaded gates or a single gate, that output state is determined by the truth table(s) for the gate(s) in the circuit, and nothing else.

- However, altering this gate circuit so as to give signal feedback from the output to one of the inputs, strange things begin to happen:



A	Output
0	?
1	1



- If A is 1, the output *must* be 1 as well.
- Such is the nature of an OR gate: any "high" (1) input forces the output "high" (1).
- If A is "low" (0), however, the logic level or state of the output in our truth table cannot be guaranteed.

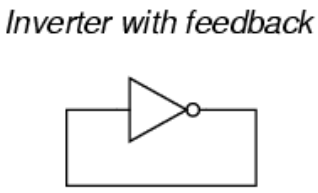
- Since the output feeds back to one of the OR gate's inputs, and knowing that any 1 input to an OR gates makes the output 1, this circuit will "latch" in the 1 output state after any time that A is 1.
- When A is 0, the output could be either 0 or 1, *depending on the circuit's prior state!*
- The proper way to complete the above truth table would be to insert the word *latch* in place of the question mark, showing that the output maintains its last state when A is 0.

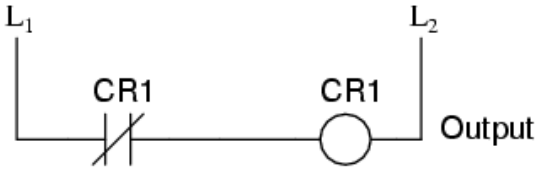
- Any digital circuit employing feedback is called a *multivibrator*.

- The example just explored with the OR gate was a very simple example of what is called a *bistable* multivibrator.
- It is called "bistable" because it can hold stable in one of *two* possible output states, either 0 or 1.

- There are also *monostable* multivibrators, which have only *one* stable output state (that other state being momentary),
- And, *astable* multivibrators, which have no stable state (oscillating back and forth between an output of 0 and 1).

- A very simple astable multivibrator is an inverter with the output fed directly back to the input:



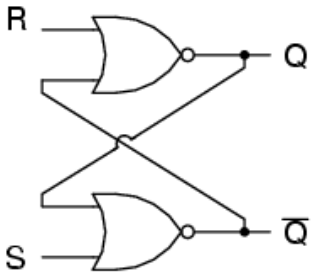


- When the input is 0, the output switches to 1.
- That 1 output gets fed back to the input as a 1.
- When the input is 1, the output switches to 0.
- That 0 output gets fed back to the input as a 0, and the cycle repeats itself.
- The result is a high frequency (several megahertz) oscillator, if implemented with a solid-state (semiconductor) inverter gate.

The S-R latch

- A bistable multivibrator has *two* stable states, as indicated by the prefix *bi* in its name.
- Typically, one state is referred to as *set* and the other as *reset*.
- The simplest bistable device, therefore, is known as a *set-reset*, or S-R, latch.

- To create an S-R latch, we can wire two NOR gates in such a way that the output of one feeds back to the input of another, and vice versa, like this:



S	R	Q	$\overline{Q}$
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	0	0

- The Q and not-Q outputs are supposed to be in opposite states.
- I say "supposed to" because making both the S and R inputs equal to 1 results in both Q and not-Q being 0.
- For this reason, having both S and R equal to 1 is called an *invalid* or *illegal* state for the S-R multivibrator.
- Otherwise, making S=1 and R=0 "sets" the multivibrator so that Q=1 and not-Q=0.
- Conversely, making R=1 and S=0 "resets" the multivibrator in the opposite state.
- When S and R are both equal to 0, the multivibrator's outputs "latch" in their prior states.

- By definition, a condition of Q=1 and not-Q=0 is *set*.
- A condition of Q=0 and not-Q=1 is *reset*.
- These terms are universal in describing the output states of any multivibrator circuit.

- The astute observer will note that the initial power-up condition of the gate S-R latch is such that both gates start in the de-energized mode.
- As such, one would expect that the circuit will start up in an invalid condition, with both Q and not-Q outputs being in the same state.
- Actually, this is true! However, the invalid condition is unstable with both S and R inputs inactive;
- as a result, the circuit will quickly stabilize in either the set or reset condition because one gate is bound to react a little faster than the other.
- If both gates were *precisely identical*, they would oscillate between high and low like an astable multivibrator upon power-up without ever reaching a point of stability!
- Fortunately for cases like this, such a precise match of components is a rare possibility.

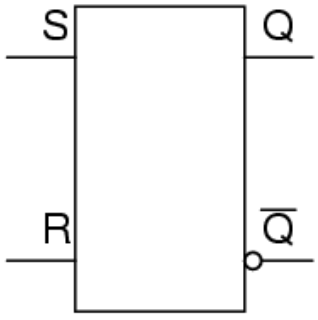
- It must be noted that although an astable (continually oscillating) condition would be extremely rare,
- there will most likely be a cycle or two of oscillation in the above circuit, and the final state of the circuit (set or reset) after power-up would be unpredictable.
- The root of the problem is a *race condition* between the two gates CR_1 and CR_2.

- A race condition occurs when two mutually-exclusive events are simultaneously initiated through different circuit elements by a single cause.
- Which gate "wins" this race is dependent on the physical characteristics of the gates and not the circuit design, so the designer cannot ensure which state the circuit will fall into after power-up.

- Race conditions should be avoided in circuit design primarily for the unpredictability that will be created.
- One way to avoid such a condition is to insert a time-delay relay into the circuit to disable one of the competing relays for a short time, giving the other one a clear advantage.

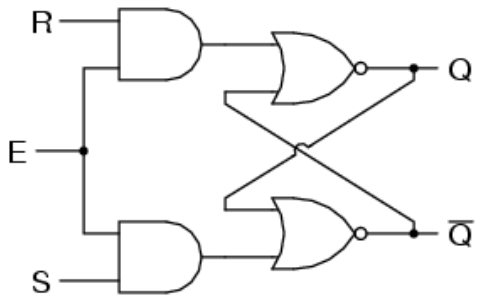
It should be mentioned that race conditions are not restricted to relay circuits. Solid-state logic gate circuits may also suffer from the ill effects of race conditions if improperly designed. Complex computer programs, for that matter, may also incur race problems if improperly designed. Race problems are a possibility for any sequential system, and may not be discovered until some time after initial testing of the system. They can be very difficult problems to detect and eliminate.

In semiconductor form, S-R latches come in prepackaged units so that you don't have to build them from individual gates. They are symbolized as such:



The gated S-R latch

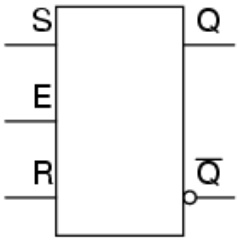
- It is sometimes useful in logic circuits to have a multivibrator which changes state only when certain conditions are met, regardless of its S and R input states.
- The conditional input is called the *enable*, and is symbolized by the letter E.
- See the following example to see how this works:



E	S	R	Q	$\overline{Q}$
0	**0**	**0**	latch	latch
0	**0**	1	latch	latch
0	1	**0**	latch	latch
0	1	1	latch	latch
1	**0**	**0**	latch	latch
1	**0**	1	**0**	1
1	1	**0**	1	**0**
1	1	1	**0**	**0**

- When the E=0, the outputs of the two AND gates are forced to 0, regardless of the states of either S or R.
- Consequently, the circuit behaves as though S and R were both 0, latching the Q and not-Q outputs in their last states.
- Only when the enable input is activated (1) will the latch respond to the S and R inputs.

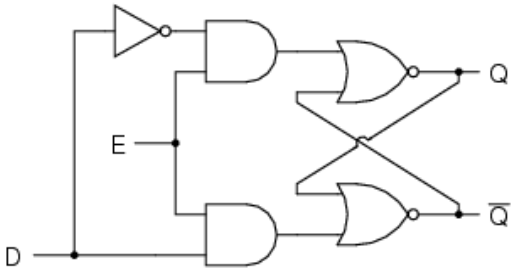
- Once again, these multivibrator circuits are available as prepackaged semiconductor devices, and are symbolized as such:



- It is also common to see the enable input designated by the letters "EN" instead of just "E."

The D latch

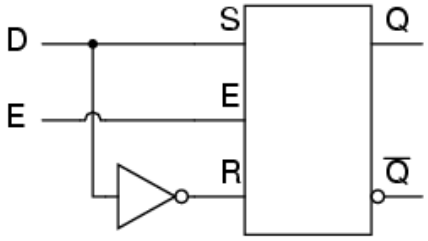
- Since the enable input on a gated S-R latch provides a way to latch the Q and not-Q outputs without regard to the status of S or R, we can eliminate one of those inputs to create a multivibrator latch circuit with no "illegal" input states.
- Such a circuit is called a D latch, and its internal logic looks like this:



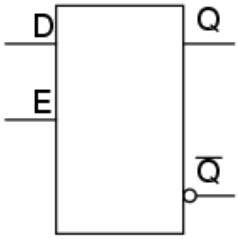
E	D	Q	$\overline{Q}$
0	**0**	latch	latch
0	1	latch	latch
1	**0**	**0**	1
1	1	1	**0**

- Note that the R input has been replaced with the complement (inversion) of the old S input, and the S input has been renamed to D.
- As with the gated S-R latch, the D latch will not respond to a signal input if the enable input is 0 -- it simply stays latched in its last state.
- When the enable input is 1, however, the Q output follows the D input.

- Since the R input of the S-R circuitry has been done away with, this latch has no "invalid" or "illegal" state.
- Q and not-Q are *always* opposite of one another.
- If the above diagram is confusing at all, the next diagram should make the concept simpler:



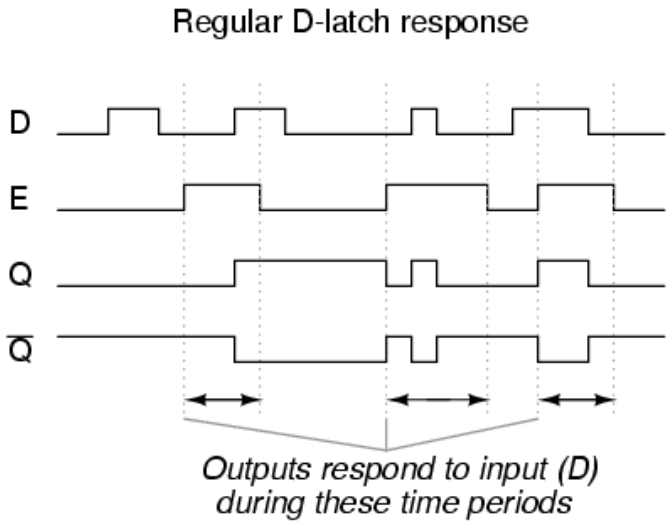
- Like both the S-R and gated S-R latches, the D latch circuit may be found as its own prepackaged circuit, complete with a standard symbol:

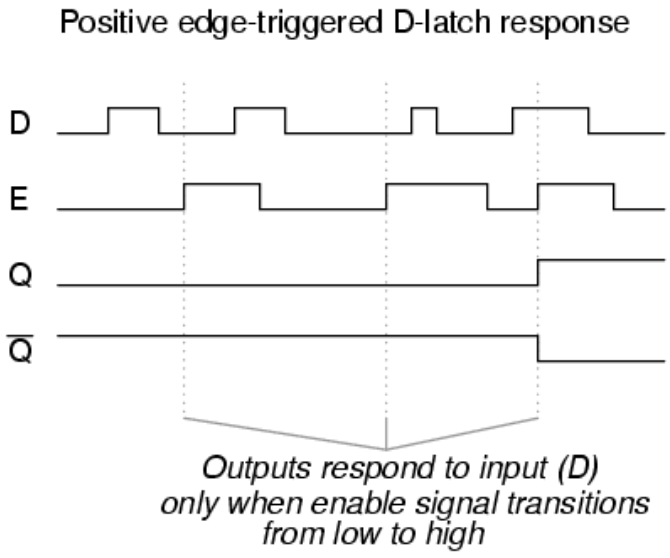


- The D latch is nothing more than a gated S-R latch with an inverter added to make R the complement (inverse) of S.
- An application for the D latch is a 1-bit memory circuit.
- You can "write" (store) a 0 or 1 bit in this latch circuit by making the enable input high (1) and setting D to whatever you want the stored bit to be.
- When the enable input is made low (0), the latch ignores the status of the D input and merrily holds the stored bit value, outputting at the stored value at Q, and its inverse on output not-Q.

Edge-triggered latches: Flip-Flops

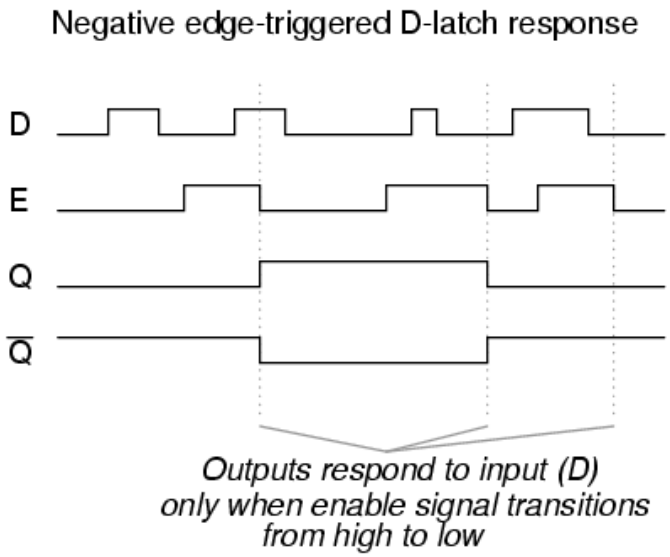
- In many digital applications it is desirable to limit the responsiveness of a latch circuit to a very short period of time instead of the entire duration that the enabling input is activated. - One method of enabling a multivibrator circuit is called *edge triggering*, where the circuit's data inputs have control only during the time that the enable input is *transitioning* from one state to another.
- Let's compare timing diagrams for a normal D latch versus one that is edge-triggered:





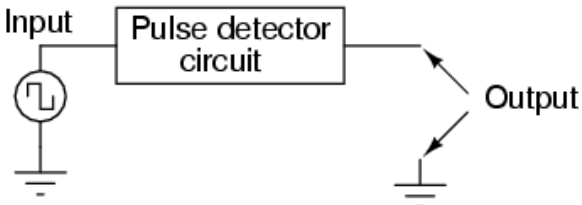
- In the first timing diagram, the outputs respond to input D whenever the enable (E) input is high, for however long it remains high.
- When the enable signal falls back to a low state, the circuit remains latched.
- In the second timing diagram, we note a distinctly different response in the circuit output(s):
- it only responds to the D input during that brief moment of time when the enable signal *changes*, or *transitions*, from low to high.
- This is known as *positive* edge-triggering.

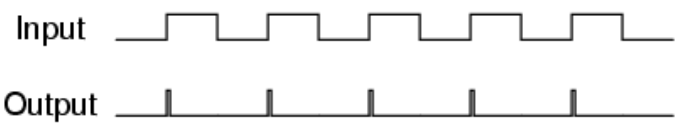
- There is such a thing as *negative* edge triggering as well, and it produces the following response to the same input signals:



- Whenever a multivibrator circuit is enabled on the transitional edge of a square-wave enable signal, it is called a *flip-flop* instead of a *latch*.
- Consequently, and edge-triggered S-R circuit is more properly known as an S-R flip-flop, and an edge-triggered D circuit as a D flip-flop.
- The enable signal is renamed to be the *clock* signal.
- Also, the data inputs (S, R, and D, respectively) of these flip-flops are referred to as *synchronous* input
- (because they have effect only at the time of the clock pulse edge (transition), thereby synchronizing any output changes with that clock pulse, rather than at the whim of the data inputs.)

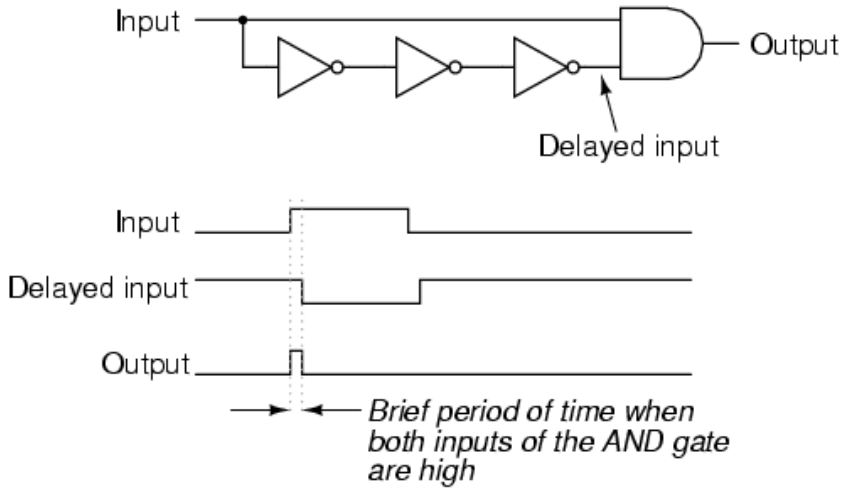
- But, how is this edge-triggering actually accomplished?
- To create a "gated" S-R latch from a regular S-R latch is easy enough with a couple of AND gates, but, to implement logic that only pays attention to the *rising or falling edge* of a changing digital signal needs something more:
- What is needed is a digital circuit that outputs a brief pulse whenever the input is activated for an arbitrary period of time
- the output of this circuitcan be used to briefly enable the latch.
- this is technically a kind of monostable multivibrator, which for now we'll call a *pulse detector*:



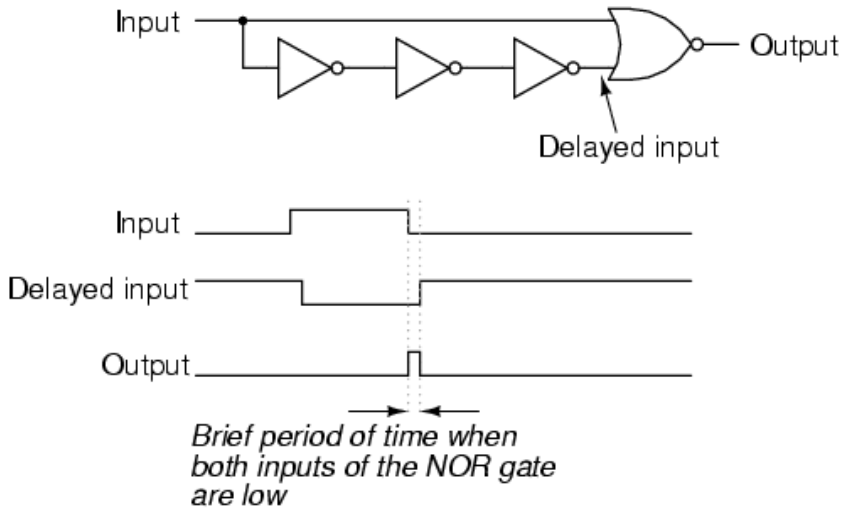


- The duration of each output pulse is set by components in the pulse circuit itself. This can be accomplished quite easily through the use of a time-delay relay with a very short delay time:

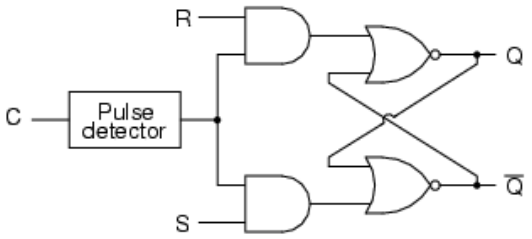
- Implementing this timing function with semiconductor components is actually quite easy, as it exploits the inherent time delay within every logic gate (known as *propagation delay*).
- An input signal is split up two ways, then a gate or a series of gates placed in one of those signal paths just to delay it a bit.
- Then both the original signal and its delayed counterpart enter into a two-input gate that outputs a high signal for the brief moment of time that the delayed signal has not yet caught up to the low-to-high change in the non-delayed signal.
- An example circuit for producing a clock pulse on a low-to-high input signal transition is shown here:



- This circuit may be converted into a negative-edge pulse detector circuit with only a change of the final gate from AND to NOR:



- Now that we know how a pulse detector can be made, we can show it attached to the enable input of a latch to turn it into a flip-flop.
- In this case, the circuit is a S-R flip-flop:



C	S	R	Q	$\overline{Q}$
⌐	0	0	latch	latch
⌐	0	1	0	1
⌐	1	0	1	0
⌐	1	1	0	0
x	0	0	latch	latch
x	0	1	latch	latch
x	1	0	latch	latch
x	1	1	latch	latch

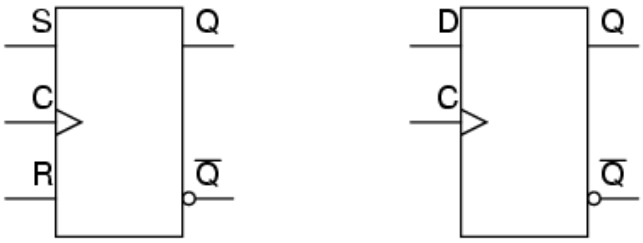
– Only when the clock signal (C) is transitioning from low to high is the circuit responsive to the S and R inputs.

- For any other condition of the clock signal ("x") the circuit will be latched.

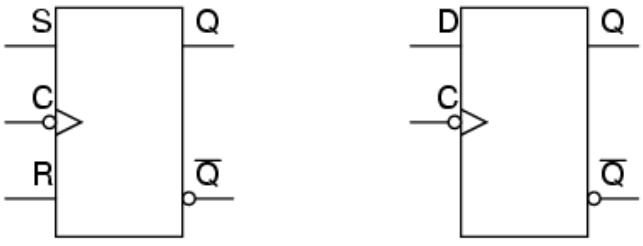
- It is important to note that the invalid state for the S-R flip-flop is maintained only for the short period of time that the pulse detector circuit allows the latch to be enabled.
- After that brief time period has elapsed, the outputs will latch into either the set or the reset state.

- Once again, the problem of a *race condition* manifests itself.
- With no enable signal, an invalid output state cannot be maintained.
- However, the valid "latched" states of the multivibrator -- set and reset -- are mutually exclusive to one another;
- Therefore, the two gates of the multivibrator circuit will "race" each other for supremacy, and whichever one attains a high output state first will "win."

- The block symbols for flip-flops are slightly different from that of their respective latch counterparts:

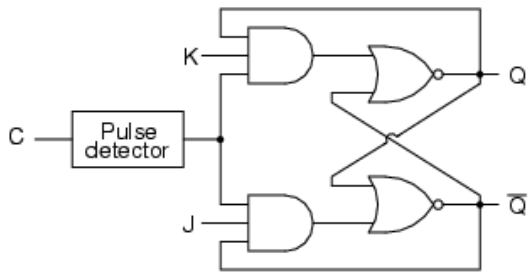


- The triangle symbol next to the clock inputs tells us that these are edge-triggered devices, and consequently that these are flip-flops rather than latches.
- The symbols above are positive edge-triggered: that is, they "clock" on the rising edge (low-to-high transition) of the clock signal.
- Negative edge-triggered devices are symbolized with a bubble on the clock input line:



- Both of the above flip-flops will "clock" on the falling edge (high-to-low transition) of the clock signal.

- Another variation on a theme of bistable multivibrators is the J-K flip-flop.
- Essentially, this is a modified version of an S-R flip-flop with no "invalid" or "illegal" output state.
- Look closely at the following diagram to see how this is accomplished:



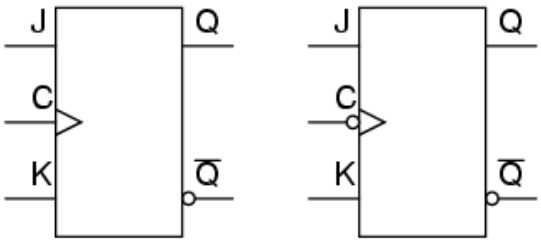
C	J	K	Q	$\overline{Q}$
⎍	0	0	latch	latch
⎍	0	1	0	1
⎍	1	0	1	0
⎍	1	1	toggle	toggle
x	0	0	latch	latch
x	0	1	latch	latch
x	1	0	latch	latch
x	1	1	latch	latch

- What used to be the S and R inputs are now called the J and K inputs, respectively.
- The old two-input AND gates have been replaced with 3-input AND gates, and the third input of each gate receives feedback from the Q and not-Q outputs.
- What this does for us is permit the J input to have effect only when the circuit is reset, and permit the K input to have effect only when the circuit is set.
- In other words, the two inputs are *interlocked*, to use a relay logic term, so that they cannot both be activated simultaneously.
- If the circuit is "set," the J input is inhibited by the 0 status of not-Q through the lower AND gate; if the circuit is "reset," the K input is inhibited by the 0 status of Q through the upper AND gate.

- When both J and K inputs are 1, however, something unique happens.
- Because of the selective inhibiting action of those 3-input AND gates, a "set" state inhibits input J so that the flip-flop acts as if J=0 while K=1 when in fact both are 1.
- On the next clock pulse, the outputs will switch ("toggle") from set (Q=1 and not-Q=0) to reset (Q=0 and not-Q=1).
- Conversely, a "reset" state inhibits input K so that the flip-flop acts as if J=1 and K=0 when in fact both are 1.
- The next clock pulse toggles the circuit again from reset to set.

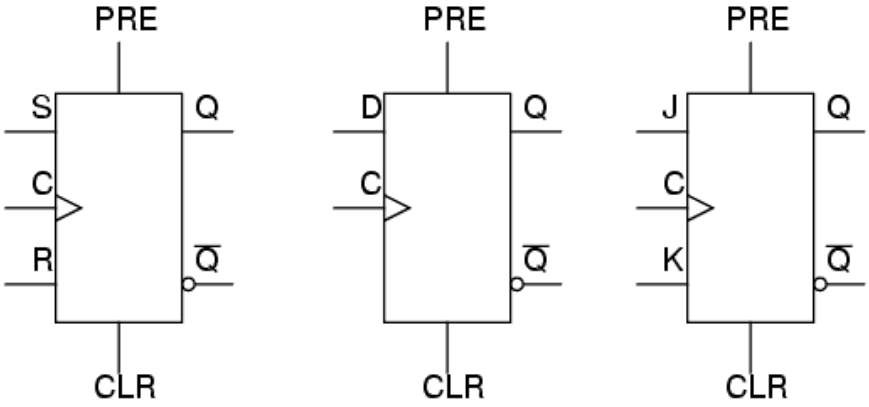
- The end result is that the S-R flip-flop's "invalid" state is eliminated (along with the race condition it engendered) giving a useful feature as a bonus: the ability to toggle between the two (bistable) output states with every transition of the clock input signal.
- There is no such thing as a J-K latch, only J-K flip-flops.
- Without the edge-triggering of the clock input, the circuit would continuously toggle between its two output states when both J and K were held high (1), making it an astable device instead of a bistable device in that circumstance.
- If we want to preserve bistable operation for all combinations of input states, we *must* use edge-triggering so that it toggles only when we tell it to, one step (clock pulse) at a time.

- The block symbol for a J-K flip-flop is a whole lot less frightening than its internal circuitry, and just like the S-R and D flip-flops, J-K flip-flops come in two clock varieties (negative and positive edge-triggered):



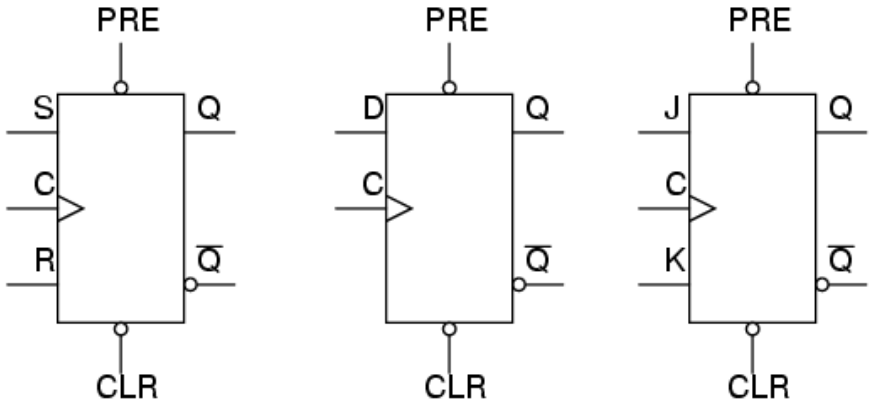
Asynchronous flip-flop inputs

- The normal data inputs to a flip flop (D, S and R, or J and K) are referred to as *synchronous* inputs because they have effect on the outputs (Q and not-Q) only in step, or in sync, with the clock signal transitions.
- These extra inputs are called *synchronous* because they can set or reset the flip-flop regardless of the status of the clock signal.
- Typically, they're called *preset* and *clear*:

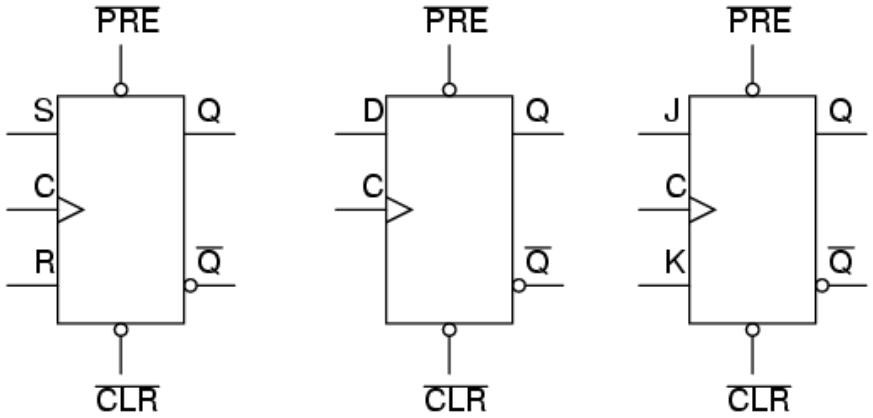


- When the preset input is activated, the flip-flop will be set (Q=1, not-Q=0) regardless of any of the synchronous inputs or the clock.
- When the clear input is activated, the flip-flop will be reset (Q=0, not-Q=1), regardless of any of the synchronous inputs or the clock.
- So, what happens if both preset and clear inputs are activated? Surprise, surprise: we get an invalid state on the output, where Q and not-Q go to the same state, the same as our old friend, the S-R latch!
- Preset and clear inputs find use when multiple flip-flops are ganged together to perform a function on a multi-bit binary word, and a single line is needed to set or reset them all at once.

- Asynchronous inputs, just like synchronous inputs, can be engineered to be active-high or active-low.
- If they're active-low, there will be an inverting bubble at that input lead on the block symbol, just like the negative edge-trigger clock inputs.



- Sometimes the designations "PRE" and "CLR" will be shown with inversion bars above them, to further denote the negative logic of these inputs:



To sum up:

- **Asynchronous** inputs on a flip-flop have control over the outputs (Q and not-Q) regardless of clock input status.

- These inputs are called the **preset (PRE)** and **clear (CLR)**. The preset input drives the flip-flop to a set state while the clear input drives it to a reset state.

- **It is possible** to drive the outputs of a **J-K flip-flop** to an **invalid condition** using the asynchronous inputs. As a result, all feedback within the multivibrator circuit is overridden.

Monostable multivibrators

one example of a monostable multivibrator has already been seen in use:
i.e. the pulse detector used within the circuitry of flip-flops, to enable the latch portion for a brief time when the clock input signal transitions from either low to high or high to low.

- The pulse detector is classified as a monostable multivibrator because it has only **one** stable state.

- By **stable**, this means a state of output where the device is able to latch or hold to forever, without external prodding.

- A latch or flip-flop, being a bistable device, can hold in either the "set" or "reset" state for an indefinite period of time.
- Once its set or reset, it will continue to latch in that state unless prompted to change by an external input.

- A **monostable device**, on the other hand, **is only able to hold in one particular state indefinitely**.
- Its other state can only be held momentarily when triggered by an external input.

- A mechanical analogy of a monostable device would be a momentary contact pushbutton switch, which spring-returns to its normal (stable) position when pressure is removed from its button actuator.
- Likewise, a standard wall (toggle) switch, such as the type used to turn lights on and off in a house, is a bistable device. It can latch in one of two modes: on or off.

- **All monostable multivibrators are timed devices**. That is, their unstable output state will hold only for a certain minimum amount of time before returning to its stable state.
- With semiconductor monostable circuits, this timing function is typically accomplished through the use of resistors and capacitors, making use of the exponential charging rates of RC circuits.
- A comparator is often used to compare the voltage across the charging (or discharging) capacitor with a steady reference voltage, and the on/off output of the comparator used for a logic signal.

Summing Up:

- A **monostable** multivibrator has only one stable output state. The other output state can only be maintained temporarily.

- Monostable multivibrators, sometimes called **one-shots**, come in two basic varieties: **retriggerable** and ***nonretriggerable***.

- One-shot circuits with very short time settings may be used to 'debounce' the 'dirty' signals created by mechanical switch contacts.

*** SECTION INCOMPLETE ***

SEQUENTIAL CIRCUITS

*** SECTION INCOMPLETE ***

Binary count sequence

examinng a four-bit binary count sequence from 0000 to 1111, a definite pattern will be evident in the "oscillations" of the bits between 0 and 1:

```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

- Note how **the least significant bit (LSB) toggles between 0 and 1 for every step** in the count sequence,
- while **each succeeding bit toggles at one-half the frequency of the one before it**.
- **The most significant bit (MSB) only toggles once** during the entire sixteen-step count sequence: **at the transition between 7** (0111) **and 8** (1000).

- **to design a digital circuit to "count"** in four-bit binary, one would have to design a series of frequency divider circuits, each circuit dividing the frequency of a square-wave pulse by a factor of 2:

- **J-K flip-flops are ideally suited for this task**, because they have the ability to **'toggle' their output state at the command of a clock pulse** when both J and K inputs are made 'high':

- If we consider the two signals (A and B) in this circuit to represent two bits of a binary number, signal A being the LSB and signal B being the MSB, we see that the count sequence is backward: from 11 to 10 to 01 to 00 and back again to 11. Although it might not be counting in the direction we might have assumed, at least it counts!

The following sections explore different types of counter circuits, all made with J-K flip-flops, and all based on the exploitation of that flip-flop's toggle mode of operation.

To Sum Up:

- **Binary count sequences follow a pattern of octave frequency division**:
- **the frequency of oscillation for each bit**, from LSB to MSB, **follows a divide-by-two pattern**.
- In other words, **the LSB will oscillate at the highest frequency**, followed by **the next bit at one-half the LSB's frequency**, and **the next bit at one-half the frequency of the bit before it**, **etc**.

- **Circuits may be built that "count"** in a binary sequence, **using J-K flip-flops set up in the "toggle" mode**.

ASYNCHROUS COUNTERS

In the previous section was a circuit using one J-K flip-flop that counted backward in a two-bit binary sequence, from 11 to 10 to 01 to 00.

Since it would be desirable to have a circuit that could count *forward* and not just backward, it would be worthwhile to examine a forward count sequence again and look for more patterns that might indicate how to build such a circuit.

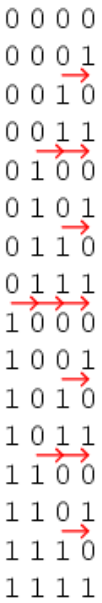
- it's know that **binary count sequences follow a pattern of octave (factor of 2) frequency division**, and,
- **J-K flip-flop multivibrators set up for the "toggle" mode are capable of performing this type of frequency division**,
- so, **one can envision a circuit made up of several J-K flip-flops, cascaded** to produce four bits of output.

- The main problem is to determine *how* to connect these flip-flops together so that they toggle at the right times to produce the proper binary sequence.
- Examinng the following binary count sequence, paying attention to patterns preceding the "toggling" of a bit between 0 and 1:

0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1

- it should be noted Note that **each bit in this four-bit sequence toggles when the bit before it** (the bit having a lesser significance, or place-weight), **toggles** in a particular direction: from 1 to 0.

- Small arrows indicate those points in the sequence where a bit toggles, the head of the arrow pointing to the previous bit transitioning from a "high" (1) state to a "low" (0) state:



- **Starting with four J-K flip-flops** connected in such a way to always be **in the "toggle" mode;**

- **it's necessary** to determine how **to connect the clock inputs** in such a way **so that each succeeding bit toggles when the bit before** it **transitions** from 1 to 0.

- **The Q outputs** of each flip-flop will serve as the respective binary bits of the final, four-bit count:

- **If** flip-flops with **negative-edge triggering** (bubble symbols on the clock inputs) **were used**, one could simply connect the clock input of each flip-flop to the Q output of the flip-flop before it,
- allowing for, as the bit before it to change from a 1 to a 0, the "falling edge" of that signal would "clock" the next flip-flop to toggle the next bit:

- This circuit would yield the following output waveforms, when "clocked" by a repetitive source of pulses from an oscillator:

- **The first flip-flop** (the one with the Q_0 output), **has a positive-edge triggered clock input, so it toggles with each rising edge** of the clock signal.

- Note how **the clock signal** in this example **has a duty cycle less than 50%**.

- the signal is shown in this manner for the purpose of demonstrating how the clock signal need not be symmetrical to obtain reliable, "clean" output bits in our four-bit binary sequence.

- In the very first flip-flop circuit shown in this chapter, **the clock signal itself was used as one of the output bits.**
- This **is a bad practice in counter design**, though, **because it necessitate**s the use of a square wave signal with a **50% duty cycle ("high" time = "low" time)** to obtain a count sequence where each and every step pauses for the same amount of time.

- Using **one J-K flip-flop for each output bit**, however, **avoids the necessity of having a symmetrical clock signal**, allowing the use of practically any variety of high/low waveform to increment the count sequence.

- As indicated by all the other arrows in the pulse diagram, each succeeding output bit is toggled by the action of the preceding bit transitioning from "high" (1) to "low" (0). This is the pattern necessary to generate an "up" count sequence.

- **A less obvious solution** for generating an "up" sequence using positive-edge triggered flip-flops **is to "clock" each flip-flop using the Q' output** of the preceding flip-flop rather than the Q output.

- **Since the Q' output will always be the exact opposite state of the Q output on a J-K flip-flop (no invalid states with this type of flip-flop)**, a high-to-low transition on the Q output will be accompanied by a low-to-high transition on the Q' output.

- In other words, **each time the Q output of a flip-flop transitions from 1 to 0, the Q' output of the same flip-flop will transition from 0 to 1**, providing the positive-going clock pulse we would need to toggle a positive-edge triggered flip-flop at the right moment:

- One way to expand the capabilities of either of these two counter circuits is to regard the Q' outputs as another set of four binary bits.

- examining the pulse diagram for such a circuit, it can be seen that **the Q' outputs generate a DOWN-counting sequence**, while **the Q outputs generate an UP-counting sequence**:

- Unfortunately, **all of the counter circuits shown thusfar share a common problem: 'the *ripple* effect'**.

- This effect is **seen in certain types of binary adder and data conversion circuits**, and is due to accumulative propagation delays between cascaded gates.

- **When the Q output** of a flip-flop **transitions from 1 to 0**, **it commands the next flip-flop to toggle.**

- If the next flip-flop toggle is a transition from 1 to 0, it will command the flip-flop after it to toggle as well, and so on.

- However, since **there is always some small amount of propagation delay between** the command to toggle (**the clock pulse**) **and** the actual toggle response (**Q and Q' outputs changing states**), any **subsequent flip-flops to be toggled will toggle some time *after* the first flip-flop has toggled**.

- Thus, **when multiple bits toggle in a binary count sequence, they will not all toggle at exactly the same time**:

- it can be seen, **the more bits that toggle** with a given clock pulse, **the more severe the accumulated delay time from LSB to MSB**.

- **When a clock pulse occurs at such a transition point** (say, on the transition from 0111 to 1000), **the output bits will "ripple" in sequence** from LSB to MSB, as each succeeding bit toggles and commands the next bit to toggle as well, with a small amount of propagation delay between each bit toggle.

- taking a close-up look at this effect **during the transition from 0111 to 1000**, it can be seen that **there will be *false* output counts generated** in the brief time period that the "ripple" effect takes place:

- **Instead of cleanly transitioning** from a "0111" output to a "1000" output, **the counter circuit will very quickly ripple from** 0111 to 0110 to 0100 to 0000 to 1000, or **from 7 to *6* to *4* to *0* and then to 8**.

- This behavior earns the counter circuit the name of 'ripple counter' or 'asynchronous counter'.

- In many applications, this effect is tolerable, since the ripple happens very, very quickly (the width of the delays has been exaggerated here as an aid to understanding the effects).

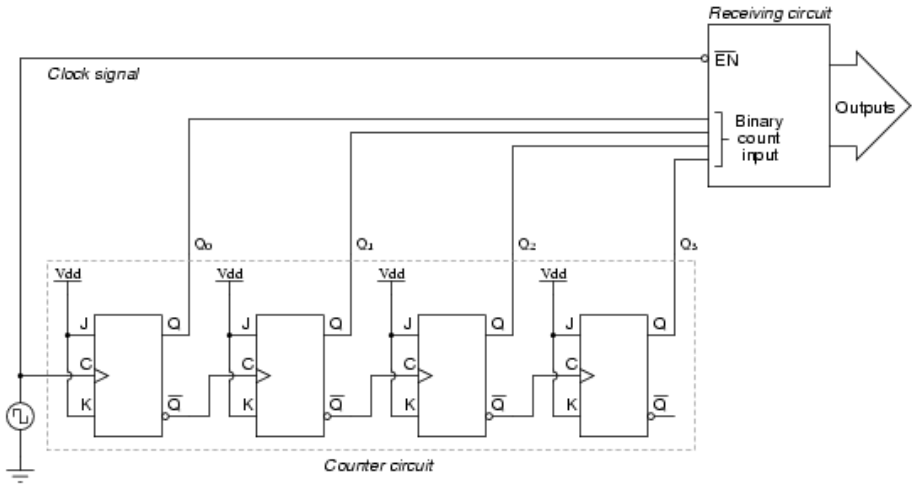
- If it was just desired to drive a set of light-emitting diodes (LEDs) with the counter's outputs, for example, this brief ripple would be of no consequence at all.
- However, to use this counter to drive the "select" inputs of a multiplexer, index a memory pointer in a microprocessor (computer) circuit, or perform some other task where false outputs could cause spurious errors, it would not be acceptable.
- There is a way to use this type of counter circuit in applications sensitive to false, ripple-generated outputs, and it involves a principle known as 'strobing'.

- Most decoder and multiplexer circuits are equipped with at least one input called the 'enable'.

- The output(s) of such a circuit will be active only when the enable input is made active.

- this enable input can be used to 'strobe' the circuit receiving the ripple counter's output so that it is disabled (and thus not responding to the counter output) during the brief period of time in which the counter outputs might be rippling, and,
- enabled only when sufficient time has passed since the last clock pulse that all rippling will have ceased.

- In most cases, the strobing signal can be the same clock pulse that drives the counter circuit:



- With an active-low Enable input, the receiving circuit will respond to the binary count of the four-bit counter circuit only when the clock signal is 'low'.
- As soon as the clock pulse goes "high," the receiving circuit stops responding to the counter circuit's output.

- Since the counter circuit is positive-edge triggered (as determined by the 'first' flip-flop clock input), all the counting action takes place on the low-to-high transition of the clock signal.
- this means that the receiving circuit will become disabled just before any toggling occurs on the counter circuit's four output bits.

- The receiving circuit will not become enabled until the clock signal returns to a low state, which should be a long enough time *after* all rippling has ceased to be "safe" to allow the new count to have effect on the receiving circuit.

- The crucial parameter here is the clock signal's "high" time: it must be at least as long as the maximum expected ripple period of the counter circuit.
- If not, the clock signal will prematurely enable the receiving circuit, while some rippling is still taking place.

- Another disadvantage of the asynchronous, or ripple, counter circuit is limited speed.

- While all gate circuits are limited in terms of maximum signal frequency, the design of asynchronous counter circuits compounds this problem by making propagation delays additive.

- Thus, even if strobing is used in the receiving circuit, an asynchronous counter circuit cannot be clocked at any frequency higher than that which allows the greatest possible accumulated propagation delay to elapse well before the next pulse.

- The solution to this problem is a counter circuit that avoids ripple altogether.
- Such a counter circuit would eliminate the need to design a "strobing" feature into whatever digital circuits use the counter output as an input, and would also enjoy a much greater operating speed than its asynchronous equivalent.

- This design of counter circuit is the subject of the next section.

To Sum Up:

- An **up-counter may be made by connecting the clock inputs of positive-edge triggered J-K flip-flops to the Q' outputs of the preceding flip-flops.**
- Another way is to use **negative-edge triggered flip-flops, connecting the clock inputs to the Q outputs of the preceding flip-flops.**
- In either case, **the J and K inputs of all flip-flops are connected to V_{cc} or V_{dd} so as to always be 'high'.**

- **Counter circuits made from cascaded J-K flip-flops where** each clock input receives its **pulses from the** output of the **previous flip-flop invariably exhibit a 'ripple effect'**,
- (**false output counts are generated** between some steps of the count sequence).
- These types of counter circuits are called - '**asynchronous counters**', or '**ripple counters**'.

- **Strobing** is a technique **applied to circuits receiving the output of an asynchronous (ripple) counter**, so that the false counts generated during the ripple time **will have no ill effect**.
- Essentially, the '**enable**' **input** of such a circuit **is connected to the counter's clock pulse** in **such** a way **that it is enabled only when the counter outputs are not changing**,
- this **will be disabled when** the periods of changing **counter output ripples occur**.

Synchronous counters

- A '**synchronous counter**', in contrast to an '**asynchronous counter**', is one whose output bits change state simultaneously, **with no ripple**.

- T**he only way to build** such a counter circuit **from J-K flip-flops is to connect all the clock inputs together**, so that each and every flip-flop receives the exact same clock pulse at the exact same time:

- Now, the question is, what to do with the J and K inputs?

- it's known that we still have to maintain the same divide-by-two frequency pattern in order to count in a binary sequence, and that this pattern is best achieved utilizing the "toggle" mode of the flip-flop, so the fact that the J and K inputs must both be (at times) "high" is clear.

- However, if we simply connect all the J and K inputs to the positive rail of the power supply as we did in the asynchronous circuit, this would clearly not work because all the flip-flops would toggle at the same time: with each and every clock pulse!

- examining the four-bit binary counting sequence again, are there are any other patterns that predict the toggling of a bit?.

- Asynchronous counter circuit design is based on the fact that each bit toggle happens at the same time that the preceding bit toggles from a "high" to a "low" (from 1 to 0).

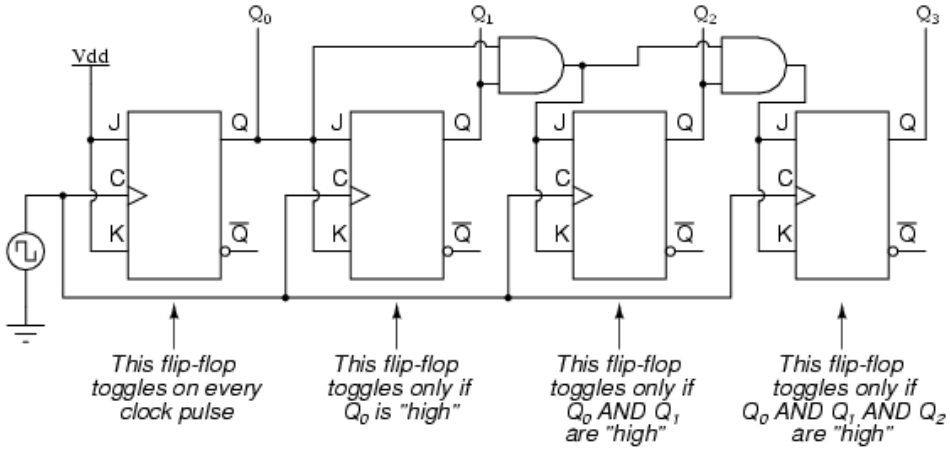
- Since the toggling of a bit cannot be clocked based on the toggling of a previous bit in a synchronous counter circuit (to do so would create a ripple effect) some other pattern must be found in the counting sequence that can be used to trigger a bit toggle:

- Examining the four-bit binary count sequence, another predictive pattern can be seen. Notice that just before a bit toggles, all preceding bits are "high:"

- This pattern is also something exploitable in designing a counter circuit.

- enabling each J-K flip-flop to toggle based on whether or not all preceding flip-flop outputs (Q) are "high," its possible to obtain the same counting sequence as the asynchronous circuit without the ripple effect, since each flip-flop in this circuit will be clocked at exactly the same time:

A four-bit synchronous "up" counter



- The result is a four-bit **'synchronous' UP-counter**.

- **Each** of the **higher-order flip-flops** are made **ready to toggle** (both J and K inputs "high") **if** the **Q outputs of all previous flip-flops are 'high'**

- **Otherwise**, the J and K **inputs** for that flip-flop **will both be 'low', placing it into the 'latch' mode** where it will maintain its present output state at the next clock pulse.

- Since the first (LSB) flip-flop needs to toggle at every clock pulse, its J and K inputs are connected to V_{cc} or V_{dd}, where they will be "high" all the time.

- The next flip-flop need only "recognize" that the first flip-flop's Q output is high to be made ready to toggle, so no AND gate is needed.

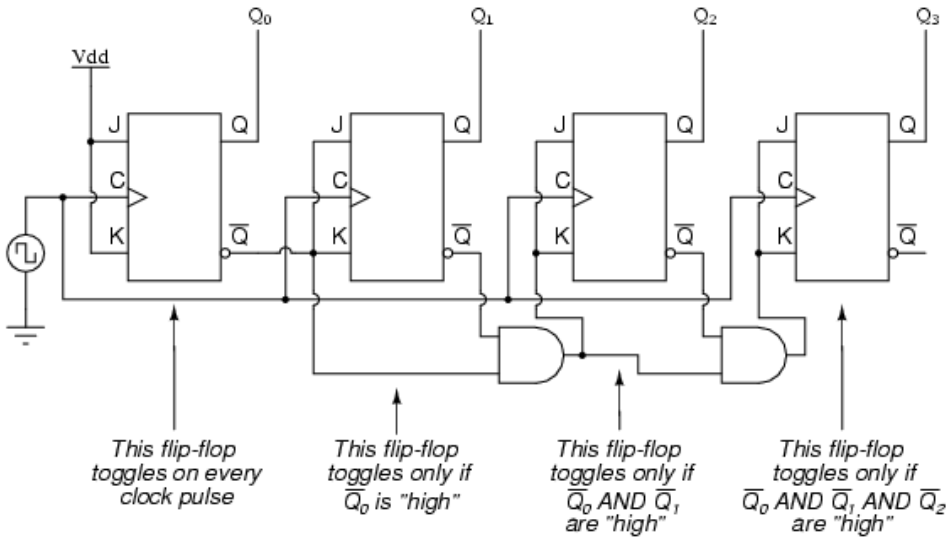
- However, the remaining flip-flops should be made ready to toggle only when *all* lower-order output bits are "high," thus the need for AND gates.

- To make a synchronous "down" counter, it's necessary to build the circuit to recognize the appropriate bit patterns predicting each toggle state while counting down.

- Not surprisingly, when the four-bit binary count sequence is examined, its seen that all preceding bits are "low" prior to a toggle (following the sequence from bottom to top):

- Since each J-K flip-flop comes equipped with a Q' output as well as a Q output, the Q' outputs can be used to enable the toggle mode on each succeeding flip-flop, since each Q' will be "high" every time that the respective Q is "low:"

A four-bit synchronous "down" counter



- Taking this idea one step further, a counter circuit with selectable between "up" and "down" count modes can be built by having dual lines of AND gates detecting the appropriate bit conditions for an "up" and a "down" counting sequence, respectively, then use OR gates to combine the AND gate outputs to the J and K inputs of each succeeding flip-flop:

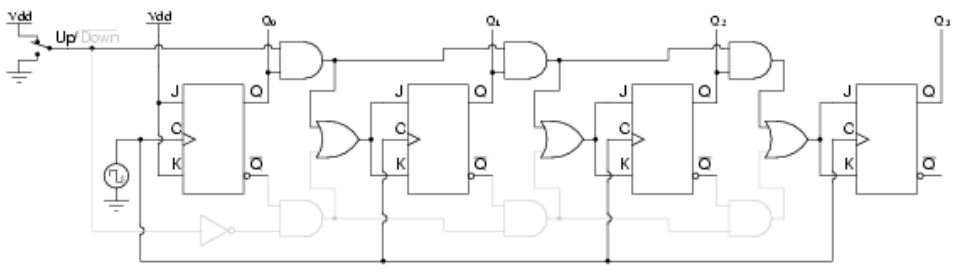
- This circuit isn't as complex as it might first appear;

- The Up/Down control input line simply enables either the upper string or lower string of AND gates to pass the Q/Q' outputs to the succeeding stages of flip-flops.

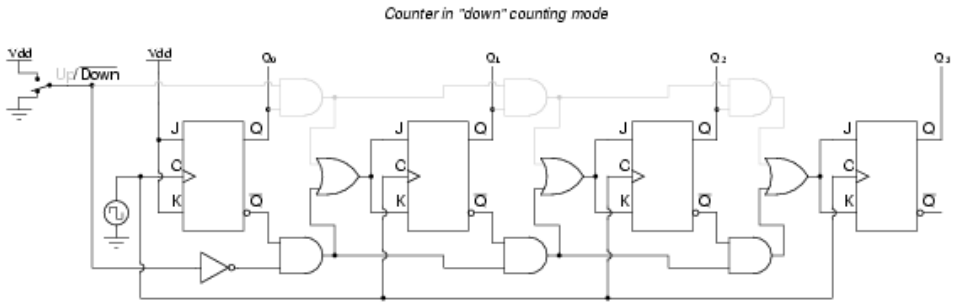
- If the Up/Down control line is 'high', the top AND gates become enabled, and the circuit functions exactly the same as the first ('up') synchronous counter circuit shown in this section.
- If the Up/Down control line is made 'low', the bottom AND gates become enabled, and the circuit functions identically to the second (DOWN-counter) circuit shown in this section.

- To illustrate, here is a diagram showing the circuit in the UP-counting mode (all disabled circuitry shown in grey rather than black):

Counter in "up" counting mode

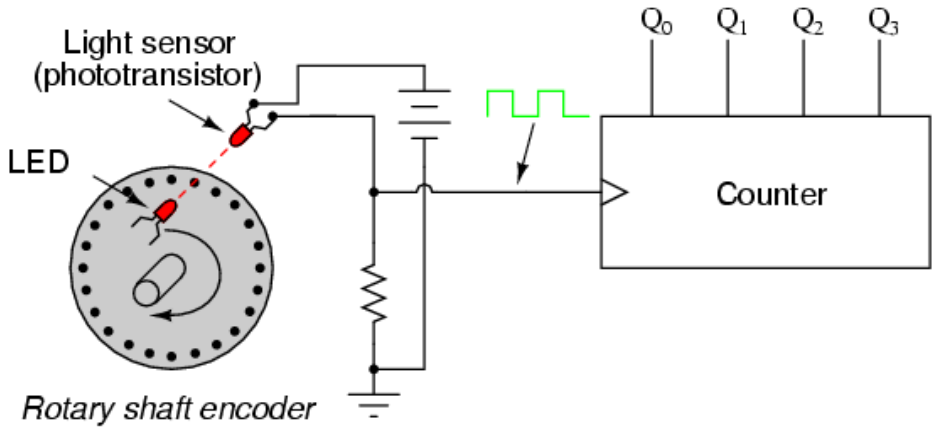


- Here, shown in the "down" counting mode, with the same grey coloring representing disabled circuitry:



- Up/down counter circuits are very useful devices.

- A common application is in machine motion control, where devices called '*rotary shaft encoders*' convert mechanical rotation into a series of electrical pulses, these pulses "clocking" a counter circuit to track total motion:



- As the machine moves, it turns the encoder shaft, making and breaking the light beam between LED and phototransistor, thereby generating clock pulses to increment the counter circuit.

- Thus, the counter integrates, or accumulates, total motion of the shaft, serving as an electronic indication of how far the machine has moved.

- If the only care is tracking total motion, and not to account for changes in the *direction* of motion, this arrangement will suffice.

- However, for the counter to *increment* with one direction of motion and *decrement* with the reverse direction of motion, an up/down counter must be used, and an encoder/decoding circuit having the ability to discriminate between different directions.

- re-designing the encoder to have two sets of LED/phototransistor pairs, those pairs aligned such that their square-wave output signals are 90^{o} out of phase with each other, the result is what is known as a 'quadrature output' encoder (the word "quadrature" simply refers to a 90^{o} angular separation).

- A phase detection circuit may be made from a D-type flip-flop, to distinguish a clockwise pulse sequence from a counter-clockwise pulse sequence:

- When the encoder rotates clockwise, the 'D' input signal square-wave will lead the 'C' input square-wave, meaning that the 'D' input will already be 'high' when the 'C' transitions from 'low' to 'high,' thus 'setting' the D-type flip-flop (making the Q output 'high') with every clock pulse.

- A 'high' Q output places the counter into the 'Up' count mode, and any clock pulses received by the clock from the encoder (from either LED) will increment it.

- Conversely, when the encoder reverses rotation, the 'D' input will lag behind the 'C' input waveform, meaning that it will be "low" when the "C" waveform transitions from 'low' to 'high', forcing the D-type flip-flop into the *reset* state (making the Q output 'low') with every clock pulse.
- This "low" signal commands the counter circuit to decrement with every clock pulse from the encoder.

- This circuit, or something very much like it, is at the heart of every position-measuring circuit based on a pulse encoder sensor.

- Such applications are very common in robotics, CNC machine tool control, and other applications involving the measurement of reversible, mechanical motion, including the encoders on you synth or groovebox.

INCOMLETE

COUNTER MODULUS

INCOMLETE

Finite State Machines

- Up to now, every circuit that was presented was a 'combinatorial' circuit.

- That means that its output is dependent only by its current inputs.

- i.e. Previous inputs for that type of circuits have no effect on the output.

- However, there are many applications where there is a need for circuits to have 'memory';

- to remember previous inputs and calculate their outputs according to them.
- a circuit whose output depends not only on the present input but also on the history of the input is called a 'Sequential Circuit'.

- this section covers how to design and build such sequential circuits.

- In order to see how this procedure works, using an example to study topic:

So let's suppose a digital quiz game that works on a clock and reads an input from a manual button.

- However, desiring the switch to transmit only one HIGH pulse to the circuit.

- If the button is hooked directly on the game circuit it will transmit HIGH for as few clock cycles as our finger can achieve.

- On a common clock frequency a finger can never be fast enough.

- The designing procedure has specific steps that must be followed in order to get the work done:

Step 1

- The first step of the design procedure is to define with simple but clear words what the circuit is to do:

"The mission is to design a secondary circuit that will transmit a HIGH pulse with duration of only one cycle when the manual button is pressed, and won't transmit another pulse until the button is depressed and pressed again."

Step 2

- The next step is to design a State Diagram. This is a diagram that is made from circles and arrows and describes visually the operation of our circuit. In mathematic terms, this diagram that describes the operation of the sequential circuit is a Finite State Machine.

- note that this is a Moore Finite State Machine. Its output is a function of only its current state, not its input. That is in contrast with the Mealy Finite State Machine, where input affects the output. In this tutorial, only the Moore Finite State Machine will be examined.

- The State Diagram of the circuit is the following: (Figure below)

A State Diagram

- Every circle represents a "state", a well-defined condition that the machine can be found at.

- In the upper half of the circle that condition is described. The description helps us remember what our circuit is supposed to do at that condition.

    - The first circle is the "stand-by" condition. This is where the circuit starts from and where it waits for another button press.
    - The second circle is the condition where the button has just been just pressed and the circuit needs to transmit a HIGH pulse.
    - The third circle is the condition where the circuit waits for the button to be released before it returns to the "stand-by" condition.

- In the lower part of the circle is the output of our circuit. If the circuit is to transmit a HIGH on a specific state, a 1 is put on that state. Otherwise a 0 is put.

- Every arrow represents a "transition" from one state to another. A transition happens once every clock cycle. Depending on the current Input, it may go to a different state each time. Notice the number in the middle of every arrow. This is the current Input.

- For example, when in the "Initial-Stand by" state and a 1 is 'read', the diagram says to go to the "Activate Pulse" state. If a 0 is read then must stay on the "Initial-Stand by" state.

- So, what does the "Machine" do exactly? It starts from the "Initial - Stand by" state and waits until a 1 is read at the Input. Then it goes to the "Activate Pulse" state and transmits a HIGH pulse on its output. If the button keeps being pressed, the circuit goes to the third state, the "Wait Loop". There it waits until the button is released (Input goes 0) while transmitting a LOW on the output. Then it's all over again!

- This is possibly the most difficult part of the design procedure, because it cannot be described by simple steps. It takes exprerience and a bit of sharp thinking in order to set up a State Diagram, but the rest is just a set of predetermined steps.

Step 3

Next, the words that describe the different states of the diagram are replaced with **binary** numbers. starting the enumeration from 0 which is assigned on the initial state. then continuing the enumeration with any state liked, until all states have their number.

The result looks something like this: (Figure below)

A State Diagram with Coded States

Step 4

- Afterwards, filling the '*State Table' is done*. This table has a very specific form. giving the table of the example and use it to explain how to fill it in;

(Figure below)

A State Table

- The first columns are as many as the bits of the highest number assigned the State Diagram. If there were 5 states, up to the number 100could have been used, which means using 3 columns. For this example, up to the number 10 were used, so only 2 columns will be needed. These columns describe the '*Current State'* of the circuit.

- To the right of the Current State columns are written the '*Input Columns'*. These will be as many as theInput variables. this example has only one Input.

- Next, writing the '*Next State Columns'*. These are as many as the Current State columns.

- Finally, writing the '*Outputs Columns'*. These are as many as the outputs. the example has only one output. Since this is a More Finite State Machine, the output is dependent on only the current input states. This is the reason the outputs column has two ''s: to result in an output Boolean function that is independant of input I. Keep on reading for further details.

- The Current State and Input columns are the Inputs of the table. filling them in with all the binary numbers from 0 to

$$2^{(\text{Number of Current State columns} + \text{Number of Input columns})} - 1$$

- It is simpler than it sounds fortunately. Usually there will be more rows than the actual States we have created in the State Diagram, but that's ok.

- Each row of the Next State columns is filled as follows: filling it in with the state that is reached when, in the State Diagram, from the Current State of the same row following the Input of the same row. If a row whose Current State number doesn't correspond to any actual State in the State Diagram its filled with Don't Care terms (X). After all, it doesn't care where it can go from a State that doesn't exist. it wouldn't be there in the first place! Again it is simpler than it sounds.

- The outputs column is filled by the output of the corresponding Current State in the State Diagram.

- The State Table is complete! It describes the behaviour of our circuit as fully as the State Diagram does.

Step 5a

- The next step is to take that theoretical "Machine" and implement it in a circuit. Most often than not, this implementation involves Flip Flops. This guide is dedicated to this kind of implementation and will describe the procedure for both D - Flip Flops as well as JK - Flip Flops. T - Flip Flops will not be included as they are too similar to the two previous cases.

- The selection of the Flip Flop to use is arbitrary and usually is determined by cost factors. The best choice is to perform both analysis and decide which type of Flip Flop results in minimum number of logic gates and lesser cost.

- First, examining the "Machine" with D-Flip Flops.

- it will need as many D - Flip Flops as the State columns, 2 in this example. For every Flip Flop one more column in the State table will be added (Figure below) with the name of the Flip Flop's input, "D" for this case.

- The column that corresponds to each Flip Flop describes **what input must e given to the Flip Flop in order to go from the Current State to the Next State**. For the D - Flip Flop this is easy: The necessary input is equal to the Next State. In the rows that contain X's, X's are put in this column as well.

A State Table with D - Flip Flop Excitations

Step 5b

- the same can be done to steps with JK - Flip Flops. There are some differences however. A JK - Flip Flop has two inputs, therefore its needed to add two columns for each Flip Flop. The content of each cell is dictated by the JK's excitation table: (Figure below)

- JK - Flip Flop Excitation Table

Q	Q_{next}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

This table says that to want to go from State Q to State Q_{next}, its needed to use the specific input for each terminal. For example, to go from 0 to 1, its needed to feed J with 1 and it **doesn't care** which input is fed to terminal K.

A State Table with JK - Flip Flop Excitations

Step 6

- now the final stage of our procedure. What remains, is to determine the Boolean functions that produce the inputs of the Flip Flops and the Output. extracting one Boolean funtion for each Flip Flop input can be done with a 'Karnaugh Map'. The input variables of this map are the Current State variables **as well as** the Inputs.

- That said, the input functions for the D - Flip Flops are the following:

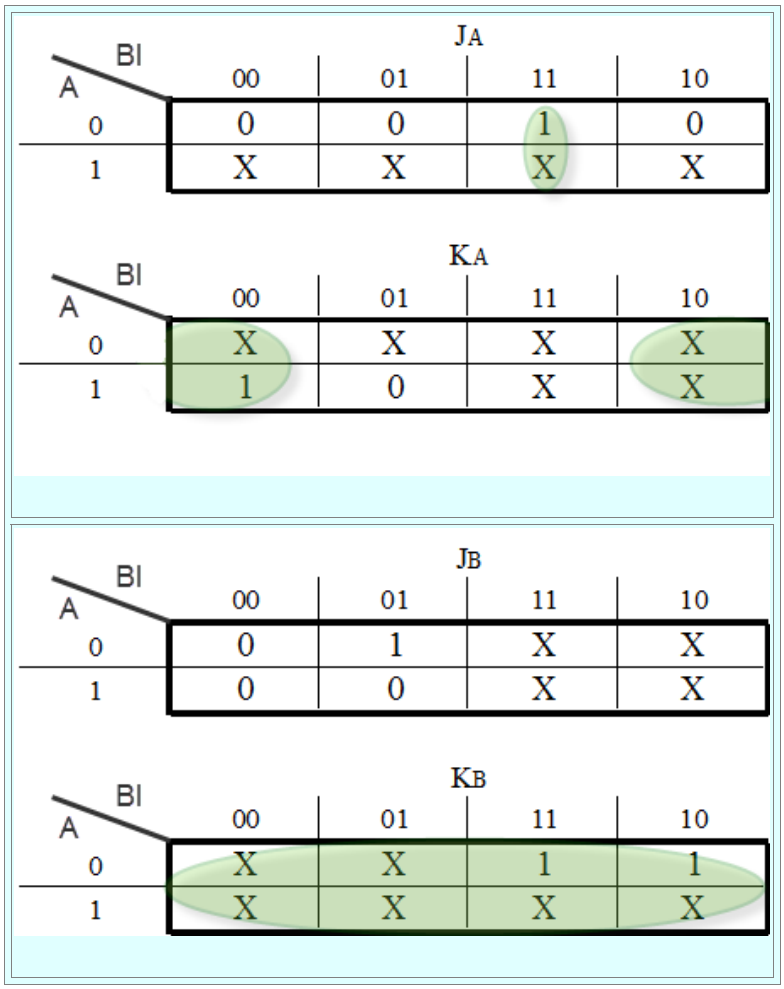
(Figure below)

Karnaugh Maps for the D - Flip Flop Inputs

$$D_A = A \cdot I + B \cdot I = (A+B) \cdot I$$
$$D_B = \overline{A} \cdot \overline{B} \cdot I$$

If the JK - Flip Flops are chosen, the functions would be the following:

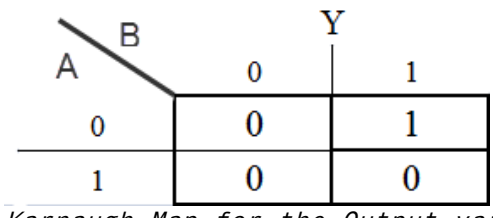
(Figure below)



Karnaugh Map for the JK - Flip Flop Input

$$J_A = B \cdot I$$
$$K_A = \overline{I}$$
$$J_B = \overline{A} \cdot I$$
$$K_B = 1$$

- A Karnaugh Map will be used to determine the function of the Output as well:
(Figure below)



Karnaugh Map for the Output variable Y

$$Y = \overline{A} \cdot B$$

Step 7

finally, design the circuit. placing the Flip Flops and using logic gates to form the Boolean functions that were calculated. The gates take input from the output of the Flip Flops and the Input of the circuit. Don't forget to connect the clock to the Flip Flops!

The D - Flip Flop version: (Figure below)

The completed D - Flip Flop Sequential Circuit

- The JK - Flip Flop version: (Figure below)

The completed JK - Flip Flop Sequential Circuit

That's it! a successfully designed and constructed a Sequential Circuit. At first it might seem a daunting task, but after practice and repetition the procedure becomes trivial. Sequential Circuits can come in handy as control parts of bigger circuits and can perform any sequential logic task imaginable. The sky is the limit! (or the circuit board, at least)

To Sum Up:

- A **Sequential Logic function** has a **'memory'** feature and **takes into account past inputs in order** to decide on the output.

- The **Finite State Machine** is an **abstract mathematical model of a sequential logic function**. It has finite inputs, outputs and number of states.
- **FSMs** are implemented in **real-life circuits** through the use of **Flip Flops**
- The **implementation procedure needs a** specific order of steps (**algorithm**), in order to be carried out.

>>>terminated early
>>>unfinished document
>>>see future transmissions

SHIFT REGISTERS

Introduction

- **Shift registers**, like counters, **are a form of 'sequential logic'**.
- Sequential logic, **unlike combinational logic** is not only affected by the present inputs, but also, by the prior history. In other words, **sequential logic remembers past events**.

- **Shift registers produce a discrete** delay of a **digital signal** or waveform. A **waveform synchronized to a 'clock'**, a repeating square wave, is **delayed** by '***n***' discrete clock counts, where '***n***' is **the number of shift register stages**.

- i.e., **a four stage shift register delays "data in" by four clocks to "data out"**. The **stages** in a shift register are ***delay stages***, typically type **"D"** Flip-Flops or type **"JK"** Flip-flops.

- Formerly, **very long (several hundred stages)** shift registers **served as digital memory**.
- **This obsolete application** is reminiscent of the acoustic mercury delay lines used as early computer memory. https://en.wikipedia.org/wiki/Delay-line_memory.

- **Serial data transmission**, over a distance of meters to kilometers, **uses shift registers to convert parallel data to serial form**.
- **Serial data communications replaces many slow parallel** data wires **with a single serial high speed** circuit.

- **Serial data over shorter** distances of tens of centimeters, uses shift registers to get data **into and out of microprocessors**.
- Numerous peripherals, including **analog to digital converters**, digital to analog converters, display drivers, and **memory**, use shift registers to **reduce the amount of wiring** in circuit boards.

- Some **specialized counter circuits** actually use shift registers to **generate repeating waveforms**.
- **Longer shift registers**, with the help of feedback **generate patterns so long** that they look like random noise, ***pseudo-noise***.
- This is Also how the patterns in the **Turing Machine modules generate patterns** although **based on true random input** into a shift register.

- Basic shift registers are **classified by structure** according to the following types:

  - **Serial-in/serial-out**
  - **Parallel-in/serial-out**
  - **Serial-in/parallel-out**
  - **Universal parallel-in/parallel-out**
  - **Ring counter**

SERIAL-IN/SERIAL OUT (SISO)

Above is shown a block diagram of a serial-in/serial-out (SISO)) shift register, which is 4-stages long.
Data at the input will be delayed by four clock periods from the input to the output of the shift register.

- **Data at 'data in'**, above, will be present at the **Stage A output after** the **first clock pulse**.
- **After the second pulse, stage A data** is transfered to **stage B output**, and 'data in' is transfered to stage **A** output.
- **After the third clock**, **stage C is replaced by stage B**; **stage B is replaced by stage A**; and, **stage A is replaced by 'data in'**.
- **After the fourth clock**, the data **originally present at 'data in' is at stage D, 'output'**.
- The '**first in**" data is '**first out**' as it is shifted from 'data in' to 'data out'.

PARALLEL-IN/SERIAL-OUT (PISO)

- **Data is loaded into all stages at once** of a parallel-in/serial-out (PISO) shift register.

- The **data** is **then shifted out via "data out"** by **clock pulses**.
- Since a 4-stage shift register is shown above, **four clock pulses are required to shift out all of the data**.
- In the diagram above, **stage D data** will **be present** at the **'data out' up until the first clock pulse**;
- **stage C data** will be present at "data out" between the **first clock and the second clock pulse**;
- **stage B data** will be present between the **second clock and the third clock**; and,
- **stage A data** will be present between the **third and the fourth clock**.
- **After the fourth clock pulse** and thereafter, successive bits of **'data in'** should **appear at 'data out'** of the shift register after a delay of **four clock pulses**.

- If **four switches** were **connected to D_A through D_D**, the s**tatus could be read #**
- **into** a microprocessor using only **one data pin and a clock pin**.
- Since **adding more switches would require no additional pins**, this approach looks **attractive for many inputs**.

SERIAL-IN/PARALELL-OUT (SIPO)

- Above, **four data bits will be shifted** in **from "data in"** by **four clock pulses** and be available at **$Q_A$through Q_D** for driving external circuitry such as LEDs, lamps or relay drivers.

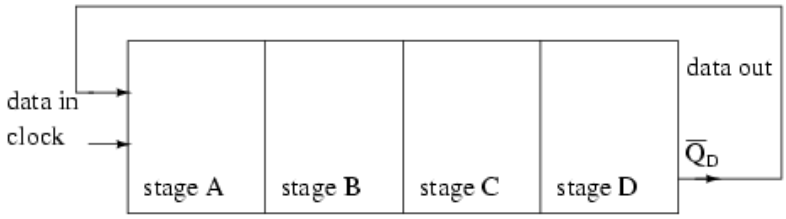
- **After the first clock**, the data at "**data in**" appears at **Q_A**.
- **After the second clock**, the **old Q_A** data appears at **Q_B**; **Q_A** receives next data from '**data in**'.
- **After the third clock**, **Q_B data** is at **Q_C**.
- **After the fourth clock**, **Q_C data** is at **Q_D**. This stage contains the data first present at '**data in**'.
- The shift register should **now contain four data bits**.

PARALELL-IN/PARALLEL-OUT (PIPO)

- A parallel-in/parallel-out (PIPO) shift register **combines** the function of the **PISO** shift register with the function of the **SIPO** shift register **to yield the** 'universal' **PIPO** shift register.
- The "do anything" shifter **comes at a price** – the **increased number of I/O** (Input/Output) **pins** may **reduce** the **number of stages** which can be packaged.

- **Data presented** at D_A **through** D_D is **parallel loaded** into the registers.
- This data at Q_A **through** Q_D may be **shifted by** the number of **pulses** presented at the **clock input**.
- The **shifted data** is available at Q_A **through** Q_D.
- The **'mode' input**, which may be more than one input, **controls parallel loading** of data from D_A **through** D_D, **shifting of data**, and the **direction of shifting**.
- There are **shift registers** which **will shift data either left or right**.

RING COUNTER



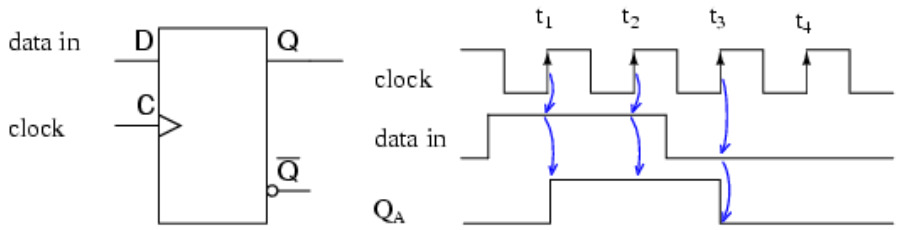
Ring Counter, shift register output fed back to input

- If the **serial output** of a shift register is **connected to the serial input**, data can be **perpetually shifted around the ring** as long as **clock pulses are present**.
- If the **output is inverted** before being fed back as shown above, we do **not have** to worry about loading the **initial data into the 'ring counter'**.

<u>Serial-in/serial-out shift register</u>

- **SISO** shift registers **delay data by one clock count for each stage**.
- **They will store a bit of data for each register**.
- A **SISO** shift register may be **one to 64 bits** in length, **longer if registers** or packages are **cascaded**.

- Below is a single stage shift register receiving data which is not synchronized to the register clock.
- The 'data in' at the **D** pin of the type **D FF** (Flip-Flop) **does not change levels when the clock changes for low to high**.
- it **may** be desired to **synchronize the data** to a **system wide clock** in a circuit board to improve the **reliability** of a digital logic circuit.



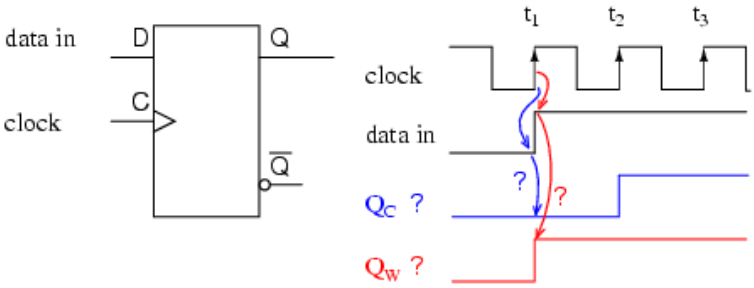
Data present at clock time is transfered from D to Q.

- The obvious point (as compared to the figure below) illustrated above is that whatever 'data in' is present at the **D** pin of a type **D** FF is transferred from D to output Q at clock time.
- Since the example shift register uses positive edge sensitive storage elements, the output **Q** follows the **D** input when the clock transitions from low to high as shown by the up arrows on the diagram above.

- There is no doubt what **logic level** is **present at clock time** because the **data is stable well before and after the clock edge**.
- This is actually seldom the case in multi-stage shift registers. But, this is an easy example to start with.

- this is only concerned with the positive, low to high, clock edge.
- The falling edge can be ignored.
- It is very easy to see **Q** follow **D** at clock time above.

- Compare this to the diagram below where the "data in" appears to change with the positive clock edge.



Does the clock t_1 see a 0 or a 1 at data in at D? Which output is correct, Q_C or Q_W ?

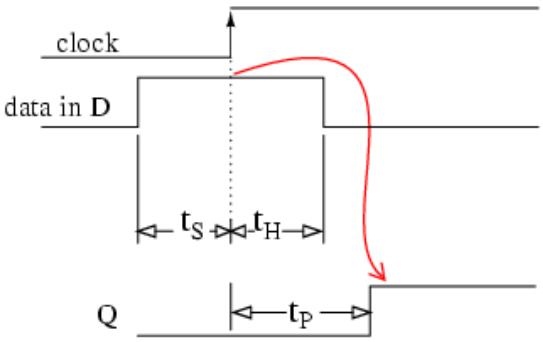
- Since "data in" appears to change at clock time t_1 above, what does the type **D** FF see at clock time? The short over simplified answer is that it sees the data that was present at **D** prior to the clock.
- That is what is transfered to **Q** at clock time t_1.

The correct waveform is Q_C.

- At t_1 Q goes to a zero if it is not already zero.
- The **D** register does not see a one until time t_2, at which time Q goes high.

- Since data, above, present at **D** is clocked to **Q** at clock time, and **Q** cannot change until the next clock time, the **D** FF delays data by one clock period, provided that the data is already synchronized to the clock.
- The Q_A waveform is the same as "data in" with a one clock period delay.

- A more detailed look at what the input of the type **D** Flip-Flop sees at clock time follows.
- Refer to the figure below.
- Since 'data in' appears to changes at clock time (above), we need further information to determine what the **D** FF sees.
If the 'data in' is from another shift register stage, another same type **D** FF, we can draw some conclusions based on '*data sheet*' information.
- Manufacturers of digital logic make available information about their parts in data sheets, formerly only available in a collection called a *data book*.
- Data books are still available; though, the manufacturer's web site is the modern source.



Data must be present (t_S) before the clock and after(t_H) the clock. Data is delayed from **D** to **Q** by propagation delay (t_P)

The following data was extracted from the CD4006b data sheet for operation at $5V_{DC}$, which serves as an example to illustrate timing.

[*]

*t_S=100ns
*t_H=60ns
*t_P=200-400ns typ/max

t_S is the *setup time*, the time data must be present before clock time. In this case data must be present at **D** 100ns prior to the clock.
- Furthermore, the data must be held for*hold time* t_H=60ns after clock time.
- These two conditions must be met to reliably clock data from **D** to **Q** of the Flip-Flop.

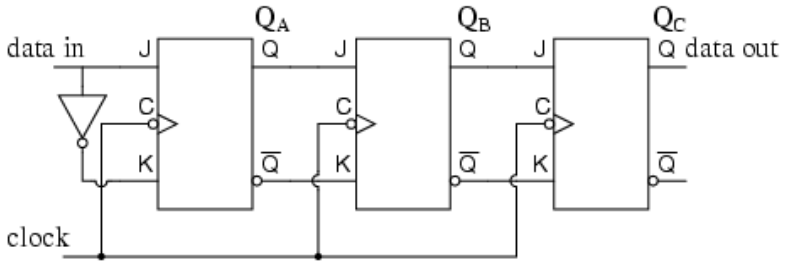
- There is no problem meeting the setup time of 60ns as the data at **D** has been there for the whole previous clock period if it comes from another shift register stage.
- For example, at a clock frequency of 1 Mhz, the clock period is 1000 µs, plenty of time.
- Data will actually be present for 1000µs prior to the clock, which is much greater than the minimum required t_S of 60ns.

- The hold time t_H=60ns is met because D connected to Q of another stage cannot change any faster than the propagation delay of the previous stage t_P=200ns. Hold time is met as long as the propagation delay of the previous **D** FF is

greater than the hold time. Data at **D** driven by another stage **Q** will not change any faster than 200ns for the CD4006b.

To summarize, output **Q** follows input D at nearly clock time if Flip-Flops are cascaded into a multi-stage shift register.

- Three type **D** Flip-Flops are cascaded Q to D and the clocks paralleled to form a three stage shift register above.



Serial-in, serial-out shift register using type "JK" storage elements

- Type **JK** FFs cascaded Q to J, Q' to K with clocks in parallel to yield an alternate form of the shift register above.

- A SISO shift register has a clock input, a data input, and a data output from the last stage.
- In general, the other stage outputs are not available Otherwise, it would be a SIPO shift register..

- The waveforms below are applicable to either one of the preceding two versions of the SISO shift register.
- The three pairs of arrows show that a three stage shift register temporarily stores 3-bits of data and delays it by three clock periods from input to output.

- At clock time t_1 a 'data in' of **0** is clocked from **D** to **Q** of all three stages.
- In particular, **D** of stage **A** sees a logic **0**, which is clocked to Q_A where it remains until time t_2.

- At clock time t_2 a "data in" of **1** is clocked from **D** to Q_A. At stages **B** and **C**, a **0**, fed from preceding stages is clocked to Q_B and Q_C.
- At clock time t_3 a "data in" of **0** is clocked from **D** to Q_A.
Q_A goes low and stays low for the remaining clocks due to 'data in' being **0**.
- Q_B goes high at t_3 due to a **1** from the previous stage.
- Q_C is still low after t_3 due to a low from the previous stage.

- Q_C finally goes high at clock t_4 due to the high fed to **D** from the previous stage Q_B.
- All earlier stages have **0**s shifted into them.
- And, after the next clock pulse at t_5, all logic **1**s will have been shifted out, replaced by **0**s

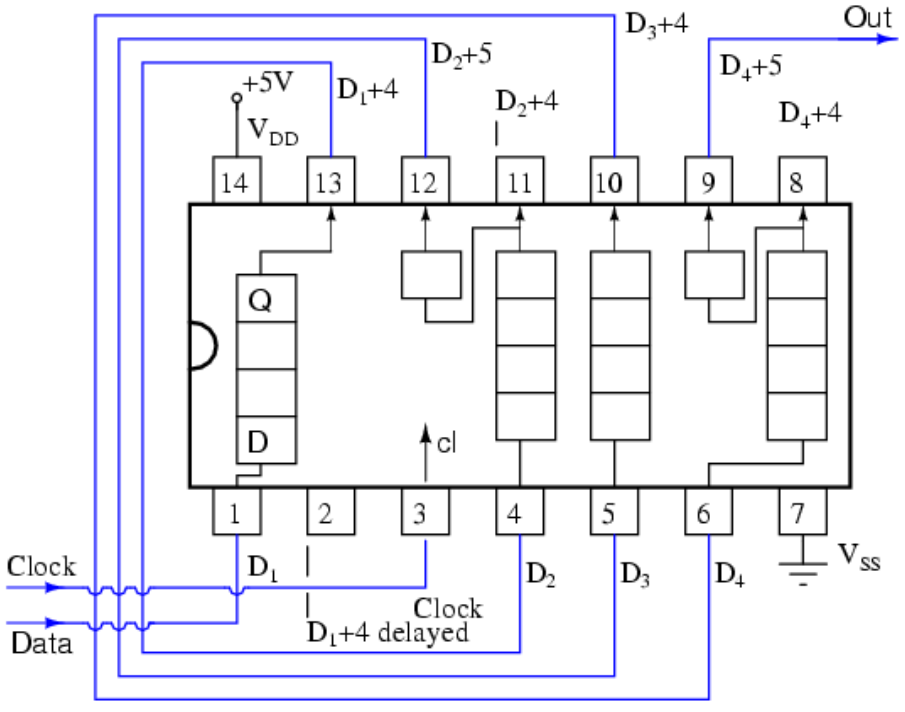
Serial-in/serial-out devices

We will take a closer look at the following parts available as integrated circuits, courtesy of Texas Instruments. For complete device data sheets follow the links.

- CD4006b 18-bit serial-in/ serial-out shift register [*]
- CD4031b 64-bit serial-in/ serial-out shift register [*]
- CD4517b dual 64-bit serial-in/ serial-out shift register [*]

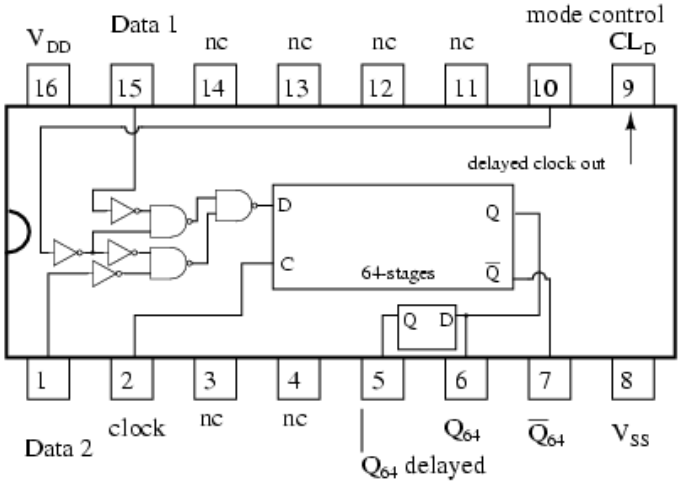
- The following serial-in/ serial-out shift registers are 4000 series *CMOS* (Complementary Metal Oxide Semiconductor) family parts.
As such, They will accept a V_{DD}, positive power supply of 3-Volts to 15-Volts. -
- The V_{SS} pin is grounded.
- The maximum frequency of the shift clock, which varies with V_{DD}, is a few megahertz. See the full data sheet for details.

- The 18-bit **CD4006b** consists of **two stages of 4-bits** and **two more stages of 5-bits** with a an output tap at 4-bits. Thus, the 5-bit stages could be used as 4-bit shift registers.
- To get a full 18-bit shift register the output of one shift register must be cascaded to the input of another and so on until all stages create a single shift register as shown below.



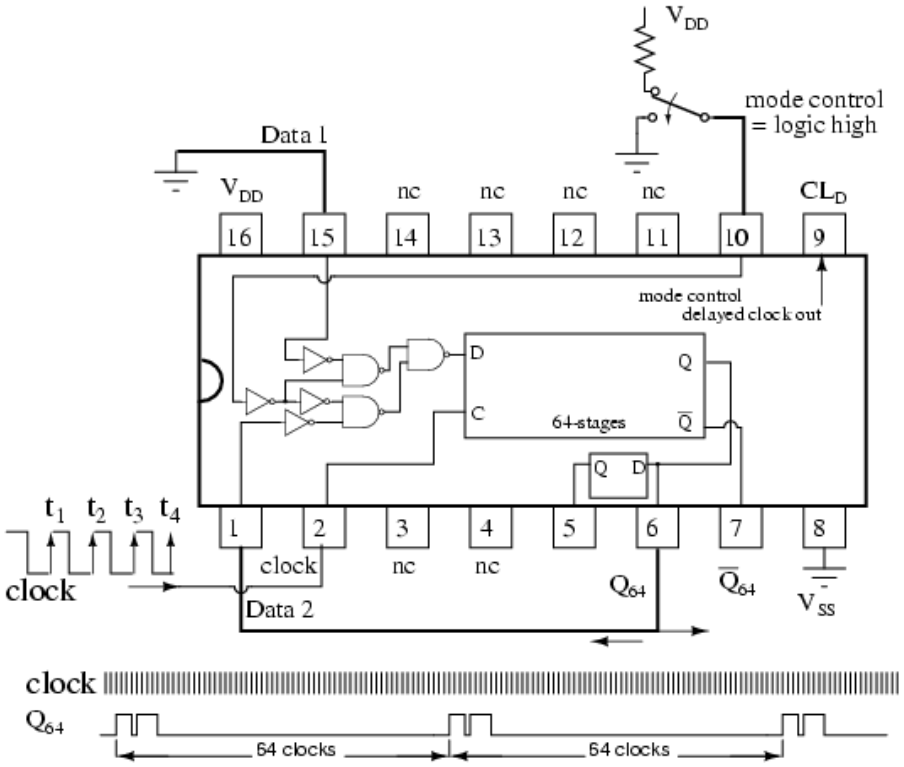
CD4006b 18-bit serial-in/ serial-out shift register

- A **CD4031** 64-bit serial-in/ serial-out shift register is shown below.
A number of pins are not connected (nc).
- **Both Q and Q'** are available from the **64th stage**, actually Q_{64} and Q'_{64}.
- There is **also a Q_{64} 'delayed'** from a half stage which is delayed by half a clock cycle.
- A major feature is a **data selector** which is at the data input to the shift register.



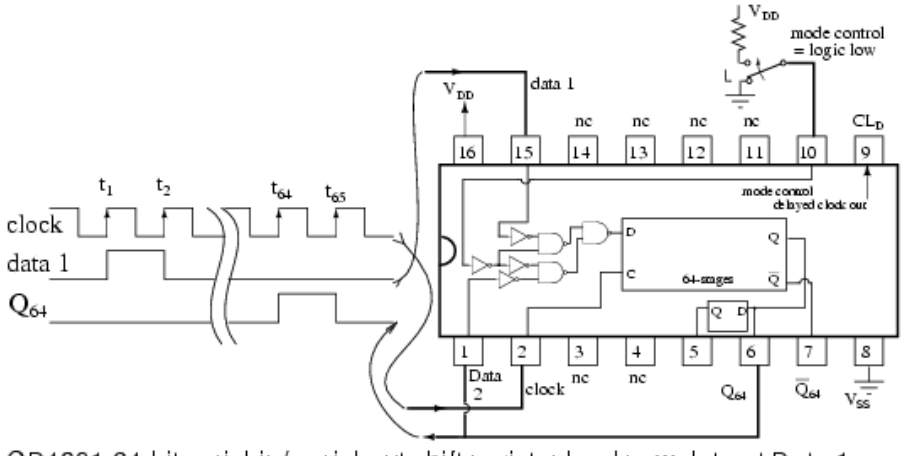
CD4031 64-bit serial-in/ serial-out shift register

- The "**mode control**" **selects between** two inputs: **data 1 and data 2**.
- If "**mode control**" is **high**, data will be selected from "**data 2**" for input to the shift register.
- In the case of "**mode control**" being logic **low**, the "**data 1**" is selected.
- Examples of this are shown in the two figures below.



CD4031 64-bit serial-in/ serial-out shift register recirculating data.

- The "**data 2**" above is wired **to** the **Q_{64}** output of the shift register. With "**mode control**" high, the **Q_{64}** output is routed back to the shifter data **input D**.
- **Data will '*recirculate*'** from output to input.
- The **data will repeat every 64 clock pulses** as shown above.
- The question that arises is how did this data pattern get into the shift register in the first place?



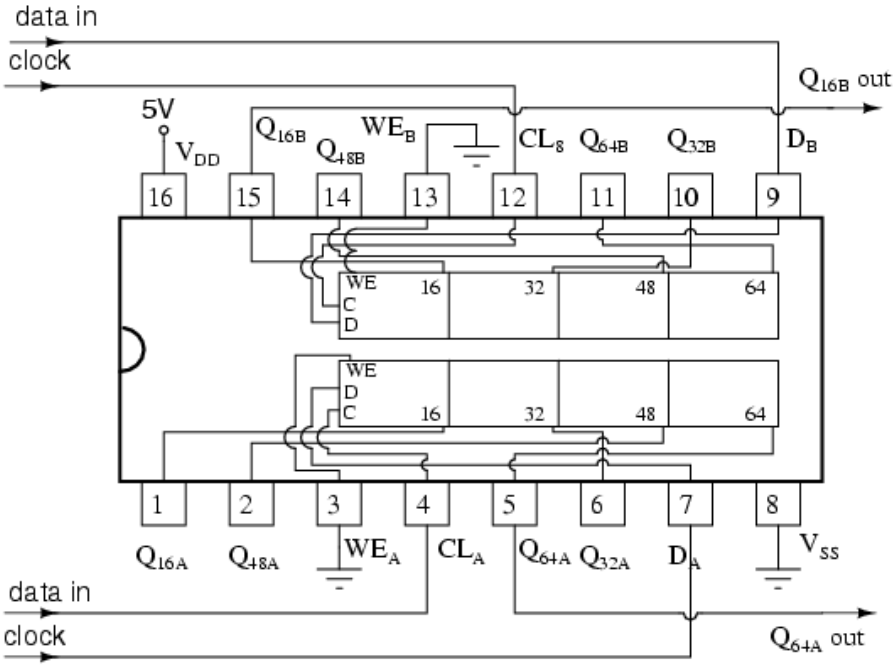
CD4031 64-bit serial-in/ serial-out shift register load new data at Data 1.

- With "**mode control**" **low**, the **CD4031** "**data 1**" is selected for input to the shifter.
- The **output, Q_{64}**, is **not recirculated** because the **lower data selector** gate is '***disabled***'.
- By disabled we mean that the logic low 'mode select' inverted twice to a low at the lower NAND gate prevents it for passing any signal on the lower pin (data 2) to the gate output.

- Thus, it is disabled.

- A **CD4517b** dual 64-bit shift register is shown above.
- Note the **taps** at the **16th, 32nd, and 48th** stages.
- That means that shift registers of those lengths can be configured from one of the **64-bit shifters**.
- Of course, the 64-bit shifters may be **cascaded** to yield an **80-bit, 96-bit, 112-bit, or 128-bit shift** register.
- The **clock CL_A and CL_B** need to be paralleled when cascading the two shifters.
- **WE_B and WE_B are grounded** for normal shifting operations.
- The **data inputs** to the shift registers A and B are **D_A** and **D_B** respectively.

- Suppose that a 16-bit shift register is required. Can this be configured with the **CD4517b**?
- How about a 64-shift register from the same part?



CD4517b dual 64-bit serial-in/ serial-out shift register, wired for 16-shift register, 64-bit shift register

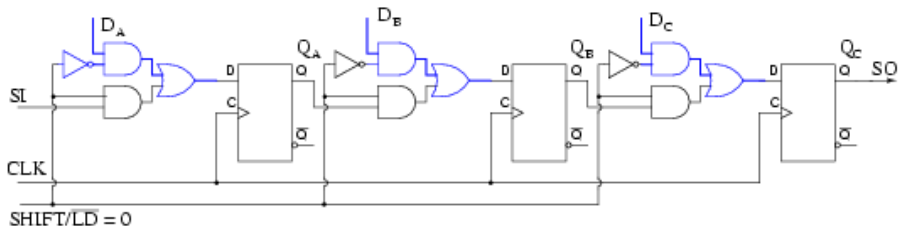
- Above is shown A **CD4517b** wired as a **16-bit** shift register for section B. The clock for **section B** is **CL_B**.
- The data is clocked in at **CL_B**.
- And the data delayed by 16-clocks is picked of off **Q_{16B}**.
- **WE_B**, the **write enable**, is **grounded**.

- Above is also shown the same **CD4517b** wired as a **64-bit** shift register for the independent **section A**.
- The **clock for section A is CL_A**. The **data enters at CL_A**.
- The **data** delayed by 64-clock pulses is picked up from **Q_{64A}**.
- **WE_A**, the **write enable** for **section A, is grounded**.

Parallel-in, serial-out (PISO) shift register

- **Parallel-in/ serial-out** shift registers do **everything that the previous** serial-in/ serial-out shift registers do **plus input data** to **all stages simultaneously**.
- The **PISO** shift register **stores data**, **shifts it** on a clock by clock basis, and **delays it** by the number of stages times the clock period.
- In addition, **PISO** really means that we **can load data in parallel into all stages** before any shifting ever begins.
- This is a way to **convert data from** a ***parallel*** format **to** a ***serial*** format.
- By parallel format we mean that the data bits are present simultaneously on individual wires, one for each data bit as shown below.
- By serial format we mean that the data bits are presented sequentially in time on a single wire or circuit as in the case of the "data out" on the block diagram below.

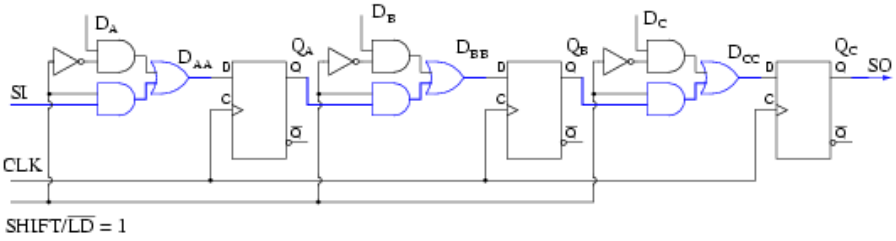
- Below is a close look at the internal details of a **3-stage** parallel-in/ serial-out shift register.
- **A stage consists** of a type **D Flip-Flop** for storage, and an **AND-OR** selector to determine whether data will load in parallel, or shift stored data to the right.
- In general, these elements will be replicated for the number of stages required.
- [Three stages are shown due to space limitations.]
- Four, eight or sixteen bits is normal for real parts.



Parallel-in/ serial-out shift register showing parallel load path

- Above is shown the parallel load path when SHIFT/LD' is logic low. The upper NAND gates serving D_A D_B D_C are enabled, passing data to the D inputs of type **D** Flip-Flops Q_A Q_B D_C respectively.
- At the next positive going clock edge, the data will be clocked from D to Q of the three FFs. Three bits of data will load into Q_A Q_B D_C at the same time.

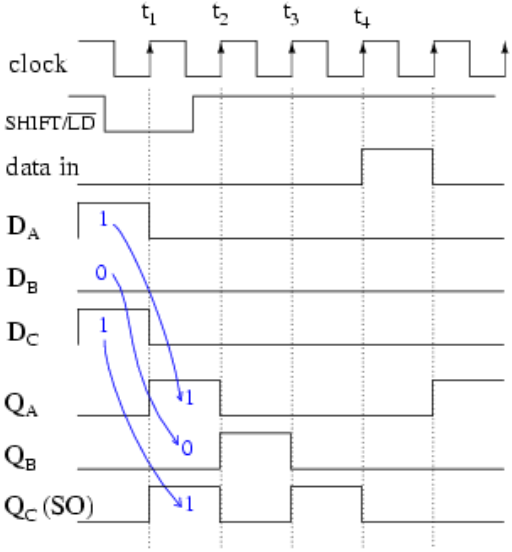
- The type of **parallel load** just described, where the **data loads on a clock pulse** is known as ***synchronous load*** because the loading of data is synchronized to the clock.
- This needs to be differentiated from *asynchronous load* where loading is controlled by the preset and clear pins of the Flip-Flops which does not require the clock.
- Only one of these load methods is used within an individual device, the **synchronous load being more common** in newer devices.



Parallel-in/ serial-out shift register showing shift path

- The shift path is shown above when **SHIFT/LD'** is logic **high**.
- The **lower AND** gates of the **pairs feeding** the **OR** gate are **enabled** giving us a shift register **connection** of **SI to D_A, Q_A to D_B, Q_B to D_C, Q_C to SO**.
- **Clock pulses** will cause **data** to be **right shifted out to SO** on successive pulses.

- The **waveforms below** show both **parallel loading** of **three bits** of data and **serial shifting** of this data. **Parallel data at D_A D_B D_C** is **converted to serial data at SO**.

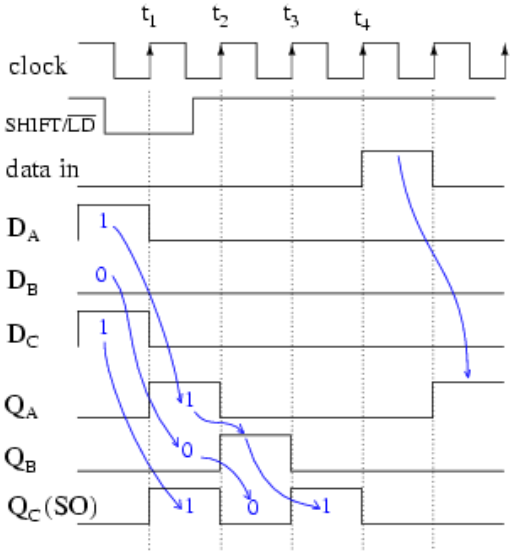


Parallel-in/ serial-out shift register load/shift waveforms

- What we previously described with words for parallel loading and shifting is now set down as waveforms above.
- As an example, presenting **101** to the parallel inputs D_{AA} D_{BB} D_{CC}.
- Next, the SHIFT/LD' goes low enabling loading of data as opposed to shifting of data.
- It needs to be low a short time before and after the clock pulse due to setup and hold requirements.
- It is considerably wider than it has to be.
- Though, with synchronous logic it is convenient to make it wide. We could have made the active low SHIFT/LD' almost two clocks wide, low almost a clock before t_1 and back high just before t_3.

- The important factor is that it needs to be low around clock time t_1 to enable parallel loading of the data by the clock.

- Note that at t_1 the data **101** at D_A D_B D_C is clocked from D to Q of the Flip-Flops as shown at Q_A Q_B Q_C at time t_1.
- This is the parallel loading of the data synchronous with the clock.



Parallel-in/ serial-out shift register load/shift waveforms

- Now that the data is loaded, we may shift it provided that SHIFT/LD' is high to enable shifting, which it is prior to t_2.
- At t_2 the data **0** at Q_C is shifted out of SO which is the same as the Q_C waveform.
- It is either shifted into another integrated circuit, or lost if there is nothing connected to SO.
- The data at Q_B, a **0** is shifted to Q_C.
The **1** at Q_A is shifted into Q_B.
- With "data in" a **0**, Q_A becomes **0**.
- After t_2, Q_A Q_B Q_C = **010**.

- After t_3, Q_A Q_B Q_C = **001**.
- This **1**, which was originally present at Q_A after t_1, is now present at SO and Q_C.
- The last data bit is shifted out to an external integrated circuit if it exists.
- After t_4 all data from the parallel load is gone.
- At clock t_5 we show the shifting in of a data **1** present on the SI, serial input.

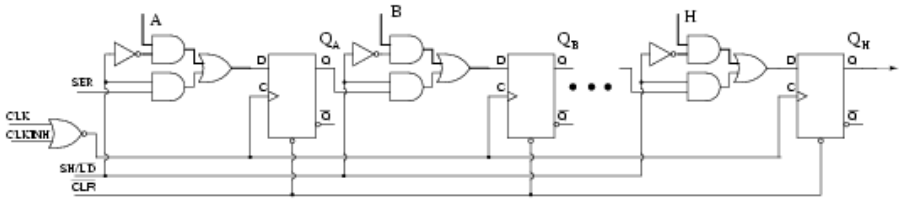
- Why provide SI and SO pins on a shift register? These connections allow us to cascade shift register stages to provide large shifters than available in a single IC (Integrated Circuit) package.
- They also allow serial connections to and from other ICs like microprocessors.

Parallel-in/serial-out (PISO) devices

Taking a closer look at parallel-in/ serial-out shift registers available as integrated circuits, courtesy of Texas Instruments.

- For complete device data sheets follow these the links.

SN74ALS166 parallel-in/ serial-out 8-bit shift register, synchronous load[*]
SN74ALS165 parallel-in/ serial-out 8-bit shift register, asynchronous load[*]
CD4014B parallel-in/ serial-out 8-bit shift register, synchronous load[*]
SN74LS647 parallel-in/ serial-out 16-bit shift register, synchronous load[*]



SN74ALS166 Parallel-in/ serial-out 8-bit shift register

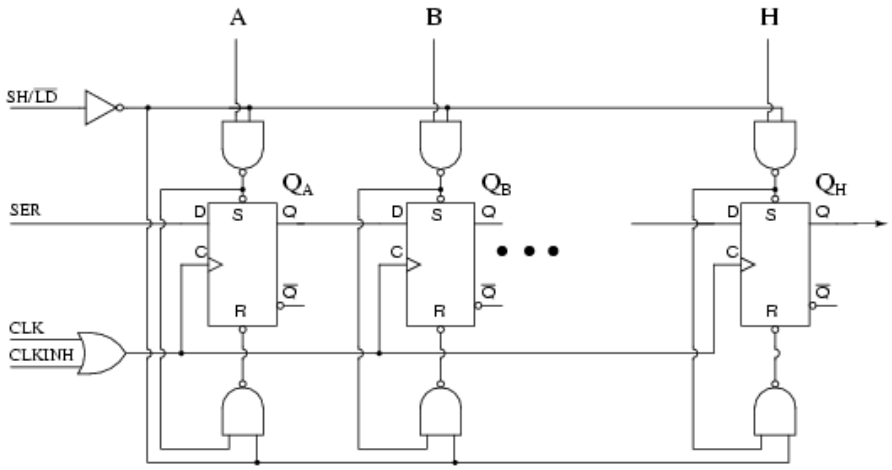
- The SN74ALS166 shown above is the closest match of an actual part to the previous parallel-in/ serial out shifter figures.
- Noting the minor changes to our figure above:
- First of all, there are 8-stages.
- Only three are shown.
- All 8-stages are shown on the data sheet available at the link above.
- The manufacturer labels the data inputs A, B, C, and so on to H.
- The SHIFT/LOAD control is called SH/LD'.
- It is abbreviated from the previous terminology, but works the same:
- parallel load if low, shift if high.
- The shift input (serial data in) is SER on the ALS166 instead of SI.
- The clock CLK is controlled by an inhibit signal, CLKINH. If CLKINH is high, the clock is inhibited, or disabled.
- Otherwise, this "real part" is the same as what we have looked at in detail.

- Above is the ANSI (American National Standards Institute) symbol for the SN74ALS166 as provided on the data sheet.
- Once its known how the part operates, it is convenient to hide the details within a symbol.
- There are many general forms of symbols. The advantage of the ANSI symbol is that the labels provide hints about how the part operates.

- The large notched block at the top of the '74ASL166 is the control section of the ANSI symbol. There is a reset indicted by **R**.
- There are three control signals: **M1** (Shift), **M2** (Load), and **C3/1 (arrow)** (inhibited clock).
- The clock has two functions.
- First, **C3** for shifting parallel data wherever a prefix of 3 appears.
- Second, whenever **M1** is asserted, as indicated by the **1** of **C3/1 (arrow)**, the data is shifted as indicated by the right pointing arrow.
- The slash (/) is a separator between these two functions.
- The 8-shift stages, as indicated by title **SRG8**, are identified by the external inputs **A, B, C,** to **H**.
- The internal **2, 3D** indicates that data, **D**, is controlled by **M2** [Load] and **C3** clock.
- In this case, we can conclude that the parallel data is loaded synchronously with the clock **C3**.
- The upper stage at **A** is a wider block than the others to accommodate the input **SER**.
- The legend **1, 3D** implies that **SER** is controlled by **M1** [Shift] and **C3** clock.

- Thus, we expect to clock in data at **SER** when shifting as opposed to parallel loading.

- The ANSI/IEEE basic gate *rectangular symbols* are provided above for comparison to the more familiar *shape symbols* so that the meaning of the symbology associated with the **CLKINH** and **CLK** pins on the previous ANSI SN74ALS166 symbol can be deciphered.
- The **CLK** and CLKINH feed an **OR** gate on the SN74ALS166 ANSI symbol.
- **OR** is indicated by **=>** on the rectangular inset symbol.
- The long triangle at the output indicates a clock.
- If there was a bubble with the arrow this would have indicated shift on negative clock edge (high to low).
- Since there is no bubble with the clock arrow, the register shifts on the positive (low to high transition) clock edge.
- The long arrow, after the legend **C3/1** pointing right indicates shift right, which is down the symbol.



SN74ALS165 Parallel-in/ serial-out 8-bit shift register, asynchronous load

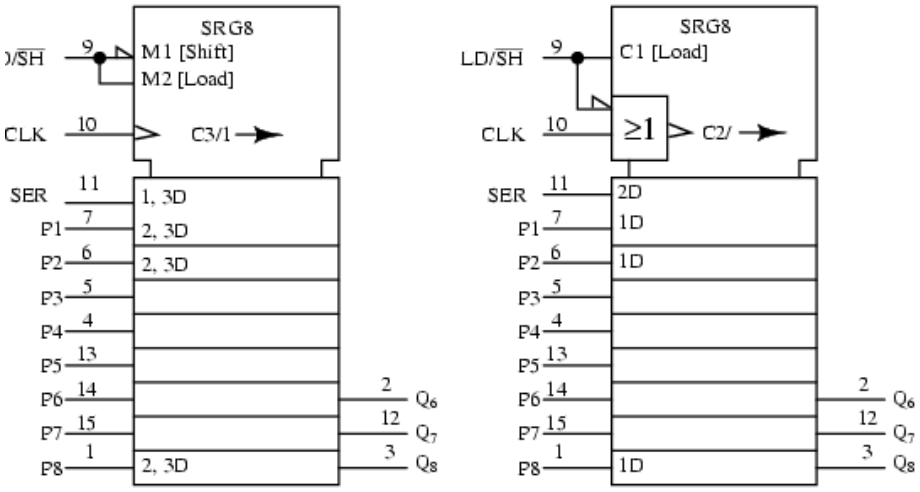
- Part of the internal logic of the SN74ALS165 parallel-in/ serial-out, asynchronous load shift register is reproduced from the data sheet above.
- See the link at the beginning of this section the for the full diagram.
- up to this point, asynchronous loading of data hasn't been looked at.
- First of all, the loading is accomplished by application of appropriate signals to the **Set** (preset) and **Reset** (clear) inputs of the Flip-Flops.
- The upper **NAND** gates feed the **Set** pins of the FFs and also cascades into the lower **NAND** gate feeding the **Reset** pins of the Ffs.
- The lower **NAND** gate inverts the signal in going from the **Set** pin to the **Reset** pin.
- First, **SH/LD'** must be pulled **Low** to enable the upper and lower **NAND** gates. If **SH/LD'** were at a logic **high** instead, the inverter feeding a logic **low** to all **NAND** gates would force a **High** out, releasing the "active low" **Set** and **Reset** pins of all Ffs.
- There would be no possibility of loading the FFs.

- With **SH/LD'** held **Low**, we can feed, for example, a data **1** to parallel input **A**, which inverts to a zero at the upper **NAND** gate output, setting FF Q_A to a **1**.
- The **0** at the **Set** pin is fed to the lower **NAND** gate where it is inverted to a **1**, releasing the **Reset** pin of Q_A.

- Thus, a data **A=1** sets Q_A=1.
- Since none of this required the clock, the loading is asynchronous with respect to the clock.
- We use an asynchronous loading shift register if we cannot wait for a clock to parallel load data, or if it is inconvenient to generate a single clock pulse.

- The only difference in feeding a data **0** to parallel input **A** is that it inverts to a **1** out of the upper gate releasing **Set**.
- This **1** at **Set** is inverted to a **0** at the lower gate, pulling **Reset** to a **Low**, which resets Q_A=0.

- The ANSI symbol for the SN74ALS166 above has two internal controls **C1** [LOAD] and **C2** clock from the **OR** function of (**CLKINH, CLK**).
- **SRG8** says 8-stage shifter.
- The arrow after **C2** indicates shifting right or down.
- **SER** input is a function of the clock as indicated by internal label **2D**.
- The parallel data inputs **A, B, C** to **H** are a function of **C1** [LOAD], indicated by internal label **1D**.
- **C1** is asserted when **sh/LD' =0** due to the half-arrow inverter at the input.
- Compare this to the control of the parallel data inputs by the clock of the previous synchronous ANSI SN75ALS166.
- Note the differences in the ANSI Data labels.



CD4014B, synchronous load CD4021B, asynchronous load

CMOS Parallel-in/ serial-out shift registers, 8-bit ANSI symbols

- On the CD4014B above, **M1** is asserted when **LD/SH'=0**.
- **M2** is asserted when **LD/SH'=1**.
- Clock **C3/1** is used for parallel loading data at **2, 3D** when **M2** is active as indicated by the **2,3** prefix labels.
- Pins **P3** to **P7** are understood to have the same internal **2,3** prefix labels as **P2** and **P8**.
- At **SER**, the **1,3D** prefix implies that **M1** and clock **C3** are necessary to input serial data.
- Right shifting takes place when M1 active is as indicated by the **1** in **C3/1 arrow**.

- The CD4021B is a similar part except for asynchronous parallel loading of data as implied by the lack of any **2** prefix in the data label **1D** for pins P1, P2, to P8.

- Of course, prefix **2** in label **2D** at input **SER** says that data is clocked into this pin.
- The **OR** gate inset shows that the clock is controlled by **LD/SH'**.

- The above SN74LS674 internal label **SRG 16** indicates 16-bit shift register.
- The **MODE** input to the control section at the top of the symbol is labeled **1,2 M3**.
- Internal **M3** is a function of input **MODE** and **G1** and **G2** as indicated by the **1,2** preceding **M3**.
- The base label **G** indicates an **AND** function of any such **G** inputs.
- Input **R/W'** is internally labeled **G1/2 EN**.
- This is an enable **EN** (controlled by **G1 AND G2**) for tristate devices used elsewhere in the symbol.
- We note that **CS'** on (pin 1) is internal **G2**.
- Chip select **CS'** also is **AND**ed with the input **CLK** to give internal clock **C4**.
- The bubble within the clock arrow indicates that activity is on the negative (high to low transition) clock edge.
- The slash (/) is a separator implying two functions for the clock. Before the slash,
- **C4** indicates control of anything with a prefix of **4**.

After the slash, the **3' (arrow)** indicates shifting. The **3'** of **C4/3'** implies shifting when M3 is de-asserted (**MODE=0**).
- The long arrow indicates shift right (down).

- Moving down below the control section to the data section, are external inputs **P0-P15**, pins (7-11, 13-23).
- The prefix **3,4** of internal label **3,4D** indicates that **M3** and the clock **C4** control loading of parallel data.
- The **D** stands for Data.
- This label is assumed to apply to all the parallel inputs, though not explicitly written out.
- Locate the label **3',4D** on the right of the **P0** (pin7) stage.
- The complemented-**3** indicates that **M3=MODE=0** inputs (shifts) **SER/Q_{15}** (pin5) at clock time, (**4** of 3',4D) corresponding to clock **C4**.
- In other words, with **MODE=0**, data shifts into **Q_0** from the serial input (pin 6).

- All other stages shift right (down) at clock time.

- Moving to the bottom of the symbol, the triangle pointing right indicates a buffer between **Q** and the output pin.
- The Triangle pointing down indicates a tri-state device.
- Previously it was stated that the tristate is controlled by enable **EN**, which is actually **G1 AND G2** from the control section.
- If **R/W=0**, the tri-state is disabled, and data can be shifted into Q_0 via **SER** (pin 6), a detail omitted above.
- actually **MODE=0, R/W'=0, CS'=0** is needed.

- The internal logic of the SN74LS674 and a table summarizing the operation of the control signals is available in the link in the bullet list, top of section.

- If **R/W'=1**, the tristate is enabled,
- Q_{15} shifts out **SER/**Q_{15} (pin 6) and recirculates to the Q_0 stage via the right hand wire to **3',4D**.
This has assumed that CS' was low giving clock C4/3' and G2 to **EN**able the tri-state.

Practical applications

- An application of a parallel-in/ serial-out shift register is to read data into a microprocessor.

- The Alarm above is controlled by a remote keypad.
- The alarm box supplies +5V and ground to the remote keypad to power it.
- The alarm reads the remote keypad every few tens of milliseconds by sending shift clocks to the keypad which returns serial data showing the status of the keys via a parallel-in/ serial-out shift register.
Thus, the nine key switches are read with four wires.
- How many wires would be required if we had to run a circuit for each of the nine keys?

- A practical application of a **parallel-in/ serial-out** shift register is to read **many switch closures** into a microprocessor on **just a few pins**.
- Some low end microprocessors only have 6-I/O (Input/Output) pins available on an 8-pin package.
- Or, we may have used most of the pins on an 84-pin package.
- We may want to **reduce the number of wires** running around a circuit board, machine, vehicle, or building.
- **This will increase the reliability of our system**.
- It has been reported that manufacturers who have reduced the number of wires in an automobile produce a more reliable product.
- In any event, **only three microprocessor pins** are required to read in **8-bits of data** from the switches in the figure above.

- We have chosen an asynchronous loading device, the CD4021B because it is easier to control the loading of data without having to generate a single parallel load clock.
- The parallel data inputs of the shift register are pulled up to +5V with a reistor on each input.
- **If all switches are open**, all **1**s will be loaded into the shift register when -themicroprocessor moves the **LD/SH'** line from low to high, then back low in anticipation of shifting.
- Any switch closures will apply logic **0**s to the corresponding parallel inputs.
- The data pattern at P1-P7 will be parallel loaded by the **LD/SH'=1** generated by the microprocessor software.

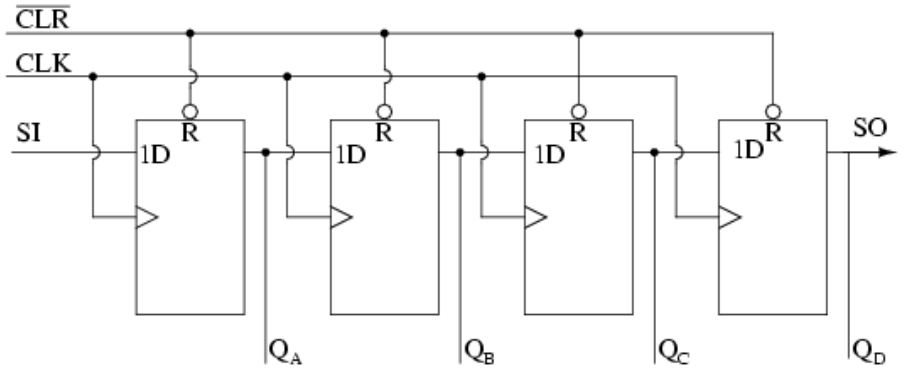
- The **microprocessor generates shift pulses** and **reads a data bit** for **each** of the **8-bits**.
- This process may be performed totally with software, or larger microprocessors may have one or more serial interfaces to do the task more quickly with hardware.
- With **LD/SH'=0**, the microprocessor generates a **0** to **1** transition on the **Shift clock line**, then reads a data bit on the **Serial data in** line. This is repeated for all 8-bits.

- The **SER** line of the shift register may be driven by another identical CD4021B circuit if more switch contacts need to be read.
- In which case, **the microprocessor generates 16-shift pulses**. **More likely**, it will be **driven by something else** compatible with this serial data format, for example, an analog to digital converter, a temperature sensor, a keyboard scanner, a serial read-only memory. As for the switch closures, they may be limit switches on the carriage of a machine, an over-temperature sensor, a magnetic reed switch, a door or window switch, an air or water pressure switch, or a solid state optical interrupter.

Serial-in, parallel-out [SIPO] shift register

- A serial-in/parallel-out **[SIPO] shift register is similar to** the **SISO** shift register **in that it shifts data into internal storage elements** and **shifts data out at the serial-out**, data-out, pin.
- It is **different in that** it makes all the **internal stages available as outputs**. Therefore, a serial-in/parallel-out shift register converts data from serial format to parallel format.
- **If four data bits** are **shifted in** by four clock pulses via a single wire at data-in, below, the **data becomes available simultaneously on the four Outputs** Q_A to Q_D after the **fourth clock pulse**.

- The **practical application** of the serial-in/parallel-out shift register is **to convert data from serial format** on a single wire **to parallel format** on multiple wires.
- Perhaps, we will illuminate four LEDs (Light Emitting Diodes) with the four outputs (**Q_A Q_B Q_C Q_D**).

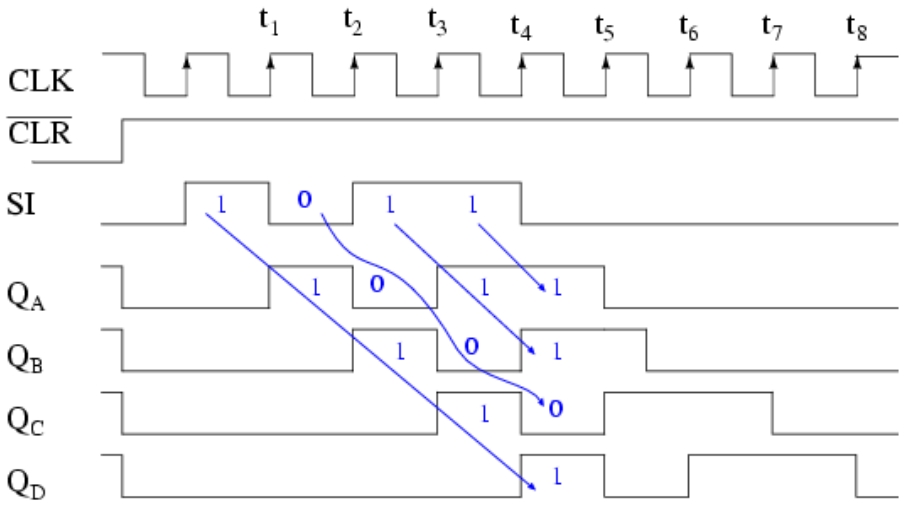


Serial-in/ Parallel out shift register details

- The above details of the serial-in/parallel-out shift register are fairly simple.
- It looks like a **serial-in/ serial-out** shift register with **taps added to each stage** output.
- Serial data shifts in at **SI** (Serial Input).

- After a number of clocks equal to the number of stages, the **first data bit in appears at SO (Q_D)** in the above figure.
- In general, there is no SO pin. The last stage (Q_D above) serves as SO and is cascaded to the next package if it exists.
- If a serial-in/parallel-out shift register is so similar to a serial-in/ serial-out shift register, why do manufacturers bother to offer both types? Why not just offer the serial-in/parallel-out shift register?
- They actually only offer the serial-in/parallel-out shift register, as long as it has no more than 8-bits.
- Note that serial-in/ serial-out shift registers come in bigger than 8-bit lengths of 18 to to 64-bits.
- It is not practical to offer a 64-bit serial-in/parallel-out shift register requiring that many output pins.

- See waveforms below for above shift register.



Serial-in/ parallel-out shift register waveforms

- The shift register has been cleared prior to any data by **CLR'**, an active low signal, which clears all type D Flip-Flops within the shift register.
- Note the serial data **1011** pattern presented at the **SI** input. This data is synchronized with the clock **CLK**.
- This would be the case if it is being shifted in from something like another shift register, for example, a parallel-in/ serial-out shift register (not shown here).

- On the first clock at **t1**, the data **1** at **SI** is shifted from **D** to **Q** of the first shift register stage.
- After **t2** this first data bit is at Q_B. After **t** it is at Q_C.
- After **t4** it is at Q_D. Four clock pulses have shifted the first data bit all the way to the last stage Q_D.
- The second data bit a **0** is at Q_C after the 4th clock.
- The third data bit a **1** is at Q_B.
- The fourth data bit another **1** is at Q_A.
- Thus, the serial data input pattern **1011** is contained in (Q_D Q_C Q_B Q_A).
- It is now available on the four outputs.
- It will available on the four outputs from just after clock t_4 to just before t_5.
- This parallel data must be used or stored between these two times, or it will be lost due to shifting out the Q_D stage on following clocks t_5 to t_8 as shown above.

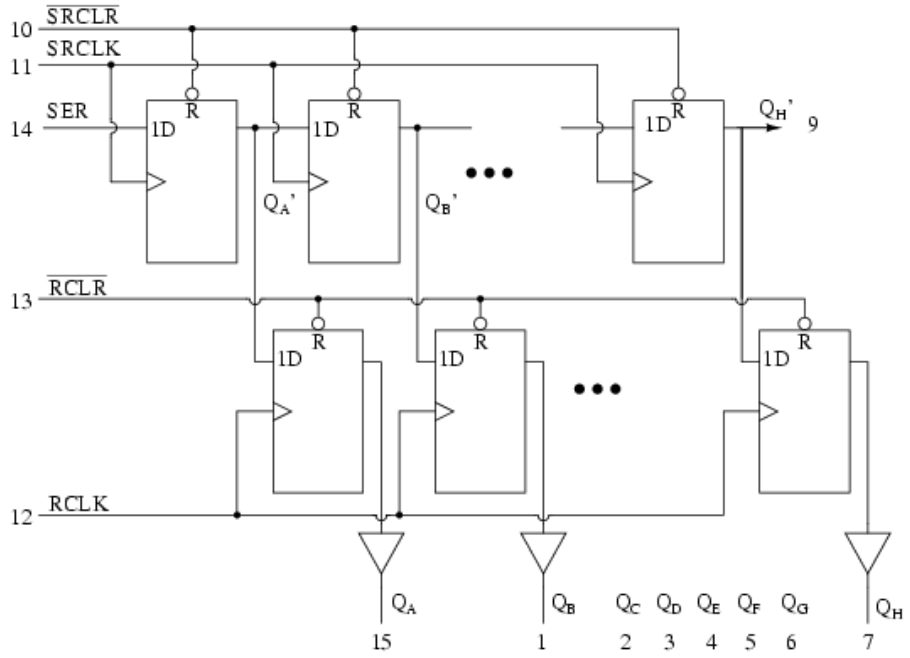
Serial-in/ parallel-out devices

- Let's take a closer look at Serial-in/ parallel-out shift registers available as integrated circuits, courtesy of Texas Instruments.
- For complete device data sheets follow the links.

- SN74ALS164A serial-in/ parallel-out 8-bit shift register[*]
- SN74AHC594 serial-in/ parallel-out 8-bit shift register with output register[*]
- SN74AHC595 serial-in/ parallel-out 8-bit shift register with output register[*]
- CD4094 serial-in/ parallel-out 8-bit shift register with output register[*] [*]

- The 74ALS164A is almost identical to our prior diagram with the exception of the two serial inputs **A** and **B**.
- The unused input should be pulled high to enable the other input.
- We do not show all the stages above.
- However, all the outputs are shown on the ANSI symbol below, along with the pin numbers.

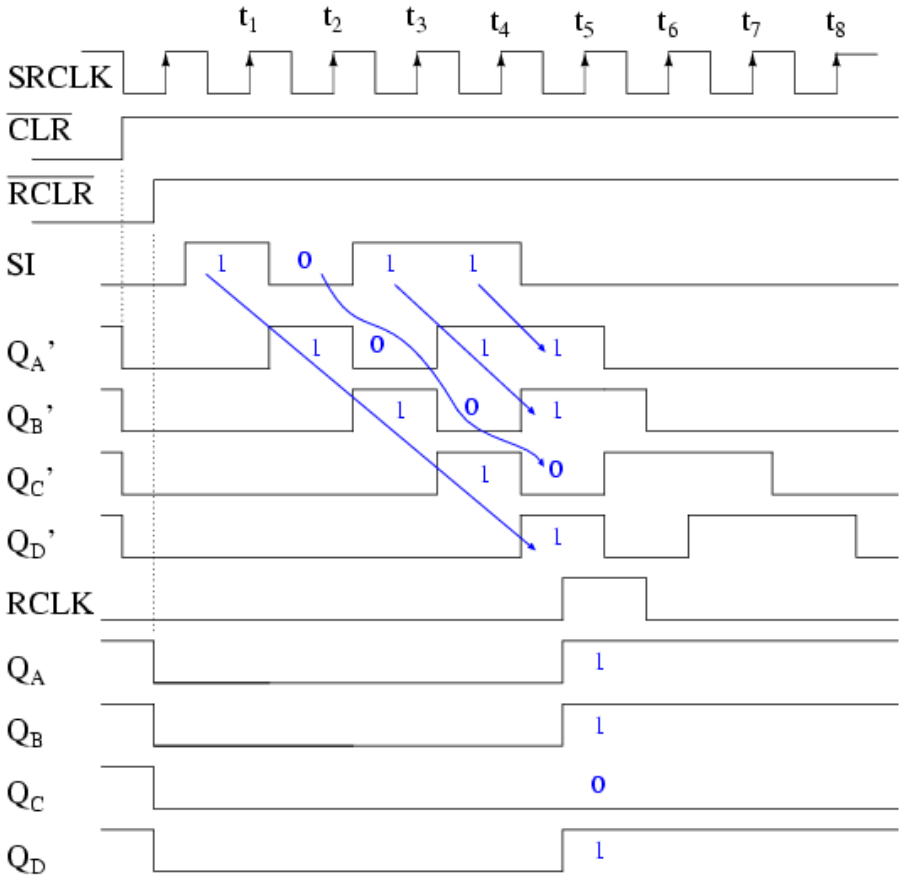
- The **CLK** input to the control section of the above ANSI symbol has two internal functions **C1**, control of anything with a prefix of **1**.
- This would be clocking in of data at **1D**. The second function, the arrow after after the slash (/) is right (down) shifting of data within the shift register.
- The eight outputs are available to the right of the eight registers below the control section.
- The first stage is wider than the others to accommodate the **A&B** input.



74AHC594 Serial-in/ Parallel out 8-bit shift register with output registers

- The above internal logic diagram is adapted from the TI (Texas Instruments) data sheet for the 74AHC594.
- The type "D" FFs in the top row comprise a serial-in/ parallel-out shift register.
- This section works like the previously described devices.
- The outputs (Q_A' Q_B' to Q_H') of the shift register half of the device feed the type "D" FFs in the lower half in parallel.
- Q_H' (pin 9) is shifted out to any optional cascaded device package.
- A single positive clock edge at RCLK will transfer the data from **D** to **Q** of the lower Ffs.
- All 8-bits transfer in parallel to the output *register* (a collection of storage elements).
- The purpose of the output register is to maintain a constant data output while new data is being shifted into the upper shift register section.
- This is necessary if the outputs drive relays, valves, motors, solenoids, horns, or buzzers.
- This feature may not be necessary when driving LEDs as long as flicker during shifting is not a problem.

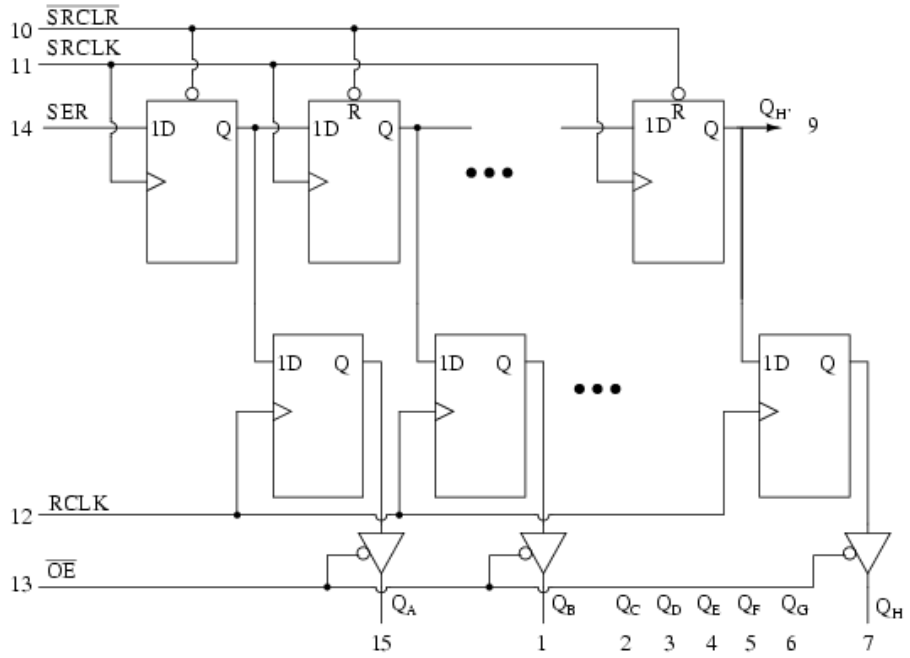
- Note that the 74AHC594 has separate clocks for the shift register (**SRCLK**) and the output register (**RCLK**).
- Also, the shifter may be cleared by **SRCLR** and, the output register by **RCLR**.
- It desirable to put the outputs in a known state at power-on, in particular, if driving relays, motors, etc.
- The waveforms below illustrate shifting and latching of data.



Waveforms for 74AHC594 serial-in/ parallel-out shift registe rwith latch

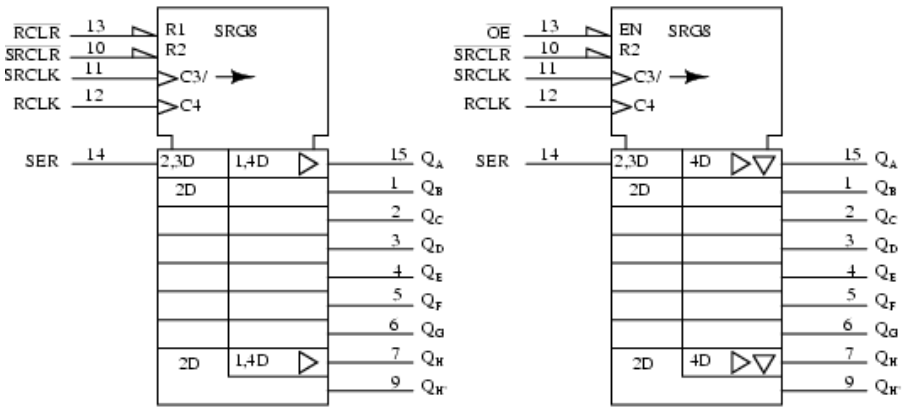
The above waveforms show shifting of 4-bits of data into the first four stages of 74AHC594, then the parallel transfer to the output register. In actual fact, the 74AHC594 is an 8-bit shift register, and it would take 8-clocks to shift in

8-bits of data, which would be the normal mode of operation. However, the 4-bits we show saves space and adequately illustrates the operation.
We clear the shift register half a clock prior to t_0 with **SRCLR'=0**. **SRCLR'** must be released back high prior to shifting. Just prior to t_0 the output register is cleared by **RCLR'=0**. It, too, is released (**RCLR'=1**).
Serial data **1011** is presented at the SI pin between clocks t_0 and t_4. It is shifted in by clocks t_1 t_2 t_3 t_4 appearing at internal shift stages Q_A' Q_B' Q_C' Q_D'. This data is present at these stages between t_4 and t_5. After t_5 the desired data (**1011**) will be unavailable on these internal shifter stages.
Between t_4 and t_5 we apply a positive going **RCLK** transferring data **1011** to register outputs Q_A Q_B Q_C Q_D . This data will be frozen here as more data (**0**s) shifts in during the succeeding **SRCLK**s (t_5 to t_8). There will not be a change in data here until another **RCLK** is applied.



74AHC595 Serial-in/ Parallel out 8-bit shift register with output registers

- The 74AHC595 is identical to the '594 except that the **RCLR'** is replaced by an **OE'** enabling a tri-state buffer at the output of each of the eight output register bits.
- Though the output register cannot be cleared, the outputs may be disconnected by **OE'=1**.
- This would allow external pull-up or pull-down resistors to force any relay, solenoid, or valve drivers to a known state during a system power-up.
- Once the system is powered-up and, say, a microprocessor has shifted and latched data into the '595, the output enable could be asserted (**OE'=0**) to drive the relays, solenoids, and valves with valid data, but, not before that time.



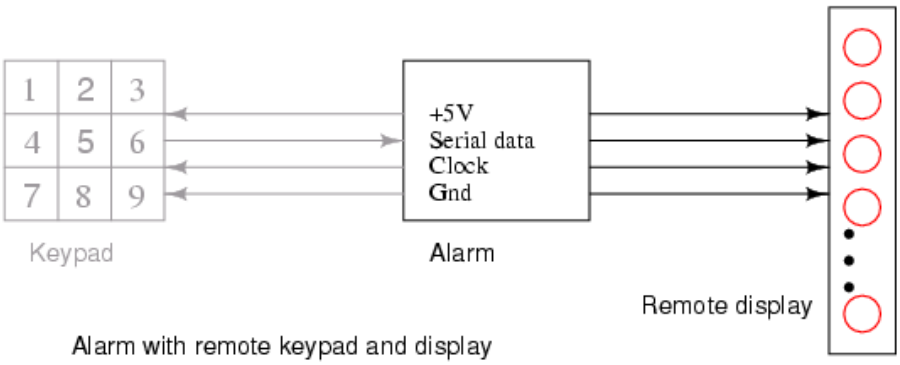
SN74AHC594 ANSI Symbol SN74AHC595 ANSI Symbol

- Above are the proposed ANSI symbols for these devices.
- **C3** clocks data into the serial input (external **SER**) as indicate by the **3** prefix of **2,3D**.
- The arrow after **C3/** indicates shifting right (down) of the shift register, the 8-stages to the left of the '595symbol below the control section.
- The **2** prefix of **2,3D** and **2D** indicates that these stages can be reset by **R2** (external **SRCLR'**).
- The **1** prefix of **1,4D** on the **'594** indicates that **R1** (external **RCLR'**) may reset the output register, which is to the right of the shift register section.
- The **'595**, which has an **EN** at external **OE'** cannot reset the output register. But, the **EN** enables tristate (inverted triangle) output buffers.
- The right pointing triangle of both the **'594** and**'595** indicates internal buffering.
- Both the **'594** and**'595** output registers are clocked by C4 as indicated by **4** of **1,4D** and **4D** respectively.

- The CD4094B is a 3 to 15V_{DC} capable latching shift register alternative to the previous 74AHC594 devices. **CLOCK**, **C**1, shifts data in at **SERIAL IN** as implied by the **1** prefix of **1D**.
- It is also the clock of the right shifting shift register (left half of the symbol body) as indicated by the /(right-arrow) of **C1**/(arrow) at the **CLOCK** input.
- **STROBE**, **C2** is the clock for the 8-bit output register to the right of the symbol body.
- The **2** of **2D** indicates that **C2** is the clock for the output register.
- The inverted triangle in the output latch indicates that the output is tristated, being enabled by **EN3**.
- The **3** preceding the inverted triangle and the **3** of **EN3** are often omitted, as any enable (**EN**) is understood to control the tristate outputs.
- Q_S and Q_S' are non-latched outputs of the shift register stage. Q_S could be cascaded to **SERIAL IN** of a succeeding device.

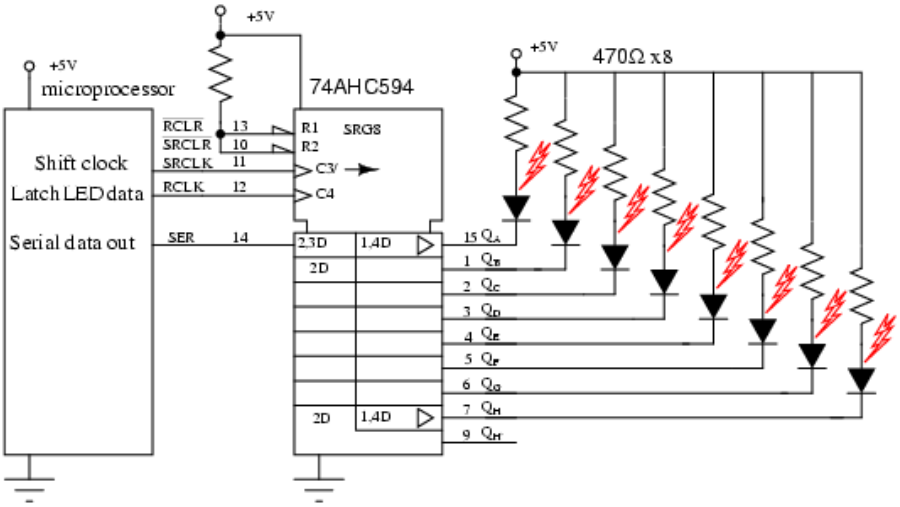
Practical applications

- A real-world application of the serial-in/ parallel-out shift register is to output data from a microprocessor to a remote panel indicator.
- Or, another remote output device which accepts serial format data.



Alarm with remote keypad and display

The figure "Alarm with remote key pad" is repeated here from the parallel-in/ serial-out section with the addition of the remote display. Thus, we can display, for example, the status of the alarm loops connected to the main alarm box. If the Alarm detects an open window, it can send serial data to the remote display to let us know. Both the keypad and the display would likely be contained within the same remote enclosure, separate from the main alarm box. However, we will only look at the display panel in this section.
If the display were on the same board as the Alarm, we could just run eight wires to the eight LEDs along with two wires for power and ground. These eight wires are much less desirable on a long run to a remote panel. Using shift registers, we only need to run five wires- clock, serial data, a strobe, power, and ground. If the panel were just a few inches away from the main board, it might still be desirable to cut down on the number of wires in a connecting cable to improve reliability. Also, we sometimes use up most of the available pins on a microprocessor and need to use serial techniques to expand the number of outputs. Some integrated circuit output devices, such as Digital to Analog converters contain serial-in/ parallel-out shift registers to receive data from microprocessors. The techniques illustrated here are applicable to those parts.



Output to LEDs from microprocessor

We have chosen the 74AHC594 serial-in/ parallel-out shift register with output register; though, it requires an extra pin, **RCLK**, to parallel load the shifted-in data to the output pins. This extra pin prevents the outputs from changing while data is shifting in. This is not much of a problem for LEDs. But, it would be a problem if driving relays, valves, motors, etc.
Code executed within the microprocessor would start with 8-bits of data to be output. One bit would be output on the "Serial data out" pin, driving **SER** of the remote 74AHC594. Next, the microprocessor generates a low to high transition on "Shift clock", driving **SRCLK** of the '595 shift register. This positive clock shifts the data bit at **SER** from "D" to "Q" of the first shift register stage.

This has no effect on the Q_A LED at this time because of the internal 8-bit output register between the shift register and the output pins (Q_A to Q_H). Finally, "Shift clock" is pulled back low by the microprocessor. This completes the shifting of one bit into the '595.

The above procedure is repeated seven more times to complete the shifting of 8-bits of data from the microprocessor into the 74AHC594 serial-in/ parallel-out shift register. To transfer the 8-bits of data within the internal '595 shift register to the output requires that the microprocessor generate a low to high transition on **RCLK**, the output register clock. This applies new data to the LEDs. The **RCLK** needs to be pulled back low in anticipation of the next 8-bit transfer of data.
The data present at the output of the '595 will remain until the process in the above two paragraphs is repeated for a new 8-bits of data. In particular, new data can be shifted into the '595 internal shift register without affecting the LEDs. The LEDs will only be updated with new data with the application of the **RCLK** rising edge.

What if we need to drive more than eight LEDs? Simply cascade another 74AHC594 **SER** pin to the Q_H' of the existing shifter. Parallel the **SRCLK** and **RCLK** pins. The microprocessor would need to transfer 16-bits of data with 16-clocks before generating an **RCLK** feeding both devices.
The discrete LED indicators, which we show, could be 7-segment LEDs. Though, there are LSI (Large Scale Integration) devices capable of driving several 7-segment digits. This device accepts data from a microprocessor in a serial format, driving more LED segments than it has pins by by multiplexing the LEDs. For example, see link below for MAX6955[*]

Parallel-in, parallel-out, universal shift register

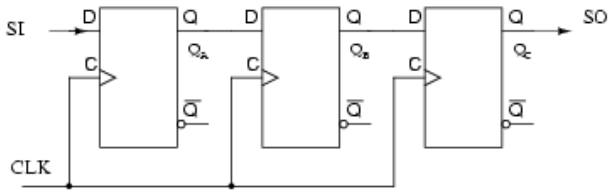
- The purpose of the parallel-in/ parallel-out shift register is to take in parallel data, shift it, then output it as shown below.
- A universal shift register is a do-everything device in addition to the parallel-in/ parallel-out function.

- Above we apply four bit of data to a parallel-in/ parallel-out shift register at D_A D_B D_C D_D.
- The mode control, which may be multiple inputs, controls parallel loading vs shifting.
- The mode control may also control the direction of shifting in some real devices.
- The data will be shifted one bit position for each clock pulse.
- The shifted data is available at the outputs Q_A Q_B Q_C Q_D .
- The "data in" and "data out" are provided for cascading of multiple stages. Though, above, we can only cascade data for right shifting.
- We could accommodate cascading of left-shift data by adding a pair of left pointing signals, "data in" and "data out", above.
- The internal details of a right shifting parallel-in/ parallel-out shift register are shown below.
- The tri-state buffers are not strictly necessary to the parallel-in/ parallel-out shift register, but are part of the real-world device shown below.

- The 74LS395 so closely matches our concept of a hypothetical right shifting parallel-in/ parallel-out shift register that we use an overly simplified version of the data sheet details above.

- See the link to the full data sheet more more details, later in this chapter.
- **LD/SH'** controls the AND-OR multiplexer at the data input to the FF's.
- If **LD/SH'=1**, the upper four AND gates are enabled allowing application of parallel inputs D_A D_B D_C D_D to the four FF data inputs.
- Note the inverter bubble at the clock input of the four Ffs.
- This indicates that the 74LS395 clocks data on the negative going clock, which is the high to low transition.
- The four bits of data will be clocked in parallel from D_A D_B D_C D_D to Q_A Q_B Q_C Q_D at the next negative going clock.
- In this "real part", **OC'** must be low if the data needs to be available at the actual output pins as opposed to only on the internal Ffs.
- The previously loaded data may be shifted right by one bit position if **LD/SH'=0** for the succeeding negative going clock edges.
- Four clocks would shift the data entirely out of our 4-bit shift register.
- The data would be lost unless our device was cascaded from Q_D' to **SER** of another device.

Above, a data pattern is presented to inputs D_A D_B D_C D_D. The pattern is loaded to Q_A Q_B Q_C Q_D . Then it is shifted one bit to the right. The incoming data is indicated by **X**, meaning the we do no know what it is. If the input (**SER**) were grounded, for example, we would know what data (**0**) was shifted in. Also shown, is right shifting by two positions, requiring two clocks.



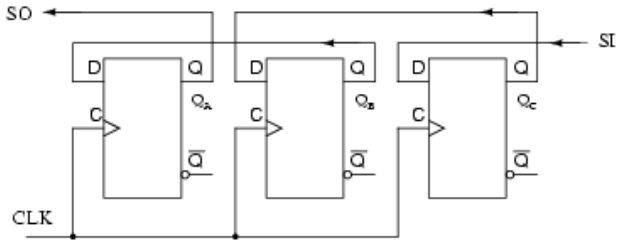
Shift right

The above figure serves as a reference for the hardware involved in right shifting of data. It is too simple to even bother with this figure, except for comparison to more complex figures to follow.

	Q_A	Q_B	Q_C
load	1	1	0
shift →	X	1	1

Load and right shift

Right shifting of data is provided above for reference to the previous right shifter.



Shift left

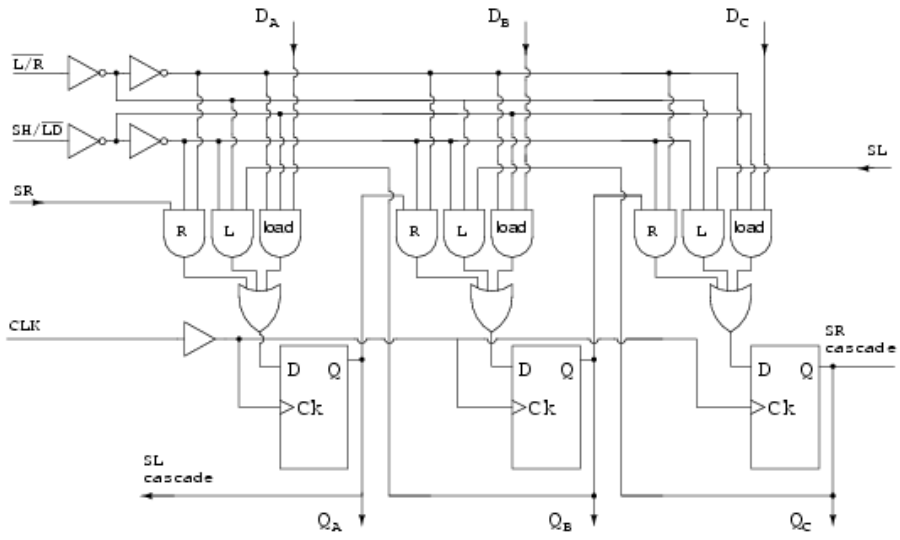
If we need to shift left, the FFs need to be rewired. Compare to the previous right shifter. Also, **SI** and **SO** have been reversed. **SI** shifts to Q_C. Q_C shifts to Q_B. Q_B shifts to Q_A. Q_A leaves on the **SO** connection, where it could cascade to

another shifter **SI**. This left shift sequence is backwards from the right shift sequence.

Above we shift the same data pattern left by one bit.
There is one problem with the "shift left" figure above. There is no market for it. Nobody manufactures a shift-left part. A "real device" which shifts one direction can be wired externally to shift the other direction. Or, should we say there is no left or right in the context of a device which shifts in only one direction. However, there is a market for a device which will shift left or right on command by a control line. Of course, left and right are valid in that context.

What we have above is a hypothetical shift register capable of shifting either direction under the control of **L'/R**. It is setup with **L'/R=1** to shift the normal direction, right. **L'/R=1** enables the multiplexer AND gates labeled **R**. This allows data to follow the path illustrated by the arrows, when a clock is applied. The connection path is the same as the"too simple" "shift right" figure above.
Data shifts in at **SR**, to Q_A, to Q_B, to Q_C, where it leaves at **SR cascade**. This pin could drive **SR** of another device to the right.
What if we change **L'/R** to **L'/R=0**?

With **L'/R=0**, the multiplexer AND gates labeled **L** are enabled, yielding a path, shown by the arrows, the same as the above "shift left" figure. Data shifts in at **SL**, to Q_C, to Q_B, to Q_A, where it leaves at **SL cascade**. This pin could drive **SL** of another device to the left.
The prime virtue of the above two figures illustrating the "shift left/ right register" is simplicity. The operation of the left right control **L'/R=0** is easy to follow. A commercial part needs the parallel data loading implied by the section title. This appears in the figure below.



Shift left/ right/ load

Now that we can shift both left and right via **L'/R**, let us add **SH/LD'**, shift/ load, and the AND gates labeled "load" to provide for parallel loading of data from inputs D_A D_B D_C. When **SH/LD'=0**, AND gates **R** and **L** are disabled, AND gates "load" are enabled to pass data D_A D_B D_C to the FF data inputs. the next clock **CLK** will clock the data to Q_A Q_B Q_C. As long as the same data is present

it will be re-loaded on succeeding clocks. However, data present for only one clock will be lost from the outputs when it is no longer present on the data inputs. One solution is to load the data on one clock, then proceed to shift on the next four clocks. This problem is remedied in the 74ALS299 by the addition of another AND gate to the multiplexer.
If **SH/LD'** is changed to **SH/LD'=1**, the AND gates labeled "load" are disabled, allowing the left/ right control **L'/R** to set the direction of shift on the **L** or **R** AND gates. Shifting is as in the previous figures.
The only thing needed to produce a viable integrated device is to add the fourth AND gate to the multiplexer as alluded for the 74ALS299. This is shown in the next section for that part.

Parallel-in/ parallel-out and universal devices

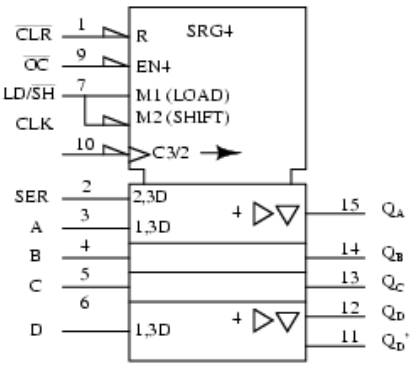
Let's take a closer look at Serial-in/ parallel-out shift registers available as integrated circuits, courtesy of Texas Instruments. For complete device data sheets, follow the links.

- SN74LS395A parallel-in/ parallel-out 4-bit shift register

[*]

- SN74ALS299 parallel-in/ parallel-out 8-bit universal shift register

[*]



SN74LS395A ANSI Symbol

We have already looked at the internal details of the SN74LS395A, see above previous figure, 74LS395 parallel-in/ parallel-out shift register with tri-state output. Directly above is the ANSI symbol for the 74LS395.

Why only 4-bits, as indicated by **SRG4** above? Having both parallel inputs, and parallel outputs, in addition to control and power pins, does not allow for any more I/O (Input/Output) bits in a 16-pin DIP (Dual Inline Package).

R indicates that the shift register stages are reset by input **CLR'** (active low-inverting half arrow at input) of the control section at the top of the symbol. **OC'**, when low, (invert arrow again) will enable (**EN4**) the four tristate output buffers (Q_A Q_B Q_C Q_D) in the data section. Load/shift' (**LD/SH'**) at pin (7) corresponds to internals **M1** (load) and **M2** (shift). Look for prefixes of **1** and **2** in the rest of the symbol to ascertain what is controlled by these.

The negative edge sensitive clock (indicated by the invert arrow at pin-10) **C3/2**has two functions. First, the **3** of **C3/2** affects any input having a prefix of **3**, say **2,3D** or **1,3D** in the data section. This would be parallel load at **A, B, C, D** attributed to **M1** and **C3** for **1,3D**. Second, **2** of **C3/2**-right-arrow indicates data clocking wherever **2** appears in a prefix (**2,3D** at pin-2). Thus we have clocking of data at **SER** into Q_A with mode **2** . The right arrow after **C3/2** accounts for shifting at internal shift register stages Q_A Q_B Q_C Q_D.

The right pointing triangles indicate buffering; the inverted triangle indicates tri-state, controlled by the **EN4**. Note, all the **4**s in the symbol associated with the **EN** are frequently omitted. Stages Q_B Q_C are understood to have the same attributes as Q_D. Q_D' cascades to the next package's **SER** to the right.

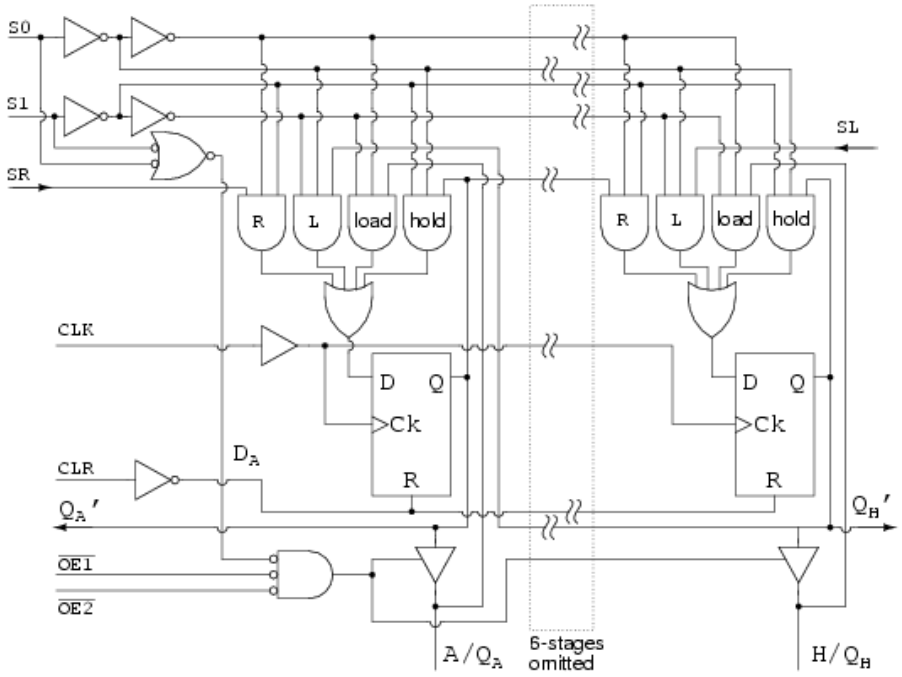
activity	mode S1	mode S0	clock	mux gate
hold	0	0	↑	hold
shift left	0	1	↑	L
shift right	1	0	↑	R
load	1	1	↑	load

S1	S0	$\overline{OE2}$	$\overline{OE1}$	tristate
X	X	X	1	disable
X	X	1	X	disable
0	0	0	0	enable
0	1	0	0	enable
1	0	0	0	enable
1	1	X	X	disable

The table above, condensed from the data '299 data sheet, summarizes the operation of the 74ALS299 universal shift/ storage register. Follow the '299 link above for full details. The Multiplexer gates **R**, **L**, **load** operate as in the previous "shift left/ right register" figures. The difference is that the mode inputs **S1** and **S0** select shift left, shift right, and load with mode set to **S1 S0 =** to **01**, **10**, and **11**respectively as shown in the table, enabling multiplexer gates **L**, **R**, and **load** respectively. See table. A minor difference is the parallel load path from the tri-state outputs. Actually the tri-state buffers are (must be) disabled by **S1 S0 = 11** to float the I/O *bus* for use as inputs. A bus is a collection of similar signals. The inputs are applied to **A**, **B** through **H** (same pins as Q_A, Q_B, through Q_H) and routed to the **load** gate in the multiplexers, and on the the **D** inputs of the FFs. Data is parallel load on a clock pulse.

The one new multiplexer gate is the AND gate labeled **hold**, enabled by **S1 S0 = 00**. The **hold** gate enables a path from the **Q** output of the FF back to the **hold** gate, to the D input of the same FF. The result is that with mode **S1 S0 = 00**, the output is continuously re-loaded with each new clock pulse. Thus, data is held. This is summarized in the table.

To read data from outputs Q_A, Q_B, through Q_H, the tri-state buffers must be enabled by **OE2', OE1' =00** and mode =**S1 S0 = 00, 01, or 10**. That is, mode is anything except **load**. See second table.



74ALS299 universal shift/ storage register with tri-state outputs

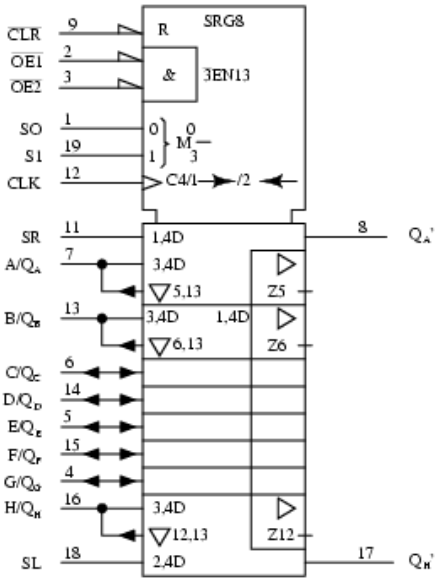
Right shift data from a package to the left, shifts in on the **SR** input. Any data shifted out to the right from stage Q_H cascades to the right via Q_H'. This output is unaffected by the tri-state buffers. The shift right sequence for **S1 S0 = 10** is:

SR > Q_A > Q_B > Q_C > Q_D > Q_E > Q_F > Q_G > Q_H (Q_H')

Left shift data from a package to the right shifts in on the **SL** input. Any data shifted out to the left from stage Q_A cascades to the left via Q_A', also unaffected by the tri-state buffers. The shift left sequence for **S1 S0 = 01** is:

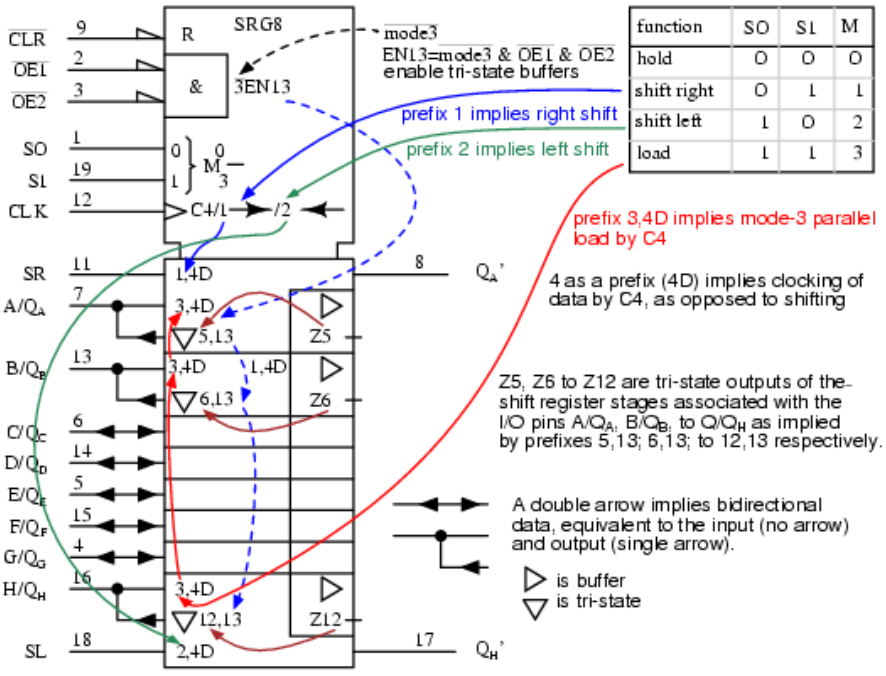
(Q_A') Q_A < Q_B < Q_C < Q_D < Q_E < Q_F < Q_G < Q_H (Q_{SL}')

Shifting may take place with the tri-state buffers disabled by one of **OE2'** or **OE1' = 1**. Though, the register contents outputs will not be accessible. See table.



SN74ALS299 ANSI Symbol

The "clean" ANSI symbol for the SN74ALS299 parallel-in/ parallel-out 8-bit universal shift register with tri-state output is shown for reference above.

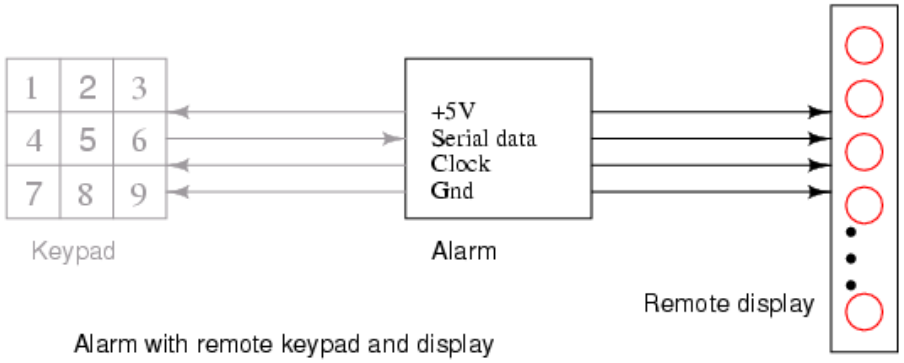


SN74ALS299 ANSI Symbol, annotated

The annotated version of the ANSI symbol is shown to clarify the terminology contained therein. Note that the ANSI mode (S0 S1) is reversed from the order (S1 S0) used in the previous table. That reverses the decimal mode numbers (1 & 2). In any event, we are in complete agreement with the official data sheet, copying this inconsistency.

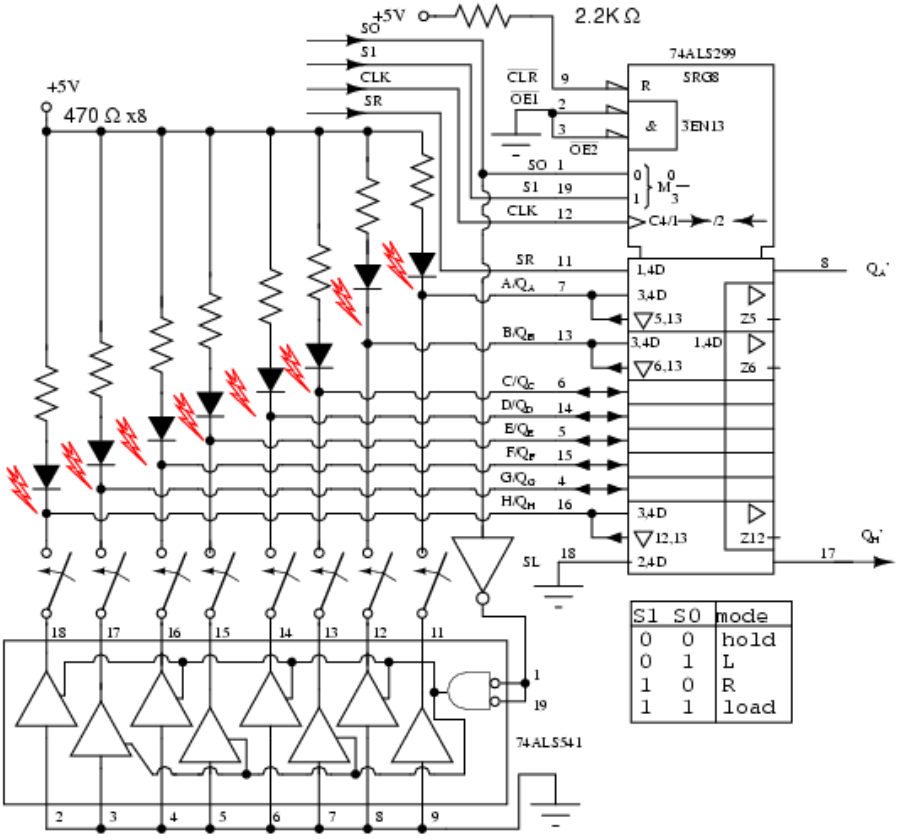
Practical applications

The Alarm with remote keypad block diagram is repeated below. Previously, we built the keypad reader and the remote display as separate units. Now we will combine both the keypad and display into a single unit using a universal shift register. Though separate in the diagram, the Keypad and Display are both contained within the same remote enclosure.



Alarm with remote keypad and display

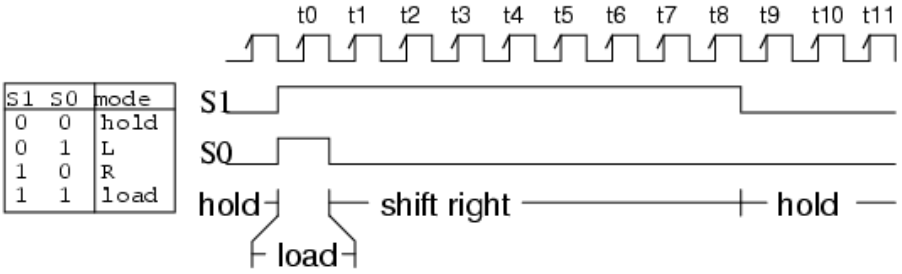
We will parallel load the keyboard data into the shift register on a single clock pulse, then shift it out to the main alarm box. At the same time , we will shift LED data from the main alarm to the remote shift register to illuminate the LEDs. We will be simultaneously shifting keyboard data out and LED data into the shift register.



74ALS299 universal shift register reads switches, drives LEDs

Eight LEDs and current limiting resistors are connected to the eight I/O pins of the 74ALS299 universal shift register. The LEDS can only be driven during Mode 3 with **S1=0 S0=0**. The **OE1'** and **OE2'** tristate enables are grounded to permenantly enable the tristate outputs during modes **0, 1, 2**. That will cause the LEDS to light (flicker) during shifting. If this were a problem the **EN1'** and **EN2'** could be ungrounded and paralleled with **S1** and **S0** respectively to only enable the tristate buffers and light the LEDS during hold, mode **3**. Let's keep it simple for this example.

During parallel loading, **S0=1** inverted to a 0, enables the octal tristate buffers to ground the switch wipers. The upper, open, switch contacts are pulled up to logic high by the resister-LED combination at the eight inputs. Any switch closure will short the input low. We parallel load the switch data into the **'299** at clock **t0** when both **S0** and **S1** are high. See waveforms below.



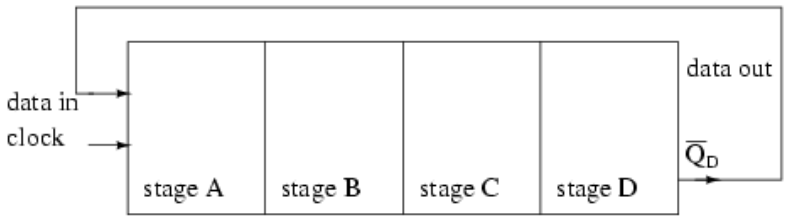
Load (t0) & shift (t1-t8) switches out of Q_H', shift LED data into SR

Once **S0** goes low, eight clocks (**t0** to**t8**) shift switch closure data out of the **'299** via the Q_h**'** pin. At the same time, new LED data is shifted in at **SR** of the **299** by the same eight clocks. The LED data replaces the switch closure data as shifting proceeds.

After the 8th shift clock, **t8**, **S1** goes low to yield hold mode (**S1 S0 = 00**). The data in the shift register remains the same even if there are more clocks, for example, **T9**, **t10**, etc. Where do the waveforms come from? They could be generated by a microprocessor if the clock rate were not over 100 kHz, in which case, it would be inconvenient to generate any clocks after **t8**. If the clock was in the megahertz range, the clock would run continuously. The clock, **S1** and **S0** would be generated by digital logic, not shown here.

Ring counters

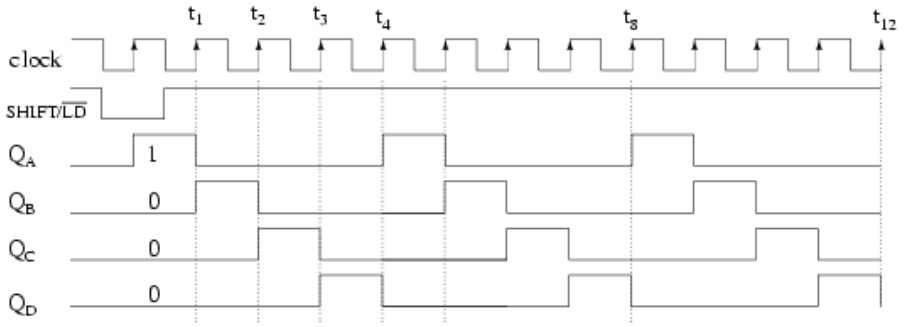
If the output of a shift register is fed back to the input. a ring counter results. The data pattern contained within the shift register will recirculate as long as clock pulses are applied. For example, the data pattern will repeat every four clock pulses in the figure below. However, we must load a data pattern. All **0**'s or all **1**'s doesn't count. Is a continuous logic level from such a condition useful?



Ring Counter, shift register output fed back to input

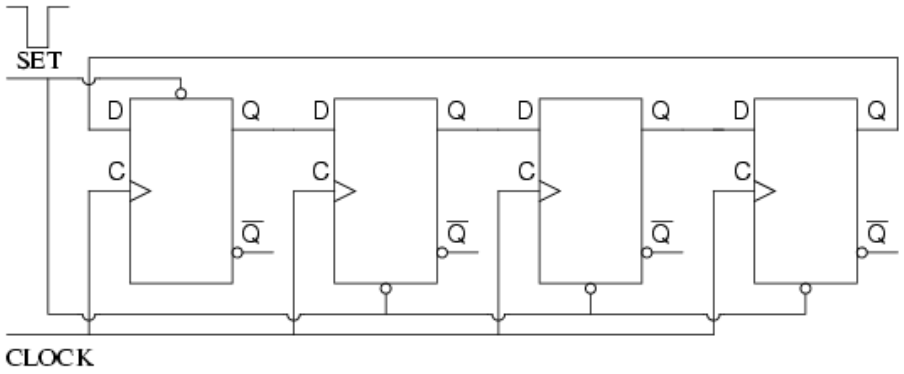
We make provisions for loading data into the parallel-in/ serial-out shift register configured as a ring counter below. Any random pattern may be loaded. The most generally useful pattern is a single **1**.

Loading binary **1000** into the ring counter, above, prior to shifting yields a viewable pattern. The data pattern for a single stage repeats every four clock pulses in our 4-stage example. The waveforms for all four stages look the same, except for the one clock time delay from one stage to the next. See figure below.



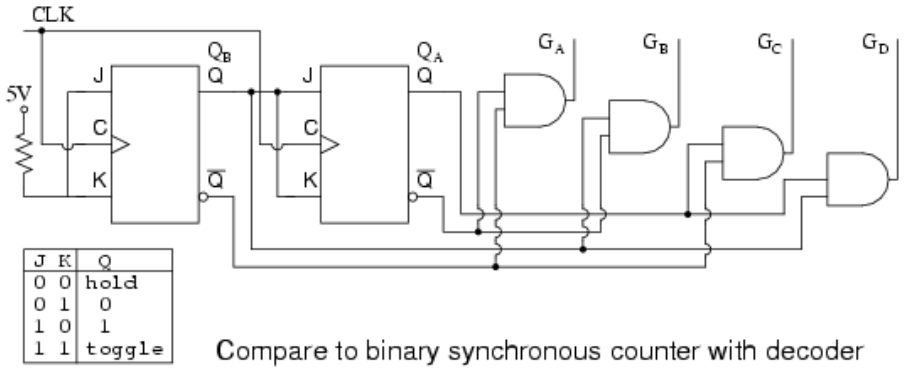
Load 1000 into 4-stage ring counter and shift

The circuit above is a divide by **4** counter. Comparing the clock input to any one of the outputs, shows a frequency ratio of 4:1. How may stages would we need for a divide by 10 ring counter? Ten stages would recirculate the **1** every **10** clock pulses.



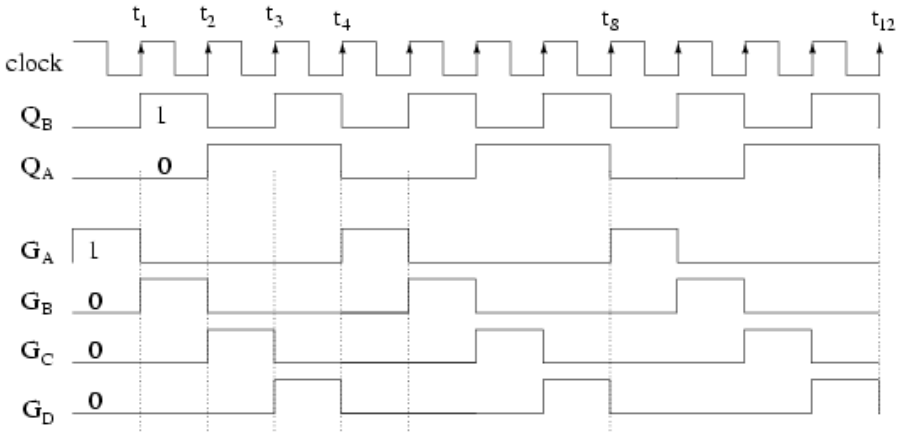
Set one stage, clear three stages

An alternate method of initializing the ring counter to **1000** is shown above. The shift waveforms are identical to those above, repeating every fourth clock pulse. The requirement for initialization is a disadvantage of the ring counter over a conventional counter. At a minimum, it must be initialized at power-up since there is no way to predict what state flip-flops will power up in. In theory, initialization should never be required again. In actual practice, the flip-flops could eventually be corrupted by noise, destroying the data pattern. A "self correcting" counter, like a conventional synchronous binary counter would be more reliable.



Compare to binary synchronous counter with decoder

The above binary synchronous counter needs only two stages, but requires decoder gates. The ring counter had more stages, but was self decoding, saving the decode gates above. Another disadvantage of the ring counter is that it is not "self starting". If we need the decoded outputs, the ring counter looks attractive, in particular, if most of the logic is in a single shift register package. If not, the conventional binary counter is less complex without the decoder.

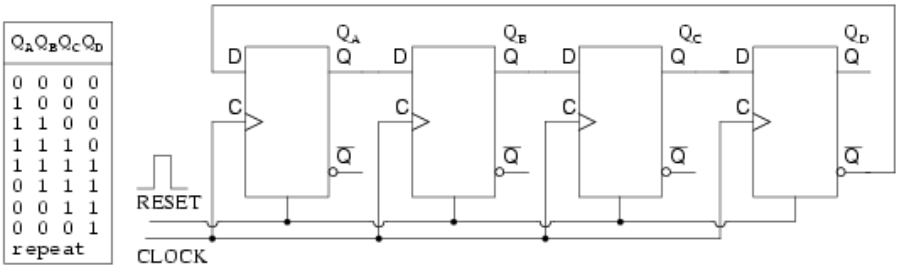


Compare to binary synchronous counter with decode. waveforms

The waveforms decoded from the synchronous binary counter are identical to the previous ring counter waveforms. The counter sequence is (Q_A Q_B) = (**00 01 10 11**).

Johnson counters

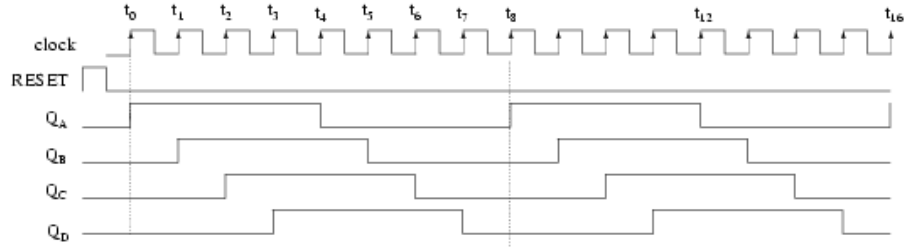
The *switch-tail ring counter*, also know as the *Johnson counter*, overcomes some of the limitations of the ring counter. Like a ring counter a Johnson counter is a shift register fed back on its' self. It requires half the stages of a comparable ring counter for a given division ratio. If the complement output of a ring counter is fed back to the input instead of the true output, a Johnson counter results. The difference between a ring counter and a Johnson counter is which output of the last stage is fed back (Q or Q'). Carefully compare the feedback connection below to the previous ring counter.



Johnson counter (note the $\overline{Q_D}$ to D_A feedback connection)

This "reversed" feedback connection has a profound effect upon the behavior of the otherwise similar circuits. Recirculating a single **1** around a ring counter divides the input clock by a factor equal to the number of stages. Whereas, a Johnson counter divides by a factor equal to twice the number of stages. For example, a 4-stage ring counter divides by **4**. A 4-stage Johnson counter divides by **8**.

Start a Johnson counter by clearing all stages to **0**s before the first clock. This is often done at power-up time. Referring to the figure below, the first clock shifts three **0**s from (Q_A Q_B Q_C) to the right into (Q_B Q_C Q_D). The **1** at Q_D' (the complement of Q) is shifted back into Q_A. Thus, we start shifting **1**s to the right, replacing the **0**s. Where a ring counter recirculated a single **1**, the 4-stage Johnson counter recirculates four **0**s then four **1**s for an 8-bit pattern, then repeats.

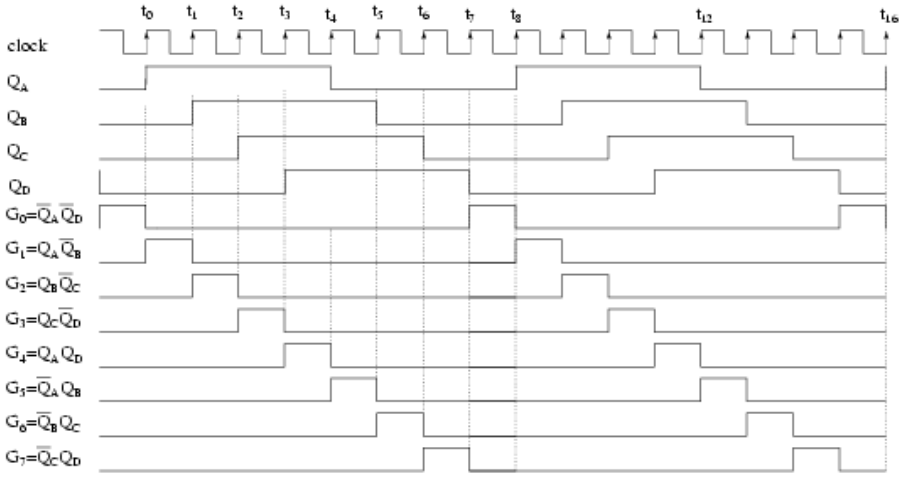


Four stage Johnson counter waveforms

The above waveforms illustrates that multi-phase square waves are generated by a Johnson counter. The 4-stage unit above generates four overlapping phases of 50% duty cycle. How many stages would be required to generate a set of three phase waveforms? For example, a three stage Johnson counter, driven by a 360 Hertz clock would generate three 120^o phased square waves at 60 Hertz.

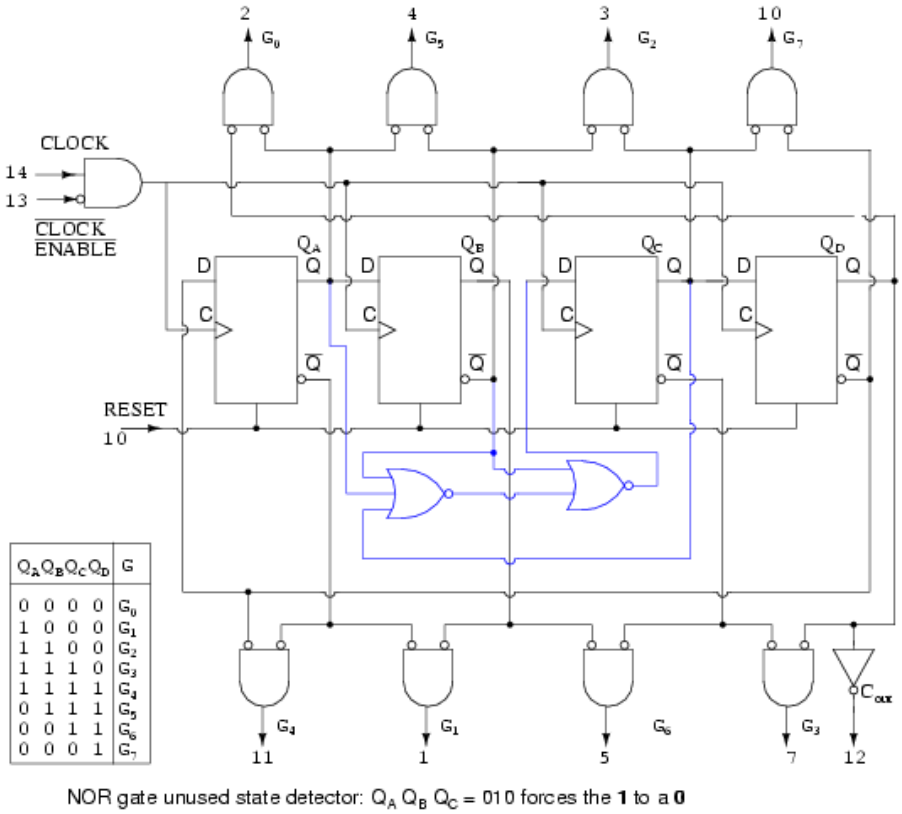
The outputs of the flop-flops in a Johnson counter are easy to decode to a single state. Below for example, the eight states of a 4-stage Johnson counter are decoded by no more than a two input gate for each of the states. In our example, eight of the two input gates decode the states for our example Johnson counter.

No matter how long the Johnson counter, only 2-input decoder gates are needed. Note, we could have used uninverted inputs to the **AND** gates by changing the gate inputs from true to inverted at the FFs, **Q** to **Q'**, (and vice versa). However, we are trying to make the diagram above match the data sheet for the CD4022B, as closely as practical.



Four stage (8-state) Johnson counter decoder waveforms

Above, our four phased square waves Q_A to Q_D are decoded to eight signals (G_0 to G_7) active during one clock period out of a complete 8-clock cycle. For example, G_0 is active high when both Q_A and Q_D are low. Thus, pairs of the various register outputs define each of the eight states of our Johnson counter example.



$Q_A Q_B Q_C Q_D$	G
0 0 0 0	G_0
1 0 0 0	G_1
1 1 0 0	G_2
1 1 1 0	G_3
1 1 1 1	G_4
0 1 1 1	G_5
0 0 1 1	G_6
0 0 0 1	G_7

NOR gate unused state detector: Q_A Q_B Q_C = 010 forces the **1** to a **0**

CD4022B modulo-8 Johnson counter with unused state detector

Above is the more complete internal diagram of the CD4022B Johnson counter. See the manufacturers' data sheet for minor details omitted. The major new addition to the diagram as compared to previous figures is the *disallowed state detector* composed of the two **NOR** gates. Take a look at the inset state table. There are 8-permissible states as listed in the table. Since our shifter has four flip-flops, there are a total of 16-states, of which there are 8-disallowed states. That would be the ones not listed in the table.

In theory, we will not get into any of the disallowed states as long as the shift register is **RESET** before first use. However, in the "real world" after many days of continuous operation due to unforeseen noise, power line disturbances, near lightning strikes, etc, the Johnson counter could get into one of the disallowed states. For high reliability applications, we need to plan for this slim possibility. More serious is the case where the circuit is not cleared at power-up. In this case there is no way to know which of the 16-states the circuit will power up in. Once in a disallowed state, the Johnson counter will not return to any of the permissible states without intervention. That is the purpose of the **NOR** gates.

Examine the table for the sequence (Q_A Q_B Q_C) = (**010**). Nowhere does this sequence appear in the table of allowed states. Therefore (**010**) is disallowed. It should never occur. If it does, the Johnson counter is in a disallowed state, which it needs to exit to any allowed state. Suppose that (Q_A Q_B Q_C) = (**010**). The second **NOR** gate will replace Q_B = **1** with a **0** at the **D** input to FF Q_C. In other words, the offending **010** is replaced by **000**. And **000**, which does appear in the table, will be shifted right. There are may triple-0 sequences in the table. This is how the **NOR** gates get the Johnson counter out of a disallowed state to an allowed state.

Not all disallowed states contain a **010** sequence. However, after a few clocks, this sequence will appear so that any disallowed states will eventually be escaped. If the circuit is powered-up without a **RESET**, the outputs will be unpredictable for a few clocks until an allowed state is reached. If this is a problem for a particular application, be sure to **RESET** on power-up.

Johnson counter devices

A pair of integrated circuit Johnson counter devices with the output states decoded is available. We have already looked at the CD4017 internal logic in the discussion of Johnson counters. The 4000 series devices can operate from 3V to 15V power supplies. The the 74HC' part, designed for a TTL compatiblity, can operate from a 2V to 6V supply, count faster, and has greater output drive capability. For complete device data sheets, follow the links.

- CD4017 Johnson counter with 10 decoded outputs

CD4022 Johnson counter with 8 decoded outputs

[*]

- 74HC4017 Johnson counter, 10 decoded outputs

[*]

The ANSI symbols for the *modulo*-10 (divide by 10) and modulo-8 Johnson counters are shown above. The symbol takes on the characteristics of a counter rather than a shift register derivative, which it is. Waveforms for the CD4022 modulo-8 and operation were shown previously. The CD4017B/ 74HC4017 decade counter is a 5-stage Johnson counter with ten decoded outputs. The operation and waveforms are similar to the CD4017. In fact, the CD4017 and CD4022 are both detailed on the same data sheet. See above links. The 74HC4017 is a more modern version of the decade counter.

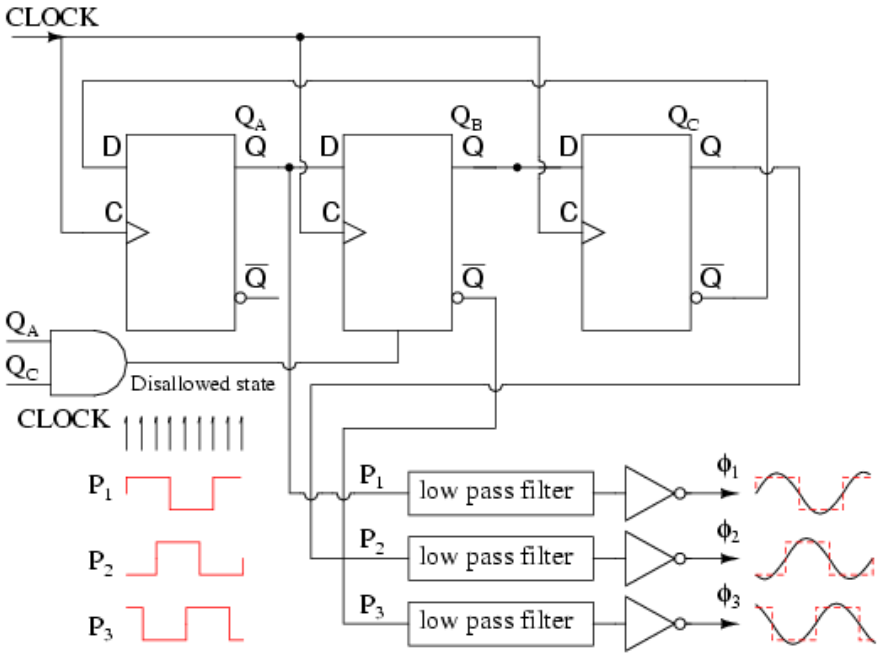
These devices are used where decoded outputs are needed instead of the binary or BCD (Binary Coded Decimal) outputs found on normal counters. By decoded, we mean one line out of the ten lines is active at a time for the '4017 in place of the four bit BCD code out of conventional counters. See previous waveforms for 1-of-8 decoding for the '4022 Octal Johnson counter.

Practical applications

The above Johnson counter shifts a lighted LED each fifth of a second around the ring of ten. Note that the 74HC4017 is used instead of the '40017 because the former part has more current drive capability. From the data sheet, (at the link

above) operating at V_{CC}= 5V, the V_{OH}= 4.6V at 4ma. In other words, the outputs can supply 4 ma at 4.6 V to drive the LEDs. Keep in mind that LEDs are normally driven with 10 to 20 ma of current. Though, they are visible down to 1 ma. This simple circuit illustrates an application of the 'HC4017. Need a bright display for an exhibit? Then, use inverting buffers to drive the cathodes of the LEDs pulled up to the power supply by lower value anode resistors.

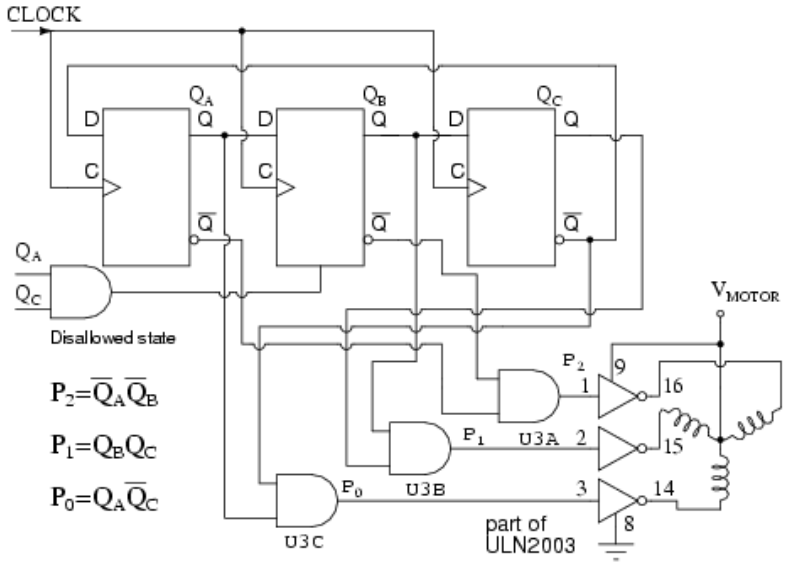
The 555 timer, serving as an astable multivibrator, generates a clock frequency determined by R_1 R_2 C_1. This drives the 74HC4017 a step per clock as indicated by a single LED illuminated on the ring. Note, if the 555 does not reliably drive the clock pin of the '4015, run it through a single buffer stage between the 555 and the '4017. A variable R_2 could change the step rate. The value of decoupling capacitor C_2 is not critical. A similar capacitor should be applied across the power and ground pins of the '4017.



Three phase square/ sine wave generator.

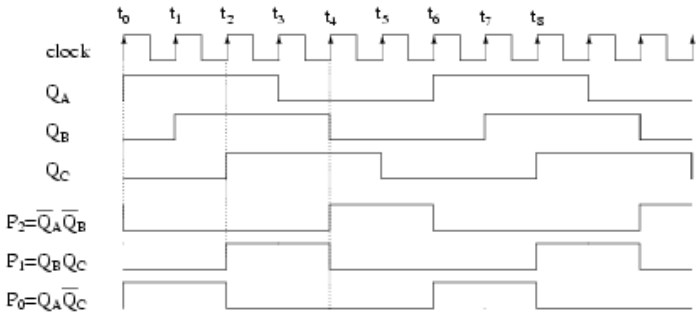
The Johnson counter above generates 3-phase square waves, phased 60° apart with respect to (Q_A Q_B Q_C). However, we need 120° phased waveforms of power applications (see Volume II, AC). Choosing P_1=Q_A P_2=Q_C P_3=Q_B' yields the 120° phasing desired. See figure below. If these (P_1 P_2 P_3) are low-pass filtered to sine waves and amplified, this could be the beginnings of a 3-phase power supply. For example, do you need to drive a small 3-phase 400 Hz aircraft motor? Then, feed 6x 400Hz to the above circuit **CLOCK**. Note that all these waveforms are 50% duty cycle.

The circuit below produces 3-phase nonoverlapping, less than 50% duty cycle, waveforms for driving 3-phase stepper motors.

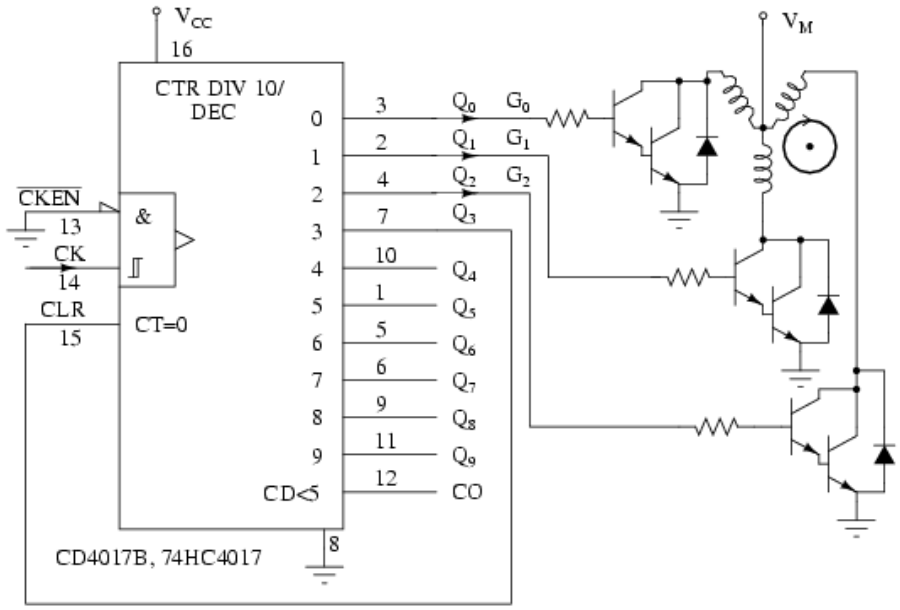


3-stage (6-state) Johnson counter decoded for 3-ϕ stepper motor.

Above we decode the overlapping outputs Q_A Q_B Q_C to non-overlapping outputs P_0 P_1 P_2 as shown below. These waveforms drive a 3-phase stepper motor after suitable amplification from the milliamp level to the fractional amp level using the ULN2003 drivers shown above, or the discrete component Darlington pair driver shown in the circuit which follow. Not counting the motor driver, this circuit requires three IC (Integrated Circuit) packages: two dual type "D" FF packages and a quad NAND gate.



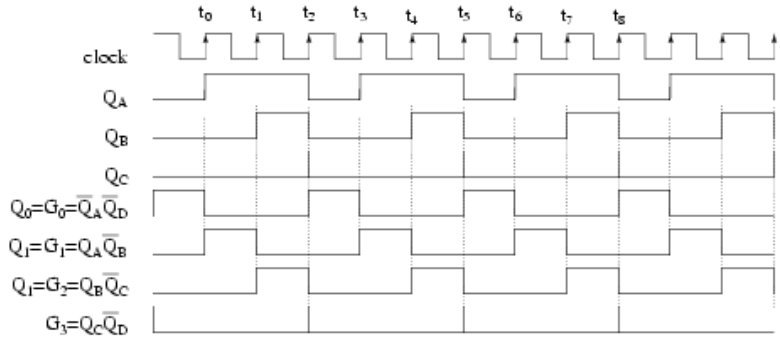
3-stage Johnson counter generates 3-Ø stepper waveform.



Johnson sequence terminated early by reset at Q_3, which is high.
for nano seconds

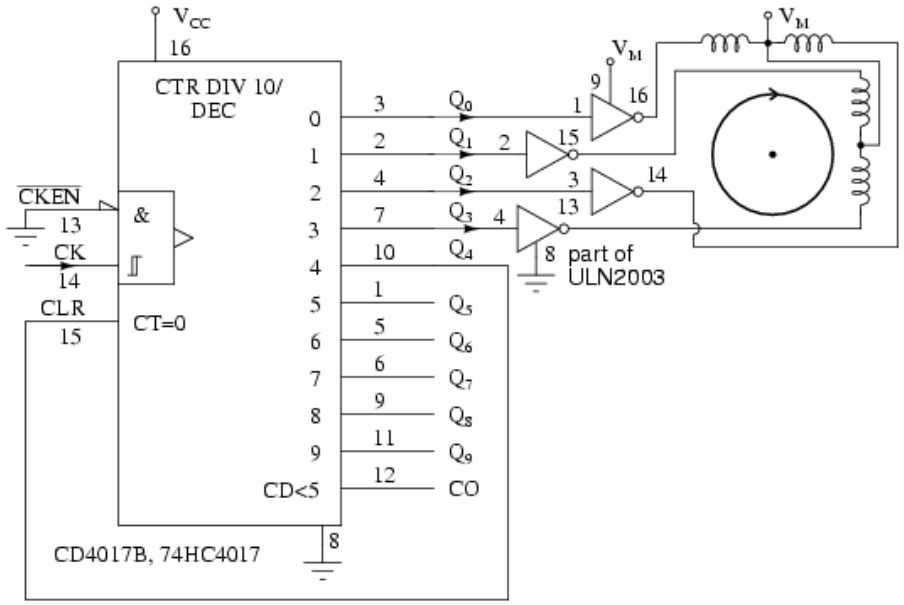
A single CD4017, above, generates the required 3-phase stepper waveforms in the circuit above by clearing the Johnson counter at count **3**. Count **3** persists for less than a microsecond before it clears its' self. The other counts (**Q_0=G_0 Q_1=G_1 Q_2=G_2**) remain for a full clock period each.

The Darlington bipolar transistor drivers shown above are a substitute for the internal circuitry of the ULN2003. The design of drivers is beyond the scope of this digital electronics chapter. Either driver may be used with either waveform generator circuit.



CD4017B 5-stage (10-state) Johnson counter resetting
at Q_C Q_B Q_A = 100 generates 3–ϕ stepper waveform.

The above waceforms make the most sense in the context of the internal logic of the CD4017 shown earlier in this section. Though, the **AND** gating equations for the internal decoder are shown. The signals **Q_A Q_B Q_C** are Johnson counter direct shift register outputs not available on pin-outs. The **Q_D** waveform shows resetting of the **'4017** every three clocks. **Q_0 Q_1 Q_2**, etc. are decoded outputs which actually are available at output pins.



Johnson counter drives unipolar stepper motor.

Above we generate waveforms for driving a *unipolar stepper motor*, which only requires one polarity of driving signal. That is, we do not have to reverse the polarity of the drive to the windings. This simplifies the power driver between the '4017 and the motor. Darlington pairs from a prior diagram may be substituted for the ULN3003.

Once again, the CD4017B generates the required waveforms with a reset after the teminal count. The decoded outputs Q_0 Q_1 Q_2 Q_3 sucessively drive the stepper motor windings, with Q_4 reseting the counter at the end of each group of four pulses.

Lessons In Electric Circuits -- Volume IV

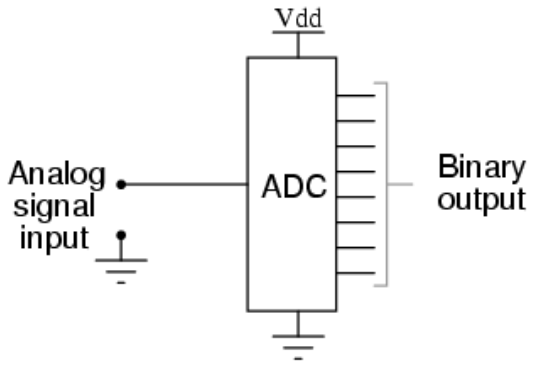
Chapter 13

DIGITAL-ANALOG CONVERSION

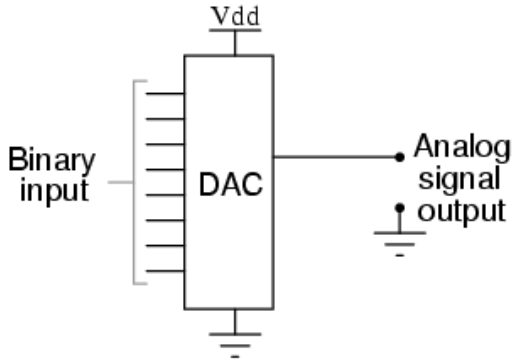
Introduction

Connecting digital circuitry to sensor devices is simple if the sensor devices are inherently digital themselves. Switches, relays, and encoders are easily interfaced with gate circuits due to the on/off nature of their signals. However, when analog devices are involved, interfacing becomes much more complex. What is needed is a way to electronically translate analog signals into digital (binary) quantities, and vice versa. An *analog-to-digital converter*, or ADC, performs the former task while a *digital-to-analog converter*, or DAC, performs the latter.

An ADC inputs an analog electrical signal such as voltage or current and outputs a binary number. In block diagram form, it can be represented as such:



A DAC, on the other hand, inputs a binary number and outputs an analog voltage or current signal. In block diagram form, it looks like this:



Together, they are often used in digital systems to provide complete interface with analog sensors and output devices for control systems such as those used in automotive engine controls:

It is much easier to convert a digital signal into an analog signal than it is to do the reverse. Therefore, we will begin with DAC circuitry and then move to ADC circuitry.

The R/2^{n}R DAC

This DAC circuit, otherwise known as the *binary-weighted-input* DAC, is a variation on the inverting summer op-amp circuit. If you recall, the classic inverting summer circuit is an operational amplifier using negative feedback for controlled gain, with several voltage inputs and one voltage output. The output voltage is the inverted (opposite polarity) sum of all input voltages:

For a simple inverting summer circuit, all resistors must be of equal value. If any of the input resistors were different, the input voltages would have different degrees of effect on the output, and the output voltage would not be a true sum. Let's consider, however, intentionally setting the input resistors at different values. Suppose we were to set the input resistor values at multiple powers of two: R, 2R, and 4R, instead of all the same value R:

Starting from V_1 and going through V_3, this would give each input voltage exactly half the effect on the output as the voltage before it. In other words, input voltage V_1 has a 1:1 effect on the output voltage (gain of 1), while input voltage V_2 has half that much effect on the output (a gain of 1/2), and V_3 half of that (a gain of 1/4). These ratios are were not arbitrarily chosen: they are the same ratios corresponding to place weights in the binary numeration system. If we drive the inputs of this circuit with digital gates so that each input is either 0 volts or full supply voltage, the output voltage will be an analog representation of the binary value of these three bits.

If we chart the output voltages for all eight combinations of binary bits (000 through 111) input to this circuit, we will get the following progression of voltages:

Binary	Output voltage
000	0.00 V
001	-1.25 V
010	-2.50 V
011	-3.75 V

Binary	Output voltage
100	-5.00 V
101	-6.25 V
110	-7.50 V
111	-8.75 V

Note that with each step in the binary count sequence, there results a 1.25 volt change in the output. This circuit is very easy to simulate using SPICE. In the following simulation, I set up the DAC circuit with a binary input of 110 (note the first node numbers for resistors R_1, R_2, and R_3: a node number of "1" connects it to the positive side of a 5 volt battery, and a node number of "0" connects it to ground). The output voltage appears on node 6 in the simulation:

```
binary-weighted dac
 v1 1 0 dc 5
 rbogus 1 0 99k
 r1 1 5 1k
 r2 1 5 2k
 r3 0 5 4k
 rfeedbk 5 6 1k
 e1 6 0 5 0 999k
 .end

node  voltage      node  voltage      node  voltage
(1)   5.0000       (5)   0.0000       (6)  -7.5000
```

We can adjust resistors values in this circuit to obtain output voltages directly corresponding to the binary input. For example, by making the feedback resistor 800 instead of 1 k , the DAC will output -1 volt for the binary input 001, -4 volts for the binary input 100, -7 volts for the binary input 111, and so on.

(with feedback resistor set at 800 ohms)

Binary	Output voltage
000	0.00 V
001	-1.00 V
010	-2.00 V
011	-3.00 V
100	-4.00 V

101	-5.00 V
110	-6.00 V
111	-7.00 V

If we wish to expand the resolution of this DAC (add more bits to the input), all we need to do is add more input resistors, holding to the same power-of-two sequence of values:

It should be noted that all logic gates must output exactly the same voltages when in the "high" state. If one gate is outputting +5.02 volts for a "high" while another is outputting only +4.86 volts, the analog output of the DAC will be adversely affected. Likewise, all "low" voltage levels should be identical between gates, ideally 0.00 volts exactly. It is recommended that CMOS output gates are used, and that input/feedback resistor values are chosen so as to minimize the amount of current each gate has to source or sink.

The R/2R DAC

An alternative to the binary-weighted-input DAC is the so-called R/2R DAC, which uses fewer unique resistor values. A disadvantage of the former DAC design was its requirement of several different precise input resistor values: one unique value per binary input bit. Manufacture may be simplified if there are fewer different resistor values to purchase, stock, and sort prior to assembly.

Of course, we could take our last DAC circuit and modify it to use a single input resistance value, by connecting multiple resistors together in series:

Unfortunately, this approach merely substitutes one type of complexity for another: volume of components over diversity of component values. There is, however, a more efficient design methodology.

By constructing a different kind of resistor network on the input of our summing circuit, we can achieve the same kind of binary weighting with only two kinds of resistor values, and with only a modest increase in resistor count. This "ladder" network looks like this:

Mathematically analyzing this ladder network is a bit more complex than for the previous circuit, where each input resistor provided an easily-calculated gain for that bit. For those who are interested in pursuing the intricacies of this circuit further, you may opt to use Thevenin's theorem for each binary input (remember to consider the

effects of the *virtual ground*), and/or use a simulation program like SPICE to determine circuit response. Either way, you should obtain the following table of figures:

Binary	Output voltage
000	0.00 V
001	-1.25 V
010	-2.50 V
011	-3.75 V
100	-5.00 V
101	-6.25 V
110	-7.50 V
111	-8.75 V

As was the case with the binary-weighted DAC design, we can modify the value of the feedback resistor to obtain any "span" desired. For example, if we're using +5 volts for a "high" voltage level and 0 volts for a "low" voltage level, we can obtain an analog output directly corresponding to the binary input (011 = -3 volts, 101 = -5 volts, 111 = -7 volts, etc.) by using a feedback resistance with a value of 1.6R instead of 2R.

Flash ADC

Also called the *parallel* A/D converter, this circuit is the simplest to understand. It is formed of a series of comparators, each one comparing the input signal to a unique reference voltage. The comparator outputs connect to the inputs of a priority encoder circuit, which then produces a binary output. The following illustration shows a 3-bit flash ADC circuit:

V_{ref} is a stable reference voltage provided by a precision voltage regulator as part of the converter circuit, not shown in the schematic. As the analog input voltage exceeds the reference voltage at each comparator, the comparator outputs will sequentially saturate to a high state. The priority encoder generates a binary number based on the highest-order active input, ignoring all other active inputs.

When operated, the flash ADC produces an output that looks something like this:

For this particular application, a regular priority encoder with all its inherent complexity isn't necessary. Due to the nature of the sequential comparator output states (each comparator saturating "high" in sequence from lowest to highest), the same "highest-order-input selection" effect may be realized through a set of Exclusive-OR gates, allowing the use of a simpler, non-priority encoder:

And, of course, the encoder circuit itself can be made from a matrix of diodes, demonstrating just how simply this converter design may be constructed:

Not only is the flash converter the simplest in terms of operational theory, but it is the most efficient of the ADC technologies in terms of speed, being limited only in comparator and gate propagation delays. Unfortunately, it is the most component-intensive for any given number of output bits. This three-bit flash ADC requires seven comparators. A four-bit version would require 15 comparators. With each additional output bit, the number of required comparators doubles. Considering that eight bits is generally considered the minimum necessary for any practical ADC (255 comparators needed!), the flash methodology quickly shows its weakness.

An additional advantage of the flash converter, often overlooked, is the ability for it to produce a non-linear output. With equal-value resistors in the reference voltage divider network, each successive binary count represents the same amount of analog signal increase, providing a proportional response. For special applications, however, the resistor values in the divider network may be made non-equal. This gives the ADC a custom, nonlinear response to the analog input signal. No other ADC design is able to grant this signal-conditioning behavior with just a few component value changes.

Speed:

Flash, tracking, successive approximation, single-slope integrating & counter, dual-slope integrating.

Lessons In Electric Circuits -- Volume IV

Chapter 15

DIGITAL STORAGE (MEMORY)

Why digital?

Although many textbooks provide good introductions to digital memory technology, I intend to make this chapter unique in presenting both past and present technologies to some degree of detail. While many of these memory designs are obsolete, their foundational principles are still quite interesting and educational, and may even find re-application in the memory technologies of the future.

The basic goal of digital memory is to provide a means to store and access binary data: sequences of 1's and 0's. The digital storage of information holds advantages over analog techniques much the same as digital communication of information holds advantages over analog communication. This is not to say that digital data storage is unequivocally superior to analog, but it does address some of the more common problems associated with analog techniques and thus finds immense popularity in both consumer and industrial applications. Digital data storage also complements digital computation technology well, and thus finds natural application in the world of computers.

The most evident advantage of digital data storage is the resistance to corruption. Suppose that we were going to store a piece of data regarding the magnitude of a voltage signal by means of magnetizing a small chunk of magnetic material. Since many magnetic materials retain their strength of magnetization very well over time, this would be a logical media candidate for long-term storage of this particular data (in fact, this is precisely how audio and video tape technology works: thin plastic tape is impregnated with particles of iron-oxide material, which can be magnetized or demagnetized via the application of a magnetic field from an electromagnet coil. The data is then retrieved from the tape by moving the magnetized tape past another coil of wire, the magnetized spots on the tape inducing voltage in that coil, reproducing the voltage waveform initially used to magnetize the tape).

If we represent an analog signal by the strength of magnetization on spots of the tape, the storage of data on the tape will be susceptible to the smallest degree of degradation of that magnetization. As the tape ages and the magnetization fades, the

analog signal magnitude represented on the tape will appear to be less than what it was when we first recorded the data. Also, if any spurious magnetic fields happen to alter the magnetization on the tape, even if its only by a small amount, that altering of field strength will be interpreted upon re-play as an altering (or corruption) of the signal that was recorded. Since analog signals have infinite resolution, the smallest degree of change will have an impact on the integrity of the data storage.

If we were to use that same tape and store the data in binary digital form, however, the strength of magnetization on the tape would fall into two discrete levels: "high" and "low," with no valid in-between states. As the tape aged or was exposed to spurious magnetic fields, those same locations on the tape would experience slight alteration of magnetic field strength, but unless the alterations were *extreme*, no data corruption would occur upon re-play of the tape. By reducing the resolution of the signal impressed upon the magnetic tape, we've gained significant immunity to the kind of degradation and "noise" typically plaguing stored analog data. On the other hand, our data resolution would be limited to the scanning rate and the number of bits output by the A/D converter which interpreted the original analog signal, so the reproduction wouldn't necessarily be "better" than with analog, merely more rugged. With the advanced technology of modern A/D's, though, the tradeoff is acceptable for most applications.

Also, by encoding different types of data into specific binary number schemes, digital storage allows us to archive a wide variety of information that is often difficult to encode in analog form. Text, for example, is represented quite easily with the binary ASCII code, seven bits for each character, including punctuation marks, spaces, and carriage returns. A wider range of text is encoded using the Unicode standard, in like manner. Any kind of numerical data can be represented using binary notation on digital media, and any kind of information that can be encoded in numerical form (which almost any kind can!) is storable, too. Techniques such as parity and checksum error detection can be employed to further guard against data corruption, in ways that analog does not lend itself to.

Digital memory terms and concepts

When we store information in some kind of circuit or device, we not only need some way to store and retrieve it, but also to locate precisely *where* in the device that it is. Most, if not all, memory devices can be thought of as a series of mail boxes, folders in a file cabinet, or some other metaphor where information can be located in a variety of places. When we refer to the actual information being stored in the memory device, we usually refer to it as the *data*. The location of this data within the storage device is typically called the *address*, in a manner reminiscent of the postal service.

With some types of memory devices, the address in which certain data is stored can be called up by means of parallel data lines in a digital circuit (we'll discuss this in more detail later in this lesson). With other types of devices, data is addressed in terms of an actual physical location on the surface of some type of media (the *tracks* and *sectors* of circular computer disks, for instance). However, some memory devices such as magnetic tapes have a one-dimensional type of data addressing: if you want to play your favorite song in the middle of a cassette tape album, you have to fast-forward to that spot in the tape, arriving at the proper spot by

means of trial-and-error, judging the approximate area by means of a counter that keeps track of tape position, and/or by the amount of time it takes to get there from the beginning of the tape. The access of data from a storage device falls roughly into two categories: *random access* and *sequential access*. Random access means that you can quickly and precisely address a specific data location within the device, and non-random simply means that you cannot. A vinyl record platter is an example of a random-access device: to skip to any song, you just position the stylus arm at whatever location on the record that you want (compact audio disks so the same thing, only they do it automatically for you). Cassette tape, on the other hand, is sequential. You have to wait to go past the other songs in sequence before you can access or address the song that you want to skip to.

The process of storing a piece of data to a memory device is called *writing*, and the process of retrieving data is called *reading*. Memory devices allowing both reading and writing are equipped with a way to distinguish between the two tasks, so that no mistake is made by the user (writing new information to a device when all you wanted to do is see what was stored there). Some devices do not allow for the writing of new data, and are purchased "pre-written" from the manufacturer. Such is the case for vinyl records and compact audio disks, and this is typically referred to in the digital world as *read-only memory*, or ROM. Cassette audio and video tape, on the other hand, can be re-recorded (re-written) or purchased blank and recorded fresh by the user. This is often called *read-write memory*.

Another distinction to be made for any particular memory technology is its volatility, or data storage permanence without power. Many electronic memory devices store binary data by means of circuits that are either latched in a "high" or "low" state, and this latching effect holds only as long as electric power is maintained to those circuits. Such memory would be properly referred to as *volatile*. Storage media such as magnetized disk or tape is *nonvolatile*, because no source of power is needed to maintain data storage. This is often confusing for new students of computer technology, because the volatile electronic memory typically used for the construction of computer devices is commonly and distinctly referred to as *RAM* (**R**andom **A**ccess **M**emory). While RAM memory is typically randomly-accessed, so is virtually every other kind of memory device in the computer! What "RAM" *really* refers to is the *volatility* of the memory, and not its mode of access. Nonvolatile memory integrated circuits in personal computers are commonly (and properly) referred to as *ROM* (**R**ead-**O**nly **M**emory), but their data contents are accessed randomly, just like the volatile memory circuits!

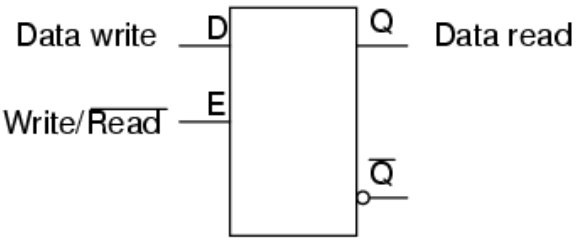
Finally, there needs to be a way to denote how much data can be stored by any particular memory device. This, fortunately for us, is very simple and straightforward: just count up the number of bits (or bytes, 1 byte = 8 bits) of total data storage space. Due to the high capacity of modern data storage devices, metric prefixes are generally affixed to the unit of bytes in order to represent storage space: 1.6 Gigabytes is equal to 1.6 billion bytes, or 12.8 billion bits, of data storage capacity. The only caveat here is to be aware of rounded numbers. Because the storage mechanisms of many random-access memory devices are typically arranged so that the number of "cells" in which bits of data can be stored appears in binary progression (powers of 2), a "one kilobyte" memory device most likely contains 1024 (2 to the power of 10) locations for data bytes rather than exactly 1000. A "64 kbyte" memory device actually holds 65,536 bytes of data (2 to the 16th power), and should probably be called a "66 Kbyte" device to be more precise. When we round numbers

in our base-10 system, we fall out of step with the round equivalents in the base-2 system.

Modern nonmechanical memory

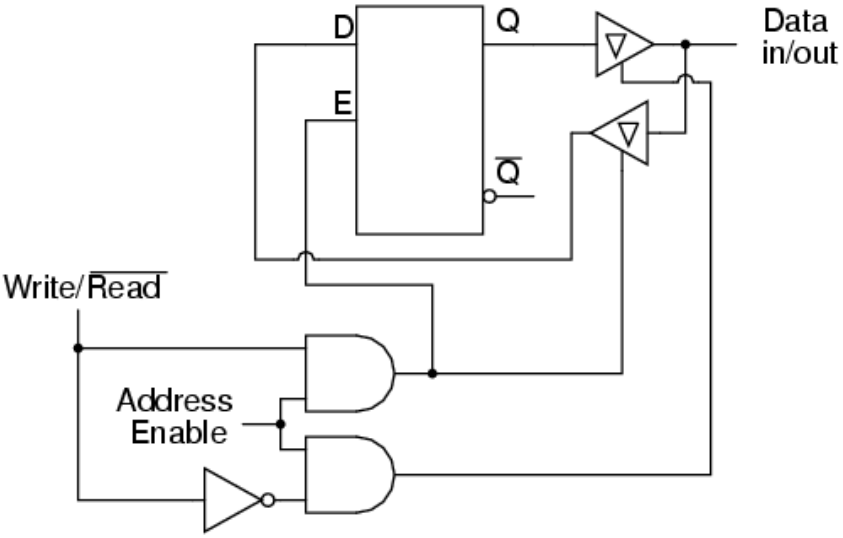
Now we can proceed to studying specific types of digital storage devices. To start, I want to explore some of the technologies which do not require any moving parts. These are not necessarily the newest technologies, as one might suspect, although they will most likely replace moving-part technologies in the future.

A very simple type of electronic memory is the bistable multivibrator. Capable of storing a single bit of data, it is volatile (requiring power to maintain its memory) and very fast. The D-latch is probably the simplest implementation of a bistable multivibrator for memory usage, the D input serving as the data "write" input, the Q output serving as the "read" output, and the enable input serving as the read/write control line:



If we desire more than one bit's worth of storage (and we probably do), we'll have to have many latches arranged in some kind of an array where we can selectively address which one (or which set) we're reading from or writing to. Using a pair of tristate buffers, we can connect both the data write input and the data read output to a common data bus line, and enable those buffers to either connect the Q output to the data line (READ), connect the D input to the data line (WRITE), or keep both buffers in the High-Z state to disconnect D and Q from the data line (unaddressed mode). One memory "cell" would look like this, internally:

Memory cell circuit



When the address enable input is 0, both tristate buffers will be placed in high-Z mode, and the latch will be disconnected from the data input/output (bus) line. Only when the address enable input is active (1) will the latch be connected to the data bus. Every latch circuit, of course, will be enabled with a different "address enable" (AE) input line, which will come from a 1-of-n output decoder:

In the above circuit, 16 memory cells are individually addressed with a 4-bit binary code input into the decoder. If a cell is not addressed, it will be disconnected from the 1-bit data bus by its internal tristate buffers: consequently, data cannot be either written or read through the bus to or from that cell. Only the cell circuit that is addressed by the 4-bit decoder input will be accessible through the data bus.

This simple memory circuit is random-access and volatile. Technically, it is known as a *static RAM*. Its total memory capacity is 16 bits. Since it contains 16 addresses and has a data bus that is 1 bit wide, it would be designated as a 16 x 1 bit static RAM circuit. As you can see, it takes an incredible number of gates (and multiple transistors per gate!) to construct a practical static RAM circuit. This makes the static RAM a relatively low-density device, with less capacity than most other types of RAM technology per unit IC chip space. Because each cell circuit consumes a certain amount of power, the overall power consumption for a large array of cells can be quite high. Early static RAM banks in personal computers consumed a fair amount of power and generated a lot of heat, too. CMOS IC technology has made it possible to lower the specific power consumption of static RAM circuits, but low storage density is still an issue.

To address this, engineers turned to the capacitor instead of the bistable multivibrator as a means of storing binary data. A tiny capacitor could serve as a memory cell, complete with a single MOSFET transistor for connecting it to the data bus for charging (writing a 1), discharging (writing a 0), or reading. Unfortunately, such tiny capacitors have very small capacitances, and their charge tends to "leak" away through any circuit impedances quite rapidly. To combat this tendency, engineers designed circuits internal to the RAM memory chip which would periodically read all cells and recharge (or "refresh") the capacitors as needed. Although this added to the complexity of the circuit, it still required far less componentry than a RAM built of multivibrators. They called this type of memory circuit a *dynamic RAM*, because of its need of periodic refreshing.

Recent advances in IC chip manufacturing has led to the introduction of *flash* memory, which works on a capacitive storage principle like the dynamic RAM, but uses the insulated gate of a MOSFET as the capacitor itself.

Before the advent of transistors (especially the MOSFET), engineers had to implement digital circuitry with gates constructed from vacuum tubes. As you can imagine, the enormous comparative size and power consumption of a vacuum tube as compared to a transistor made memory circuits like static and dynamic RAM a practical impossibility. Other, rather ingenious, techniques to store digital data without the use of moving parts were developed.

Lessons In Electric Circuits -- Volume IV

Chapter 16

PRINCIPLES OF DIGITAL COMPUTING

A binary adder

Suppose we wanted to build a device that could add two binary bits together. Such a device is known as a half-adder, and its gate circuit looks like this:

The symbol represents the "sum" output of the half-adder, the sum's least significant bit (LSB). C_{out} represents the "carry" output of the half-adder, the sum's most significant bit (MSB).

If we were to implement this same function in ladder (relay) logic, it would look like this:

Either circuit is capable of adding two binary digits together. The mathematical "rules" of how to add bits together are intrinsic to the hard-wired logic of the circuits. If we wanted to perform a different arithmetic operation with binary bits, such as multiplication, we would have to construct another circuit. The above circuit designs will only perform one function: add two binary bits together. To make them do something else would take re-wiring, and perhaps different componentry.

In this sense, digital arithmetic circuits aren't much different from analog arithmetic (operational amplifier) circuits: they do exactly what they're wired to do, no more and no less. We are not, however, restricted to designing digital computer circuits in this manner. It is possible to embed the mathematical "rules" for any arithmetic operation in the form of digital data rather than in hard-wired connections between gates. The

result is unparalleled flexibility in operation, giving rise to a whole new kind of digital device: the *programmable computer*.

While this chapter is by no means exhaustive, it provides what I believe is a unique and interesting look at the nature of programmable computer devices, starting with two devices often overlooked in introductory textbooks: *look-up table memories* and *finite-state machines*.

Look-up tables

Having learned about digital memory devices in the last chapter, we know that it is possible to store binary data within solid-state devices. Those storage "cells" within solid-state memory devices are easily addressed by driving the "address" lines of the device with the proper binary value(s). Suppose we had a ROM memory circuit written, or programmed, with certain data, such that the address lines of the ROM served as inputs and the data lines of the ROM served as outputs, generating the characteristic response of a particular logic function. Theoretically, we could program this ROM chip to emulate whatever logic function we wanted without having to alter any wire connections or gates.

Consider the following example of a 4 x 2 bit ROM memory (a very small memory!) programmed with the functionality of a half adder:

If this ROM has been written with the above data (representing a half-adder's truth table), driving the A and B address inputs will cause the respective memory cells in the ROM chip to be enabled, thus outputting the corresponding data as the (Sum) and C_{out} bits. Unlike the half-adder circuit built of gates or relays, this device can be set up to perform any logic function at all with two inputs and two outputs, not just the half-adder function. To change the logic function, all we would need to do is write a different table of data to another ROM chip. We could even use an EPROM chip which could be re-written at will, giving the ultimate flexibility in function.

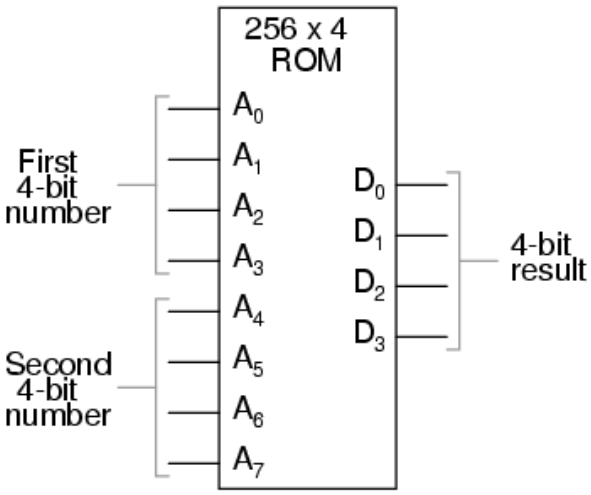
It is vitally important to recognize the significance of this principle as applied to digital circuitry. Whereas the half-adder built from gates or relays *processes* the input bits to arrive at a specific output, the ROM simply *remembers* what the outputs should be for any given combination of inputs. This is not much different from the "times tables" memorized in grade school: rather than having to calculate the product of 5 times 6 (5 + 5 + 5 + 5 + 5 + 5 = 30), school-children are taught to remember that 5 x 6 = 30, and then expected to recall this product from memory as needed. Likewise, rather than the logic function depending on the functional arrangement of hard-wired gates or relays (hardware), it depends solely on the data written into the memory (software).

Such a simple application, with definite outputs for every input, is called a *look-up table*, because the memory device simply "looks up" what the output(s) should to be for any given combination of inputs states.

This application of a memory device to perform logical functions is significant for several reasons:

- Software is much easier to change than hardware.
- Software can be archived on various kinds of memory media (disk, tape), thus providing an easy way to document and manipulate the function in a "virtual" form; hardware can only be "archived" abstractly in the form of some kind of graphical drawing.
- Software can be copied from one memory device (such as the EPROM chip) to another, allowing the ability for one device to "learn" its function from another device.
- Software such as the logic function example can be designed to perform functions that would be extremely difficult to emulate with discrete logic gates (or relays!).

The usefulness of a look-up table becomes more and more evident with increasing complexity of function. Suppose we wanted to build a 4-bit adder circuit using a ROM. We'd require a ROM with 8 address lines (two 4-bit numbers to be added together), plus 4 data lines (for the signed output):



With 256 addressable memory locations in this ROM chip, we would have a fair amount of programming to do, telling it what binary output to generate for each and every combination of binary inputs. We would also run the risk of making a mistake in our programming and have it output an incorrect sum, if we weren't careful. However, the flexibility of being able to configure this function (or any function) through software alone generally outweighs that costs.

Consider some of the advanced functions we could implement with the above "adder." We know that when we add two sets of numbers in 2's complement signed notation, we risk having the answer overflow. For instance, if we try to add 0111 (decimal 7) to 0110 (decimal 6) with only a 4-bit number field, the answer we'll get is 1001 (decimal -7) instead of the correct value, 13 (7 + 6), which cannot be expressed using 4 signed bits. If we wanted to, we could avoid the strange answers given in overflow conditions by programming this look-up table circuit to output something else in conditions where we know overflow will occur (that is, in any case where the real sum would exceed +7 or -8). One alternative might be to program the ROM to

output the quantity 0111 (the maximum positive value that can be represented with 4 signed bits), or any other value that we determined to be more appropriate for the application than the typical overflowed "error" value that a regular adder circuit would output. It's all up to the programmer to decide what he or she wants this circuit to do, because we are no longer limited by the constraints of logic gate functions.

The possibilities don't stop at customized logic functions, either. By adding more address lines to the 256 x 4 ROM chip, we can expand the look-up table to include multiple functions:

With two more address lines, the ROM chip will have 4 times as many addresses as before (1024 instead of 256). This ROM could be programmed so that when A8 and A9 were both low, the output data represented the *sum* of the two 4-bit binary numbers input on address lines A0 through A7, just as we had with the previous 256 x 4 ROM circuit. For the addresses A8=1 and A9=0, it could be programmed to output the *difference* (subtraction) between the first 4-bit binary number (A0 through A3) and the second binary number (A4 through A7). For the addresses A8=0 and A9=1, we could program the ROM to output the difference (subtraction) of the two numbers in reverse order (second - first rather than first - second), and finally, for the addresses A8=1 and A9=1, the ROM could be programmed to compare the two inputs and output an indication of equality or inequality. What we will have then is a device that can perform four different arithmetical operations on 4-bit binary numbers, all by "looking up" the answers programmed into it.

If we had used a ROM chip with more than two additional address lines, we could program it with a wider variety of functions to perform on the two 4-bit inputs. There are a number of operations peculiar to binary data (such as parity check or Exclusive-ORing of bits) that we might find useful to have programmed in such a look-up table.

Devices such as this, which can perform a variety of arithmetical tasks as dictated by a binary input code, are known as *Arithmetic Logic Units* (ALUs), and they comprise one of the essential components of computer technology. Although modern ALUs are more often constructed from very complex combinational logic (gate) circuits for reasons of speed, it should be comforting to know that the exact same functionality may be duplicated with a "dumb" ROM chip programmed with the appropriate look-up table(s). In fact, this exact approach was used by IBM engineers in 1959 with the development of the IBM 1401 and 1620 computers, which used look-up tables to perform addition, rather than binary adder circuitry. The machine was fondly known as the "CADET," which stood for "**C**an't **A**dd, **D**oesn't **E**ven **T**ry."

A very common application for look-up table ROMs is in control systems where a custom mathematical function needs to be represented. Such an application is found in computer-controlled fuel injection systems for automobile engines, where the proper air/fuel mixture ratio for efficient and clean operation changes with several environmental and operational variables. Tests performed on engines in research laboratories determine what these ideal ratios are for varying conditions of engine load, ambient air temperature, and barometric air pressure. The variables are measured with sensor transducers, their analog outputs converted to digital signals with A/D circuitry, and those parallel digital signals used as address inputs to a high-

capacity ROM chip programmed to output the optimum digital value for air/fuel ratio for any of these given conditions.

Sometimes, ROMs are used to provide one-dimensional look-up table functions, for "correcting" digitized signal values so that they more accurately represent their real-world significance. An example of such a device is a *thermocouple transmitter*, which measures the millivoltage signal generated by a junction of dissimilar metals and outputs a signal which is supposed to *directly* correspond to that junction temperature. Unfortunately, thermocouple junctions do not have perfectly linear temperature/voltage responses, and so the raw voltage signal is not perfectly proportional to temperature. By digitizing the voltage signal (A/D conversion) and sending that digital value to the address of a ROM programmed with the necessary correction values, the ROM's programming could eliminate some of the nonlinearity of the thermocouple's temperature-to-millivoltage relationship, so that the final output of the device would be more accurate. The popular instrumentation term for such a look-up table is a digital *characterizer*.

Another application for look-up tables is in special code translation. A 128 x 8 ROM, for instance, could be used to translate 7-bit ASCII code to 8-bit EBCDIC code:

Again, all that is required is for the ROM chip to be properly programmed with the necessary data so that each valid ASCII input will produce a corresponding EBCDIC output code.

Finite-state machines

Feedback is a fascinating engineering principle. It can turn a rather simple device or process into something substantially more complex. We've seen the effects of feedback intentionally integrated into circuit designs with some rather astounding effects:

- Comparator + negative feedback ----------> controllable-gain amplifier
- Comparator + positive feedback ----------> comparator with hysteresis
- Combinational logic + positive feedback --> multivibrator

In the field of process instrumentation, feedback is used to transform a simple measurement system into something capable of control:

- Measurement system + negative feedback ---> closed-loop control system

Feedback, both positive and negative, has the tendency to add whole new dynamics to the operation of a device or system. Sometimes, these new dynamics find useful application, while other times they are merely interesting. With look-up tables programmed into memory devices, feedback from the data outputs back to the address inputs creates a whole new type of device: the *Finite State Machine*, or *FSM*:

The above circuit illustrates the basic idea: the data stored at each address becomes the next storage location that the ROM gets addressed to. The result is a specific sequence of binary numbers (following the sequence programmed into the ROM) at the output, over time. To avoid signal timing problems, though, we need to connect the data outputs back to the address inputs through a 4-bit D-type flip-flop, so that the sequence takes place step by step to the beat of a controlled clock pulse:

An analogy for the workings of such a device might be an array of post-office boxes, each one with an identifying number on the door (the address), and each one containing a piece of paper with the address of another P.O. box written on it (the data). A person, opening the first P.O. box, would find in it the address of the next P.O. box to open. By storing a particular pattern of addresses in the P.O. boxes, we can dictate the sequence in which each box gets opened, and therefore the sequence of which paper gets read.

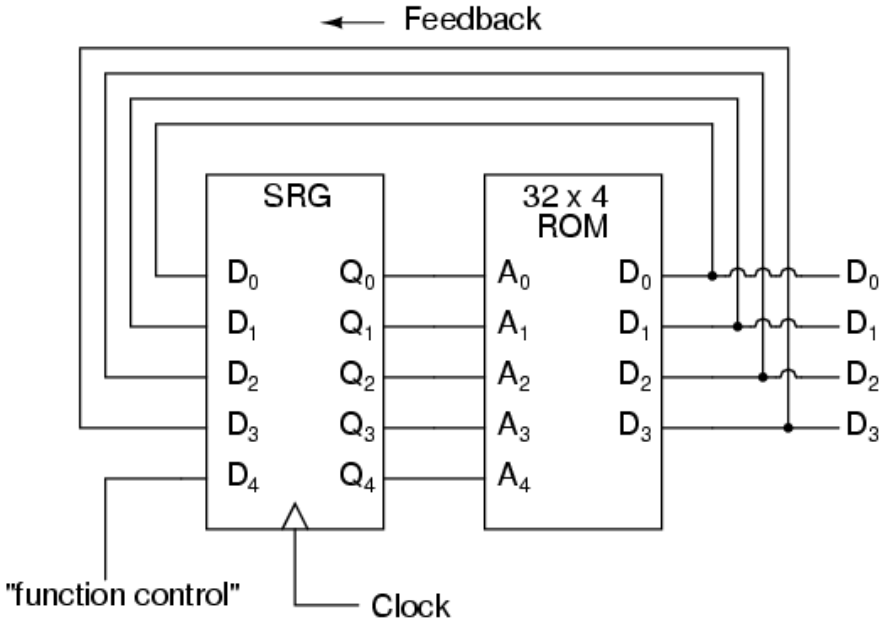
Having 16 addressable memory locations in the ROM, this Finite State Machine would have 16 different stable "states" in which it could latch. In each of those states, the identity of the next state would be programmed in to the ROM, awaiting the signal of the next clock pulse to be fed back to the ROM as an address. One useful application of such an FSM would be to generate an arbitrary count sequence, such as Gray Code:

```
Address  -----> Data          Gray Code count sequence:
0000   -------> 0001                0   0000
0001   -------> 0011                1   0001
0010   -------> 0110                2   0011
0011   -------> 0010                3   0010
0100   -------> 1100                4   0110
0101   -------> 0100                5   0111
0110   -------> 0111                6   0101
0111   -------> 0101                7   0100
1000   -------> 0000                8   1100
1001   -------> 1000                9   1101
1010   -------> 1011               10   1111
1011   -------> 1001               11   1110
1100   -------> 1101               12   1010
1101   -------> 1111               13   1011
1110   -------> 1010               14   1001
1111   -------> 1110               15   1000
```

Try to follow the Gray Code count sequence as the FSM would do it: starting at 0000, follow the data stored at that address (0001) to the next address, and so on (0011), and so on (0010), and so on (0110), etc. The result, for the program table shown, is that the sequence of addressing jumps around from address to address in what looks like a haphazard fashion, but when you check each address that is accessed, you will find that it follows the correct order for 4-bit Gray code. When the FSM arrives at its

last programmed state (address 1000), the data stored there is 0000, which starts the whole sequence over again at address 0000 in step with the next clock pulse.

We could expand on the capabilities of the above circuit by using a ROM with more address lines, and adding more programming data:



Now, just like the look-up table adder circuit that we turned into an Arithmetic Logic Unit (+, -, x, / functions) by utilizing more address lines as "function control" inputs, this FSM counter can be used to generate more than one count sequence, a different sequence programmed for the four feedback bits (A0 through A3) for each of the two function control line input combinations (A4 = 0 or 1).

Address	-----> Data	Address	-----> Data
00000	-------> 0001	10000	-------> 0001
00001	-------> 0010	10001	-------> 0011
00010	-------> 0011	10010	-------> 0110
00011	-------> 0100	10011	-------> 0010
00100	-------> 0101	10100	-------> 1100
00101	-------> 0110	10101	-------> 0100
00110	-------> 0111	10110	-------> 0111
00111	-------> 1000	10111	-------> 0101
01000	-------> 1001	11000	-------> 0000
01001	-------> 1010	11001	-------> 1000
01010	-------> 1011	11010	-------> 1011
01011	-------> 1100	11011	-------> 1001
01100	-------> 1101	11100	-------> 1101
01101	-------> 1110	11101	-------> 1111
01110	-------> 1111	11110	-------> 1010
01111	-------> 0000	11111	-------> 1110

If A4 is 0, the FSM counts in binary; if A4 is 1, the FSM counts in Gray Code. In either case, the counting sequence is arbitrary: determined by the whim of the programmer. For that matter, the counting sequence doesn't even have to have 16

steps, as the programmer may decide to have the sequence recycle to 0000 at any one of the steps at all. It is a completely flexible counting device, the behavior strictly determined by the software (programming) in the ROM.

We can expand on the capabilities of the FSM even more by utilizing a ROM chip with additional address input and data output lines. Take the following circuit, for example:

Here, the D0 through D3 data outputs are used exclusively for feedback to the A0 through A3 address lines. Date output lines D4 through D7 can be programmed to output something other than the FSM's "state" value. Being that four data output bits are being fed back to four address bits, this is still a 16-state device. However, having the output data come from other data output lines gives the programmer more freedom to configure functions than before. In other words, this device can do far more than just count! The programmed output of this FSM is dependent not only upon the state of the feedback address lines (A0 through A3), but also the states of the input lines (A4 through A7). The D-type flip/flop's clock signal input does not have to come from a pulse generator, either. To make things more interesting, the flip/flop could be wired up to clock on some external event, so that the FSM goes to the next state only when an input signal tells it to.

Now we have a device that better fulfills the meaning of the word "programmable." The data written to the ROM is a program in the truest sense: the outputs follow a pre-established order based on the inputs to the device and which "step" the device is on in its sequence. This is very close to the operating design of the *Turing Machine*, a theoretical computing device invented by Alan Turing, mathematically proven to be able to solve any known arithmetic problem, given enough memory capacity.

Microprocessors

Early computer science pioneers such as Alan Turing and John Von Neumann postulated that for a computing device to be really useful, it not only had to be able to generate specific outputs as dictated by programmed instructions, but it also had to be able to write data to memory, and be able to act on that data later. Both the program steps and the processed data were to reside in a common memory "pool," thus giving way to the label of the *stored-program computer*. Turing's theoretical machine utilized a sequential-access tape, which would store data for a control circuit to read, the control circuit re-writing data to the tape and/or moving the tape to a new position to read more data. Modern computers use random-access memory devices instead of sequential-access tapes to accomplish essentially the same thing, except with greater capability.

A helpful illustration is that of early automatic machine tool control technology. Called *open-loop*, or sometimes just *NC* (numerical control), these control systems would direct the motion of a machine tool such as a lathe or a mill by following instructions programmed as holes in paper tape. The tape would be run one direction through a "read" mechanism, and the machine would blindly follow the instructions on the tape without regard to any other conditions. While these devices eliminated the

burden of having to have a human machinist direct every motion of the machine tool, it was limited in usefulness. Because the machine was blind to the real world, only following the instructions written on the tape, it could not compensate for changing conditions such as expansion of the metal or wear of the mechanisms. Also, the tape programmer had to be acutely aware of the sequence of previous instructions in the machine's program to avoid troublesome circumstances (such as telling the machine tool to move the drill bit laterally while it is still inserted into a hole in the work), since the device had no memory other than the tape itself, which was read-only. Upgrading from a simple tape reader to a Finite State control design gave the device a sort of memory that could be used to keep track of what it had already done (through feedback of some of the data bits to the address bits), so at least the programmer could decide to have the circuit remember "states" that the machine tool could be in (such as "coolant on," or tool position). However, there was still room for improvement.

The ultimate approach is to have the program give instructions which would include the writing of new data to a read/write (RAM) memory, which the program could easily recall and process. This way, the control system could record what it had done, and any sensor-detectable process changes, much in the same way that a human machinist might jot down notes or measurements on a scratch-pad for future reference in his or her work. This is what is referred to as CNC, or *Closed-loop Numerical Control*.

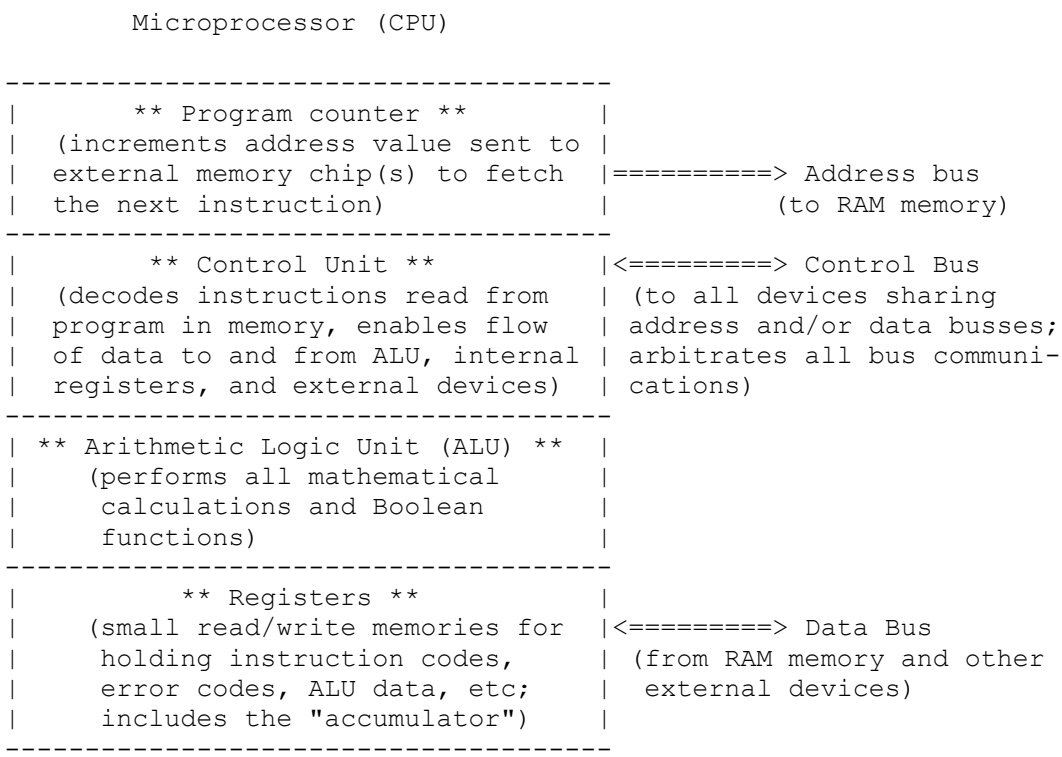
Engineers and computer scientists looked forward to the possibility of building digital devices that could modify their own programming, much the same as the human brain adapts the strength of inter-neural connections depending on environmental experiences (that is why memory retention improves with repeated study, and behavior is modified through consequential feedback). Only if the computer's program were stored in the same writable memory "pool" as the data would this be practical. It is interesting to note that the notion of a self-modifying program is still considered to be on the cutting edge of computer science. Most computer programming relies on rather fixed sequences of instructions, with a separate field of data being the only information that gets altered.

To facilitate the stored-program approach, we require a device that is much more complex than the simple FSM, although many of the same principles apply. First, we need read/write memory that can be easily accessed: this is easy enough to do. Static or dynamic RAM chips do the job well, and are inexpensive. Secondly, we need some form of logic to process the data stored in memory. Because standard and Boolean arithmetic functions are so useful, we can use an Arithmetic Logic Unit (ALU) such as the look-up table ROM example explored earlier. Finally, we need a device that controls how and where data flows between the memory, the ALU, and the outside world. This so-called *Control Unit* is the most mysterious piece of the puzzle yet, being comprised of tri-state buffers (to direct data to and from buses) and decoding logic which interprets certain binary codes as instructions to carry out. Sample instructions might be something like: "add the number stored at memory address 0010 with the number stored at memory address 1101," or, "determine the parity of the data in memory address 0111." The choice of which binary codes represent which instructions for the Control Unit to decode is largely arbitrary, just as the choice of which binary codes to use in representing the letters of the alphabet in the ASCII standard was largely arbitrary. ASCII, however, is now an internationally

recognized standard, whereas control unit instruction codes are almost always manufacturer-specific.

Putting these components together (read/write memory, ALU, and control unit) results in a digital device that is typically called a *processor*. If minimal memory is used, and all the necessary components are contained on a single integrated circuit, it is called a *microprocessor*. When combined with the necessary bus-control support circuitry, it is known as a *Central Processing Unit*, or CPU.

CPU operation is summed up in the so-called *fetch/execute cycle*. *Fetch* means to read an instruction from memory for the Control Unit to decode. A small binary counter in the CPU (known as the *program counter* or *instruction pointer*) holds the address value where the next instruction is stored in main memory. The Control Unit sends this binary address value to the main memory's address lines, and the memory's data output is read by the Control Unit to send to another holding register. If the fetched instruction requires reading more data from memory (for example, in adding two numbers together, we have to read both the numbers that are to be added from main memory or from some other source), the Control Unit appropriately addresses the location of the requested data and directs the data output to ALU registers. Next, the Control Unit would execute the instruction by signaling the ALU to do whatever was requested with the two numbers, and direct the result to another register called the *accumulator*. The instruction has now been "fetched" and "executed," so the Control Unit now increments the program counter to step the next instruction, and the cycle repeats itself.



As one might guess, carrying out even simple instructions is a tedious process. Several steps are necessary for the Control Unit to complete the simplest of mathematical procedures. This is especially true for arithmetic procedures such as exponents, which involve repeated executions ("iterations") of simpler functions. Just imagine the sheer quantity of steps necessary within the CPU to update the bits of information for the graphic display on a flight simulator game! The only thing which makes such a tedious process practical is the fact that microprocessor circuits are able to repeat the fetch/execute cycle with great speed.

In some microprocessor designs, there are minimal programs stored within a special ROM memory internal to the device (called *microcode*) which handle all the sub-steps necessary to carry out more complex math operations. This way, only a single instruction has to be read from the program RAM to do the task, and the programmer doesn't have to deal with trying to tell the microprocessor how to do every minute step. In essence, its a processor inside of a processor; a program running inside of a program.

Microprocessor programming

The "vocabulary" of instructions which any particular microprocessor chip possesses is specific to that model of chip. An Intel 80386, for example, uses a completely different set of binary codes than a Motorola 68020, for designating equivalent functions. Unfortunately, there are no standards in place for microprocessor instructions. This makes programming at the very lowest level very confusing and specialized.

When a human programmer develops a set of instructions to directly tell a microprocessor how to do something (like automatically control the fuel injection rate to an engine), they're programming in the CPU's own "language." This language, which consists of the very same binary codes which the Control Unit inside the CPU chip decodes to perform tasks, is often referred to as *machine language*. While machine language software can be "worded" in binary notation, it is often written in hexadecimal form, because it is easier for human beings to work with. For example, I'll present just a few of the common instruction codes for the Intel 8080 micro-processor chip:

Hexadecimal	Binary	Instruction description
7B	01111011	Move contents of register A to register E
87	10000111	Add contents of register A to register D
1C	00011100	Increment the contents of register E by 1
D3	11010011	Output byte of data to data bus

Even with hexadecimal notation, these instructions can be easily confused and forgotten. For this purpose, another aid for programmers exists called *assembly*

language. With assembly language, two to four letter mnemonic words are used in place of the actual hex or binary code for describing program steps. For example, the instruction 7B for the Intel 8080 would be "MOV A,E" in assembly language. The mnemonics, of course, are useless to the microprocessor, which can only understand binary codes, but it is an expedient way for programmers to manage the writing of their programs on paper or text editor (word processor). There are even programs written for computers called *assemblers* which understand these mnemonics, translating them to the appropriate binary codes for a specified target microprocessor, so that the programmer can write a program in the computer's native language without ever having to deal with strange hex or tedious binary code notation.

Once a program is developed by a person, it must be written into memory before a microprocessor can execute it. If the program is to be stored in ROM (which some are), this can be done with a special machine called a *ROM programmer*, or (if you're masochistic), by plugging the ROM chip into a breadboard, powering it up with the appropriate voltages, and writing data by making the right wire connections to the address and data lines, one at a time, for each instruction. If the program is to be stored in volatile memory, such as the operating computer's RAM memory, there may be a way to type it in by hand through that computer's keyboard (some computers have a mini-program stored in ROM which tells the microprocessor how to accept keystrokes from a keyboard and store them as commands in RAM), even if it is too dumb to do anything else. Many "hobby" computer kits work like this. If the computer to be programmed is a fully-functional personal computer with an operating system, disk drives, and the whole works, you can simply command the assembler to store your finished program onto a disk for later retrieval. To "run" your program, you would simply type your program's filename at the prompt, press the Enter key, and the microprocessor's Program Counter register would be set to point to the location ("address") on the disk where the first instruction is stored, and your program would run from there.

Although programming in machine language or assembly language makes for fast and highly efficient programs, it takes a lot of time and skill to do so for anything but the simplest tasks, because each machine language instruction is so crude. The answer to this is to develop ways for programmers to write in "high level" languages, which can more efficiently express human thought. Instead of typing in dozens of cryptic assembly language codes, a programmer writing in a high-level language would be able to write something like this . . .

```
Print "Hello, world!"
```

. . . and expect the computer to print "Hello, world!" with no further instruction on how to do so. This is a great idea, but how does a microprocessor understand such "human" thinking when its vocabulary is so limited?

The answer comes in two different forms: *interpretation*, or *compilation*. Just like two people speaking different languages, there has to be some way to transcend the language barrier in order for them to converse. A translator is needed to translate each person's words to the other person's language, one way at a time. For the

microprocessor, this means another program, written by another programmer in machine language, which recognizes the ASCII character patterns of high-level commands such as Print (P-r-i-n-t) and can translate them into the necessary bite-size steps that the microprocessor can directly understand. If this translation is done during program execution, just like a translator intervening between two people in a live conversation, it is called "interpretation." On the other hand, if the entire program is translated to machine language in one fell swoop, like a translator recording a monologue on paper and then translating all the words at one sitting into a written document in the other language, the process is called "compilation."

Interpretation is simple, but makes for a slow-running program because the microprocessor has to continually translate the program between steps, and that takes time. Compilation takes time initially to translate the whole program into machine code, but the resulting machine code needs no translation after that and runs faster as a consequence. Programming languages such as BASIC and FORTH are interpreted. Languages such as C, C++, FORTRAN, and PASCAL are compiled. Compiled languages are generally considered to be the languages of choice for professional programmers, because of the efficiency of the final product.

Naturally, because machine language vocabularies vary widely from microprocessor to microprocessor, and since high-level languages are designed to be as universal as possible, the interpreting and compiling programs necessary for language translation must be microprocessor-specific. Development of these interpreters and compilers is a most impressive feat: the people who make these programs most definitely earn their keep, especially when you consider the work they must do to keep their software product current with the rapidly-changing microprocessor models appearing on the market!

To mitigate this difficulty, the trend-setting manufacturers of microprocessor chips (most notably, Intel and Motorola) try to design their new products to be *backwardly compatible* with their older products. For example, the entire instruction set for the Intel 80386 chip is contained within the latest Pentium IV chips, although the Pentium chips have additional instructions that the 80386 chips lack. What this means is that machine-language programs (compilers, too) written for 80386 computers will run on the latest and greatest Intel Pentium IV CPU, but machine-language programs written specifically to take advantage of the Pentium's larger instruction set will not run on an 80386, because the older CPU simply doesn't have some of those instructions in its vocabulary: the Control Unit inside the 80386 cannot decode them.

Building on this theme, most compilers have settings that allow the programmer to select which CPU type he or she wants to compile machine-language code for. If they select the 80386 setting, the compiler will perform the translation using only instructions known to the 80386 chip; if they select the Pentium setting, the compiler is free to make use of all instructions known to Pentiums. This is analogous to telling a translator what minimum reading level their audience will be: a document translated for a child will be understandable to an adult, but a document translated for an adult may very well be gibberish to a child.

To understand the basis of some basic clock and sequencer circuits, here's a little on comparators and 555 timer ic's:

Voltage comparator

How to use an op-amp as a comparator
- This only requires a single operational amplifier.
- The models 1458 and 353 are both "dual" op-amp units, with two complete amplifier circuits housed in the same 8-pin DIP package.

SCHEMATIC DIAGRAM

- A *comparator* circuit compares two voltage signals and determines which one is greater.

- The result of this comparison is indicated by the output voltage:

- if the op-amp's output is saturated in the positive direction, the non-inverting input (+) is a greater, or more positive, voltage than the inverting input (-);
- [all voltages measured with respect to ground].
- If the op-amp's voltage is near the negative supply voltage (in this case, 0 volts, or ground potential), it means the inverting input (-) has a greater voltage applied to it than the non-inverting input (+).

- In the example circuit, two potentiometers supply variable voltages to be compared by the op-amp.
- The output status of the op-amp is indicated visually by the LED. By adjusting the two potentiometers and observing the LED, one can easily comprehend the function of a comparator circuit.

555 TIMER BASED CLOCK CIRCUITS

- see this link for how a 555 timer ic works:

PWM power controller

SCHEMATIC DIAGRAM

INSTRUCTIONS

- This circuit uses a 555 timer to generate a sawtooth voltage waveform across a capacitor, then compares that signal against a steady voltage provided by a potentiometer, using an op-amp as a comparator.

- The comparison of these two voltage signals produces a square-wave output from the op-amp, varying in duty cycle according to the potentiometer's position.

- Controlling electrical power through a load by means of quickly switching it on and off, and varying the "on" time, is known as *pulse-width modulation*, or *PWM*.

- When the transistor is in cutoff, its power dissipation is zero because there is no current through it.
- When the transistor is saturated, its dissipation is very low because there is little voltage dropped between collector and emitter while it is conducting current.

- When you examine the schematic, you will notice *two* operational amplifiers connected in parallel.
- This is done to provide maximum current output to the base terminal of the power transistor.
- A single op-amp (one-half of a 1458 IC) may not be able to provide sufficient output current to drive the transistor into saturation, so two op-amps are used in tandem.
- This should only be done if the op-amps in question are overload-protected, which the 1458 series of op-amps are. Otherwise, it is possible (though unlikely) that one op-amp could turn on before the other, and damage result from the two outputs short-circuiting each other (one driving "high" and the other driving "low" simultaneously).
- The inherent short-circuit protection offered by the 1458 allows for direct driving of the power transistor base without any need for a current-limiting resistor.

- The three diodes in series connecting the op-amps' outputs to the transistor's base are there to drop voltage and ensure the transistor falls into cutoff when the op-amp outputs go low".
- Because the 1458 op-amp cannot swing its output voltage all the way down to ground potential, but only to within about 2 volts of ground, a direct connection from the op-amp to the transistor would mean the transistor would never fully turn off.
- Adding three silicon diodes in series drops approximately 2.1 volts (0.7 volts times 3) to ensure there is minimal voltage at the transistor's base when the op-amp outputs go "low."

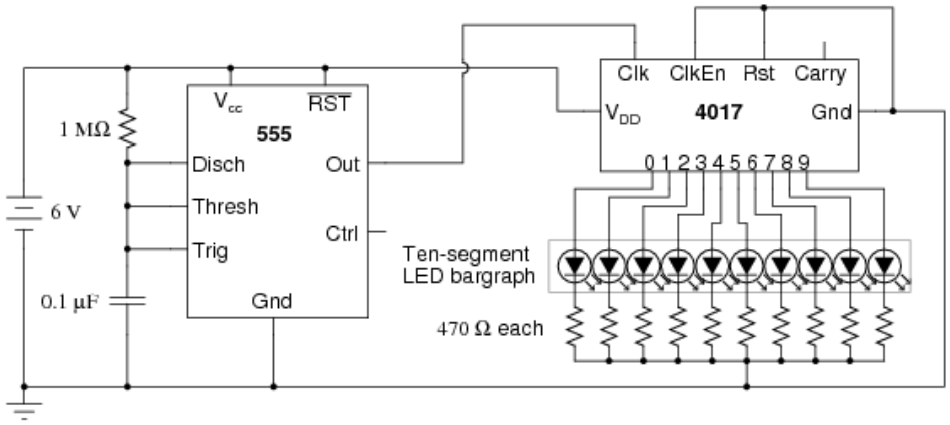
- It is interesting to listen to the op-amp output signal through an audio detector as the potentiometer is adjusted through its full range of motion.
- Adjusting the potentiometer has no effect on signal frequency, but it greatly affects duty cycle.
- Note the difference in tone quality, or *timbre*, as the potentiometer varies - the duty cycle from 0% to 50% to 100%.
Varying the duty cycle has the effect of changing the harmonic content of the waveform, which makes the tone sound different.

- You might notice a particular uniqueness to the sound heard through the detector headphones when the potentiometer is in center position (50% duty cycle - 50% load power), versus a kind of similarity in sound just above or below 50% duty cycle.
- This is due to the absence or presence of even-numbered harmonics.
- Any waveform that is symmetrical above and below its centerline, such as a square wave with a 50% duty cycle, contains *no* even-numbered harmonics, only odd-numbered.
- If the duty cycle is below or above 50%, the waveform will *not* exhibit this symmetry, and there will be even-numbered harmonics.
- The presence of these even-numbered harmonic frequencies can be detected by the human ear, as some of them correspond to *octaves* of the fundamental frequency and thus "fit" more naturally into the tone scheme.

LED sequencer

1.

SCHEMATIC DIAGRAM



INSTRUCTIONS

- The model 4017 integrated circuit is a CMOS counter with ten output terminals. One of these ten terminals will be in a "high" state at any given time, with all others being "low," giving a "one-of-ten" output sequence.
- If low-to-high voltage pulses are applied to the "clock" (Clk) terminal of the 4017, it will increment its count, forcing the next output into a "high" state.

– With a 555 timer connected as an astable multivibrator (oscillator) of low frequency, the 4017 will cycle through its ten-count sequence, lighting up each LED, one at a time, and "recycling" back to the first LED.
- Different resistor and capacitor values on the 555 timer will create different flash rates.

- Two terminals on the 4017 chip, "Reset" and "Clock Enable," are maintained in a "low" state by means of a connection to the negative side of the battery (ground).
- This is necessary if the chip is to count freely. If the "Reset" terminal is made "high," the 4017's output will be reset back to 0 (pin #3 "high," all other output pins "low").
- If the "Clock Enable" is made "high," the chip will stop responding to the clock signal and pause in its counting sequence.

- If the 4017's "Reset" terminal is connected to one of its ten output terminals, its counting sequence will be cut short, or *truncated*.[Notice how many (or how few) LEDs light up with the "Reset" connected to any one of the outputs]

- Counters such as the 4017 may be used as digital frequency dividers, to take a clock signal and produce a pulse occurring at some integer factor of the clock frequency.
- For example, if the clock signal from the 555 timer is 200 Hz, and the 4017 is configured for a full-count sequence (the "Reset" terminal connected to ground, giving a full, ten-step count), a signal with a period ten times as long (20 Hz) will be present at any of the 4017's output terminals.
- In other words, each output terminal will cycle *once* for every *ten* cycles of the clock signal: a frequency ten times as slow.

- By connecting the 4017's "Reset" terminal to one of the output terminals, a truncated sequence will result.
- If we are using the 4017 as a frequency divider, this means the output frequency will be a different factor of the clock frequency: 1/9, 1/8, 1/7, 1/6, 1/5, 1/4, 1/3, or 1/2, depending on which output terminal we connect the "Reset" jumper wire to.

And for fun and inspiration:

Simple combination lock

- This circuit may be built using only one 8-position DIP switch, but the concept is easier to understand if two switch assemblies are used.
- The idea is, one switch acts to hold the correct code for unlocking the lock, while the other switch serves as a data entry point for the person trying to open the lock.

- In real life, of course, the switch assembly with the "key" code set on it must be hidden from the sight of the person opening the lock, which means it must be physically located *elsewhere* from where the data entry switch assembly is.
- This requires two switch assemblies.
- However, if you understand this concept clearly, you may build a working circuit with only one 8-position switch, using the left four switches for data entry and the right four switches to hold the "key" code.

SCHEMATIC DIAGRAM

- This circuit illustrates the use of XOR (Exclusive-OR) gates as bit comparators.

- Four of these XOR gates compare the respective bits of two 4-bit binary numbers, each number "entered" into the circuit via a set of switches.
- If the two numbers match, bit for bit, the green "Go" LED will light up when the "Enter" pushbutton switch is pressed.
- If the two numbers do not exactly match, the red "No go" LED will light up when the "Enter" pushbutton is pressed.

[- Because four bits provides a mere sixteen possible combinations, this lock circuit is not very sophisticated.
- If it were used in a real application such as a home security system, the "No go" output would have to be connected to some kind of siren or other alarming device, so that the entry of an incorrect code would deter an unauthorized person from attempting another code entry.
- Otherwise, it would not take much time to try all combinations (0000 through 1111) until the correct one was found!]

[[- The "key" code that must be matched at the data entry switch array should be hidden from view, of course.
- If this were part of a real security system, the data entry switch assembly would be located *outside* the door, and the key code switch assembly *behind* the door with the rest of the circuitry.
- In this experiment, you will likely locate the two switch assemblies on two different breadboards, but it is entirely possible to build the circuit using just a single (8-position) DIP switch assembly.
- Again, the purpose of the example is not to make a real security system, but merely to introduce you to the principle of XOR gate code comparisonand give patch/circuit ideas.]]

- It is the nature of an XOR gate to output a "high" (1) signal if the input signals are *not* the same logic state.

- The four XOR gates' output terminals are connected through a diode network which functions as a four-input OR gate:

- if *any* of the four XOR gates outputs a "high" signal -- indicating that the entered code and the key code are not identical -- then a "high" signal will be passed on to the NOR gate logic.
- If the two 4-bit codes are identical, then none of the XOR gate outputs will be "high," and the pull-down resistor connected to the common sides of the diodes will provide a "low" signal state to the NOR logic.

- The NOR gate logic performs a simple task:
prevent either of the LEDs from turning on if the "Enter" pushbutton is not pressed. Only when this pushbutton is pressed can either of the LEDs energize.

- If the Enter switch is pressed and the XOR outputs are all "low," the "Go" LED will light up, indicating that the correct code has been entered.
- If the Enter switch is pressed and any of the XOR outputs are "high," the "No go" LED will light up, indicating that an incorrect code has been entered.

[- Again, if this were a real security system, it would be wise to have the "No go" output do something that deters an unauthorized person from discovering the correct code by trial-and-error. In other words, there should be some sort of *penalty* for entering an incorrect code. Let your imagination guide your design of this detail!]