**Chapter 11: Digital Logic**

Table of Contents

[11.1 Boolean Algebra 2](#_Toc50662239)

[11.2 Gates 4](#_Toc50662240)

[11.3 Combinational Circuits 6](#_Toc50662241)

[Implementation of Boolean Functions 6](#_Toc50662242)

[Multiplexers 11](#_Toc50662243)

[Decoders 12](#_Toc50662244)

[Read-Only Memory 13](#_Toc50662245)

[Adders 14](#_Toc50662246)

[11.4 Sequential Circuits 15](#_Toc50662247)

[Flip-Flops 15](#_Toc50662248)

[Registers 18](#_Toc50662249)

[Counters 19](#_Toc50662250)

[11.5 Programmable Logic Devices 21](#_Toc50662251)

[Programmable Logic Array 21](#_Toc50662252)

[Field-Programmable Gate Array 22](#_Toc50662253)

## 11.1 Boolean Algebra

The digital circuitry in digital computers and other digital systems is designed with the use of a mathematical discipline called Boolean algebra. It is used in analysis, since it is an economical way to describe the function of digital circuity, and in design, since, given a desired function, Boolean algebra can develop a simplified implementation of that function.

Boolean algebra makes use of logical variables and operations. The variables may take a value of to represent TRUE or to represent FALSE. The basic logical operations are AND, OR and NOT which are symbolically represented as , and (NOT ). When it is not ambiguous, the AND operation is also shown as a simple concatenation of the operations e.g. .

The operation AND yields TRUE if and only if both of its operations are true. The operation OR yields true if either or both of the operands are true. The operation NOT inverts the value of the operand. In the absence of parentheses, the AND operation takes precedence over the OR operation.

A truth table is often used along with an operation which shows the result of the operation for every possible combination of operands.

There are three more useful operators, XOR, NAND and NOR. The XOR of two logical operands if if and only if exactly one of the operands has the value . The NAND function is the complement of the NOT of the AND function and the NOR function is the complement of the NOT of the OR function.

There are a few identities from Boolean algebra listed below. The basic postulates are accepted without proof. The other postulates can be derived from the basic ones.

|  |  |  |
| --- | --- | --- |
| **Basic Postulates** | | |
|  |  | Commutative Laws |
|  |  | Distributive Laws |
|  |  | Identity Elements |
|  |  | Inverse Elements |
| **Other Identities** | | |
|  |  |  |
|  |  |  |
|  |  | Associative Laws |
|  |  | De Morgan’s Theorem |

## 11.2 Gates

A gate is an electronic circuit that produces an output signal that is a simple Boolean operation on its input signals. Logical functions are implemented by the interconnection of gates. The basic gates used in digital logic are AND, OR, NOT, NAND, NOR and XOR.

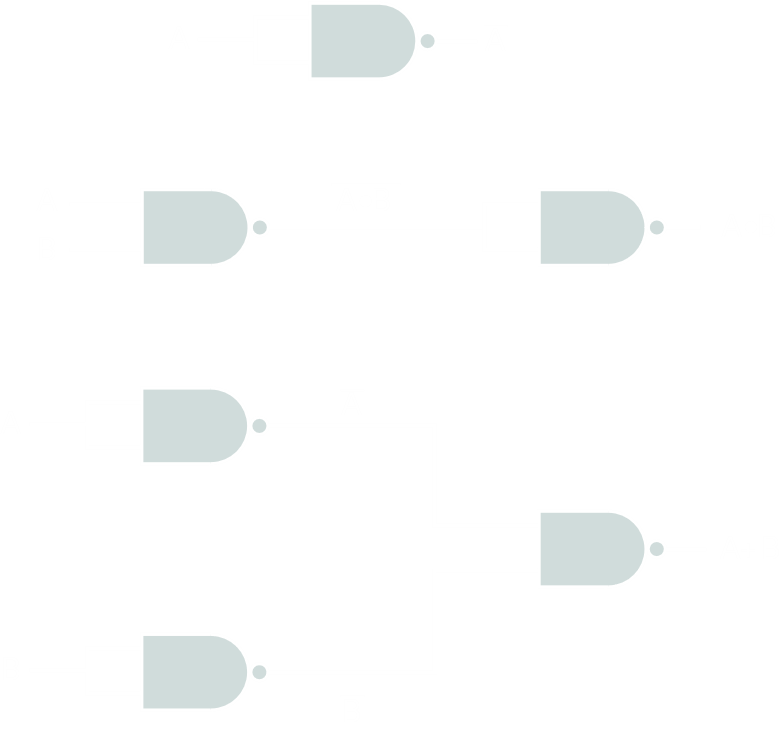
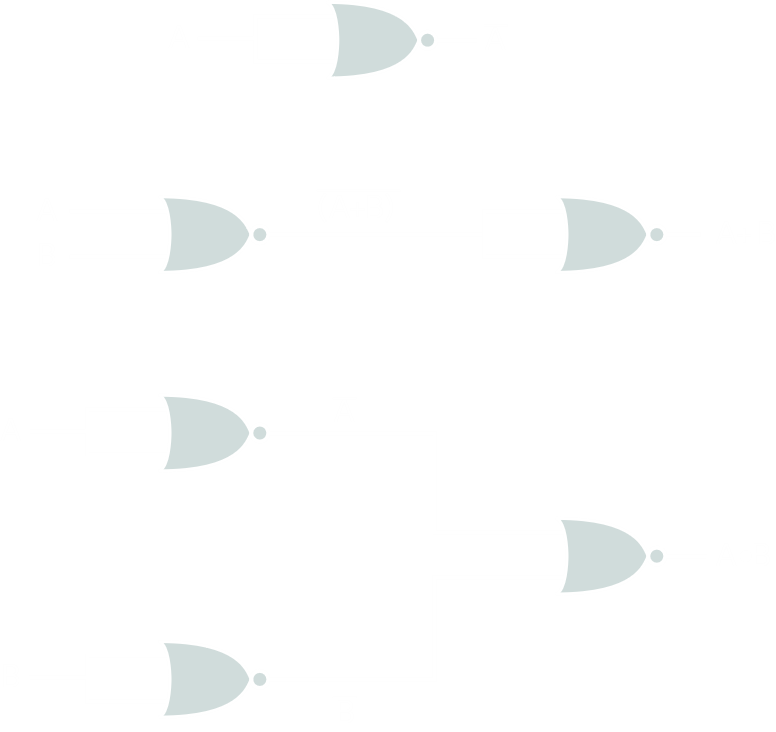
|  |  |  |  |
| --- | --- | --- | --- |
| Name | Graphical Symbol | Algebraic Function | Truth Table |
| AND |  | or | |  |  |  | | --- | --- | --- | | A | B | F | | 0 | 0 | 0 | | 0 | 1 | 0 | | 1 | 0 | 0 | | 1 | 1 | 1 | |
| OR |  |  | |  |  |  | | --- | --- | --- | | A | B | F | | 0 | 0 | 0 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 1 | |
| NOT |  | or | |  |  | | --- | --- | | A | F | | 0 | 1 | | 1 | 0 | |
| NAND |  |  | |  |  |  | | --- | --- | --- | | A | B | F | | 0 | 0 | 1 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 0 | |
| NOR |  |  | |  |  |  | | --- | --- | --- | | A | B | F | | 0 | 0 | 1 | | 0 | 1 | 0 | | 1 | 0 | 0 | | 1 | 1 | 0 | |
| XOR |  |  | |  |  |  | | --- | --- | --- | | A | B | F | | 0 | 0 | 0 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 0 | |

Each of the gates shown, except NOT, can have more than 2 inputs. When one or more values at the input are changed, the correct output signals appear almost instantly, delayed only by the propagation time of signals through the gate (the gate delay).

It is simpler to use only one or two types of gates for a single implementation. Thus, functionally complete sets of gates are used. Any Boolean function can be implemented using only the gates in a single set. The functionally complete sets are:

* AND, OR and NOT
* AND and NOT
* OR and NOT
* NAND
* NOR

AND and NOT can be used to create an OR using De Morgan’s theorem. Similarly, OR and NOT can be used to create an AND. All three of the basic operations can be implemented using only NAND or only NOR gates.

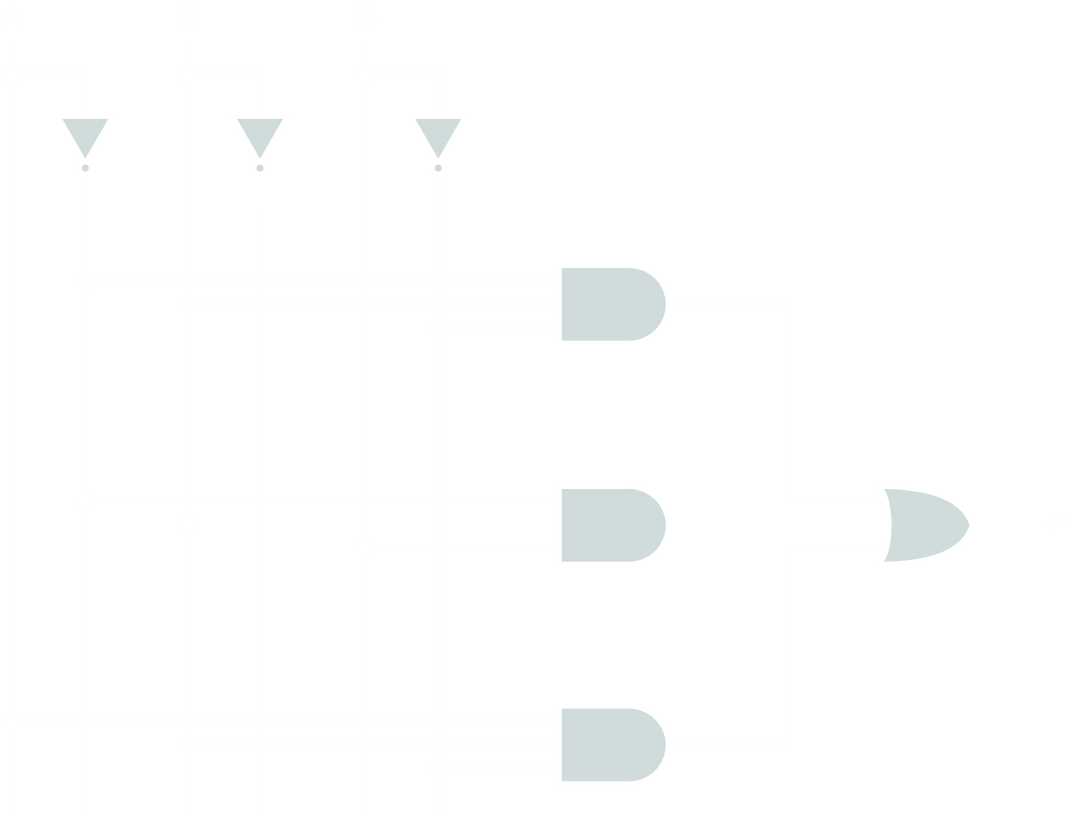
## 11.3 Combinational Circuits

### Implementation of Boolean Functions

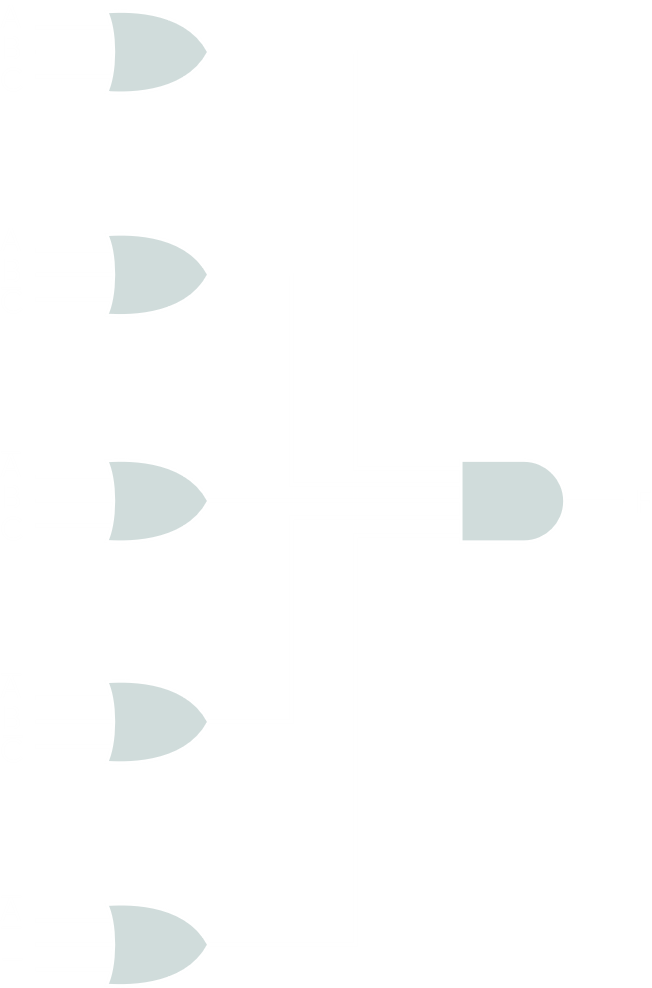
A combinational circuit is an interconnected set of gates whose output at any time is a function only of the inputs at that time. A single Boolean function can have a number of possible inputs, which all result in either a or a as output. For example, for the function , the truth table stands:

|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **B** | **C** | **D** |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

For obvious reasons, the function as it is currently depicted is called the sum of products (SOP) form.



The SOP form essentially says that is true if any of the input conditions that cause to be are true. Alternatively, we could say that is true if none of the input conditions that cause to be false are true, i.e. . This can also be rewritten using De Morgan’s theorem using the general formula on each term. This is thus called the product of sums (POS) form.



At this point, it should be obvious that since SOP has one term for each and POS has one term for each , which form to use depends on whether the truth table has more s or . However, we also need to consider that it may be preferable to implement the function using a single type of gate, i.e. NAND or NOR, and that sometimes it is possible to derive a simpler Boolean expression from the truth table than either SOP or POS. This second point means fewer gates will be needed. Three methods can be used to achieve this, algebraic simplification, Karnaugh maps and Quine-McCluskey tables.

Algebraic simplification involved applying the identities seen earlier to reduce the Boolean expression to one with fewer elements. This however, can only be done with simple expression. More complex expressions require a systematic approach.

Karnaugh maps are convenient when representing Boolean functions with up to 4 variables.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ABCD | 00 | 01 | 11 | 10 |
| 00 | 1 | 1 | 1 | 1 |
| 01 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 |

Before starting to work with the Karnaugh map, it is essential to put the expression in the canonical form, where each term contains every variable. In the map, if adjacent cells have s in them, the variable that differs in those cells can be ignored in the expression. Thus, and have been ignored above. Note that groups of cells can only be formed with cells. We can have overlaps, but every needs to be included. If any group exists that is completely overlapped by other groups, that group can be eliminated since is results are redundant.

Under certain conditions, some combinations do not matter and are called ‘don’t care’ conditions. They are marked with a and can be considered to be a or while making the groups, whichever is more convenient.

The Quine-McCluskey method is used with more than four variables and is suitable for programming on a computer to give an automatic tool for producing minimized Boolean expressions.

Consider the expression

The first step is to create a list of all the terms in the expression. We group together the terms that have the same number of uncomplemented variables (same number of s). Next, we make a list of every possible pair between a member of one group and a member of the group just after it, for which the terms only differ by a single variable. We leave the variable that is different blank. If a term is used in a pair, we place a checkmark next to it. We repeat this until a new list cannot be created since there are not terms that can be paired.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| First List | | Second List | | Third List | |
| 0001 | ✓ | \_001 |  | 10\_ \_ |  |
| 0100 | ✓ | 01\_0 |  | 10\_ \_ |  |
| 1000 | ✓ | 100\_ | ✓ |  |
| 0110 | ✓ | 10\_0 | ✓ |  |
| 1001 | ✓ | 011\_ |  |  |
| 1010 | ✓ | 10\_1 | ✓ |  |
| 0111 | ✓ | 101\_ | ✓ |  |
| 1011 | ✓ | \_111 |  |  |
| 1111 | ✓ | 1\_11 |  |  |

The terms that have no checkmarks next to them are used to construct a final table that is used to check which terms from the original expression each of the terms we have found can cover.

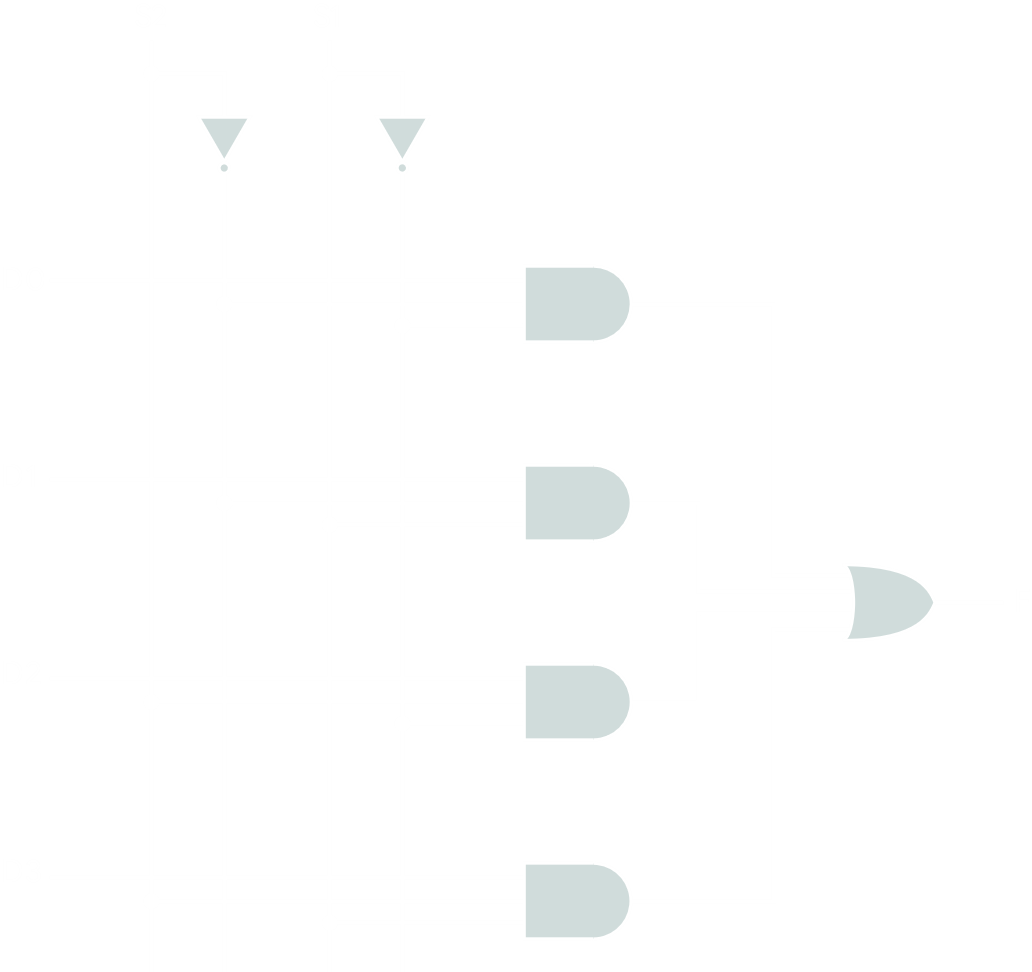
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  | X |  | X | X |  | X |  |
|  | X |  |  |  | X |  |  |  |  |
|  |  | X |  | X |  |  |  |  |  |
|  |  |  |  | X |  |  | X |  |  |
|  |  |  |  |  |  |  | X |  | X |
|  |  |  |  |  |  |  |  | X | X |
|  | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |  |

From this table, we pick the minimum number of terms possible with which we can cover all the terms from the original expression. In this case, this is , , and . Thus, the final expression is . If by some chance there are terms from the original expression that are not covered by the new terms we have found, we need to take additional terms so that they can be covered.

It is sometimes desirable to implement a Boolean function using just NAND or just NOR gates. In these cases, De Morgan’s theorem can help us construct the required expression from the original one.

### Multiplexers

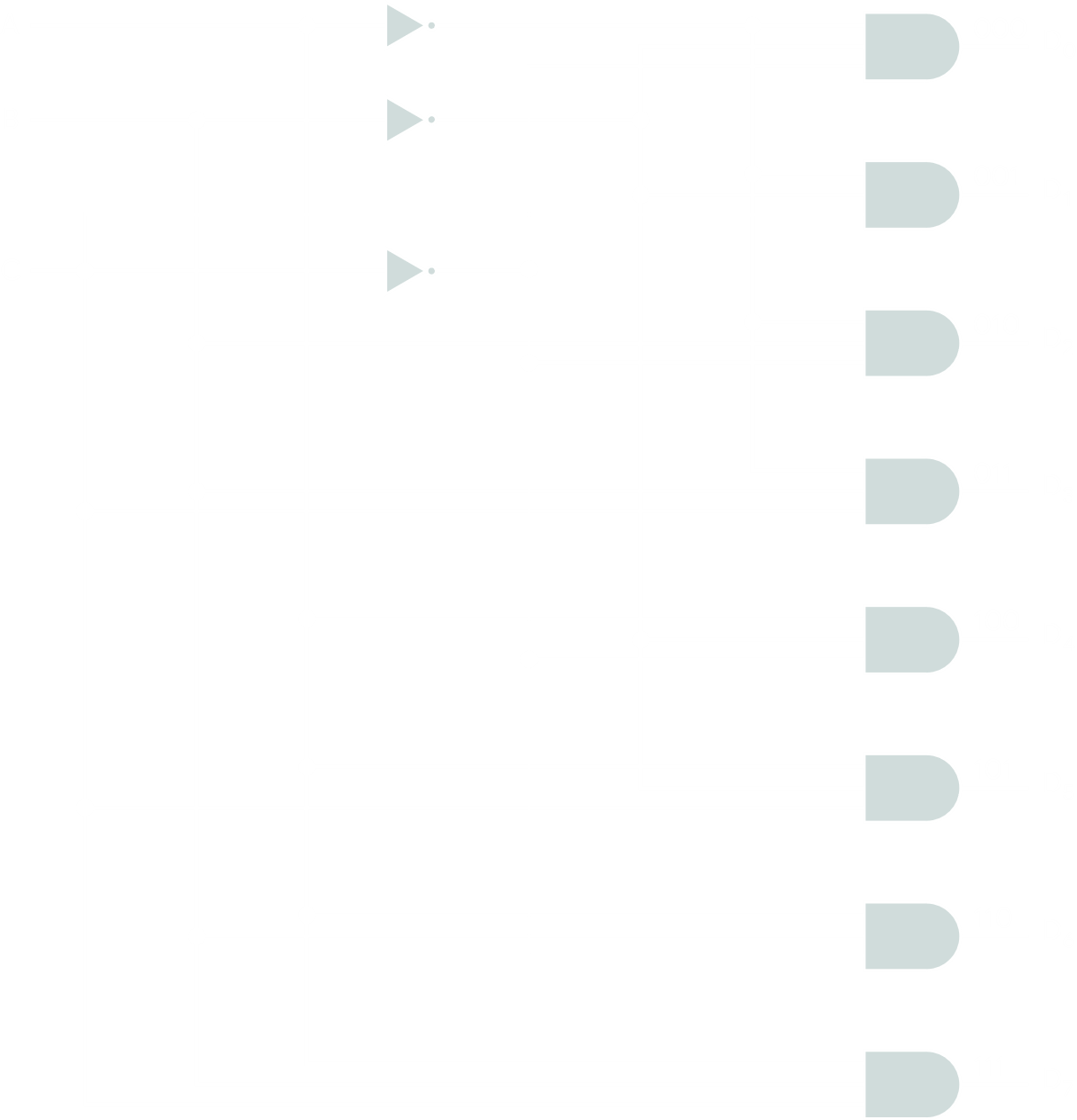
A multiplexer connects multiple inputs to a single output, selecting a single input to be passed to the output. For a 4-to-1 multiplexer like the one below, we would require 2 selection bits (since ) to select which input to pass.



Multiplexers are used in digital circuits to control signals and data routing. For example, in the program counter (PC), instructions may come from a binary counter, the instruction register or the ALU.

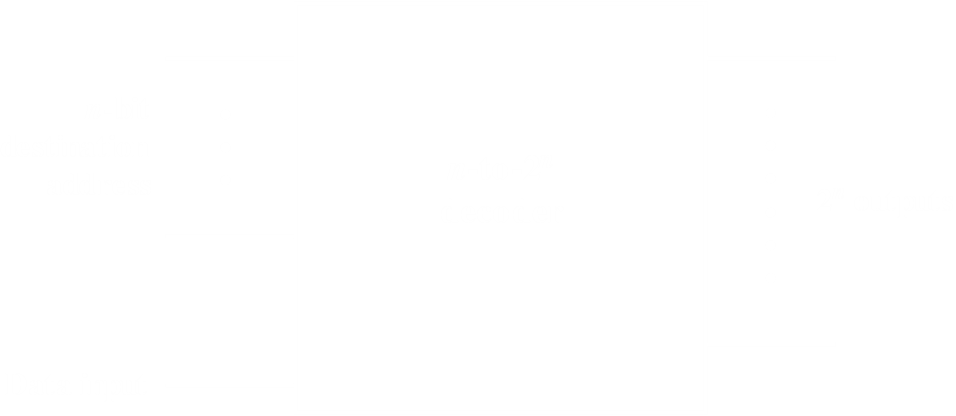
### Decoders

A decoder is a combinational circuit with multiple output lines, only one of which is asserted at any time, depending on the patter of input lines. For a decoder with inputs, there are outputs.



Decoders are used in things like address decoding for memory addressing. All the addresses already exist, but one must be selected.

Using an additional input line, a decoder can be used as a demultiplexer, which connects a single input to multiple inputs. The inputs are used to select a particular output line to which the extra input, the data input, will go.



Another way of looking at this same device is to imagine the data line to be an ‘enable’ line, which would allow us to control the timing of the decoder.

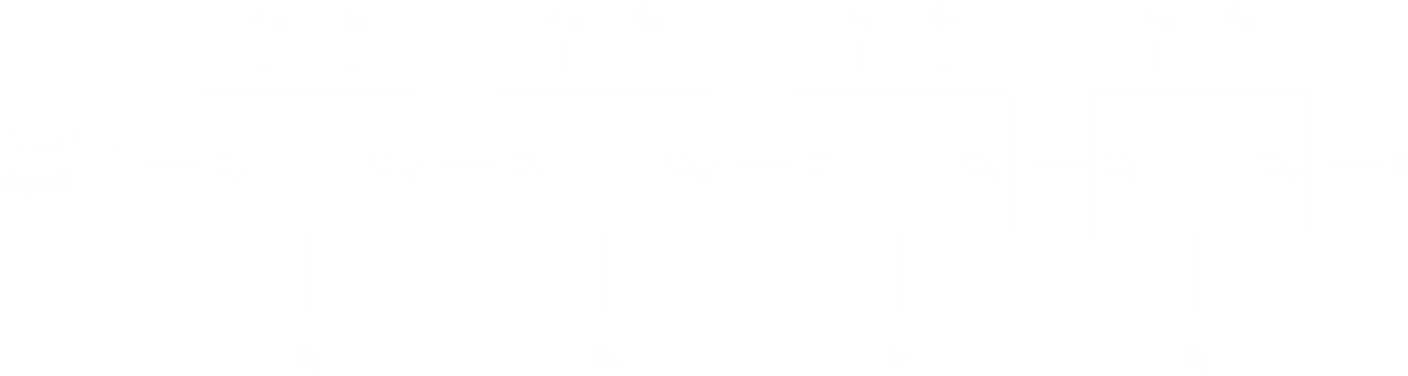
### Read-Only Memory

Combinational circuits have outputs that depend purely on their current input with no history of prior outputs. As such, they are referred to as ‘memoryless’ circuits. However, ROM is implemented using combinational circuitry. The binary information stored in ROM is permanent and input given to the ROM always produces the same output since it is read-only. Thus, the outputs are a function of the present inputs, making it a combinational circuit.

A ROM can be made with a decoder and a set of OR gates. The inputs could be used to select a particular address from which to retrieve data.

### Adders

To perform addition on -bit numbers, we use a set of adders such that the carry from one adder is provided as input to the next one.



This implementation however, could cause a large amount of delay for larger sets of adders. The carry lookahead approach solves this by using Boolean algebra to predict what the output for each carry should be based on the initial inputs.

Due to how complicated the process quickly becomes, it is only done with 4 or 8 bits at a time, i.e. the carry is allowed to ripple through sets of 4 or 8 bits.

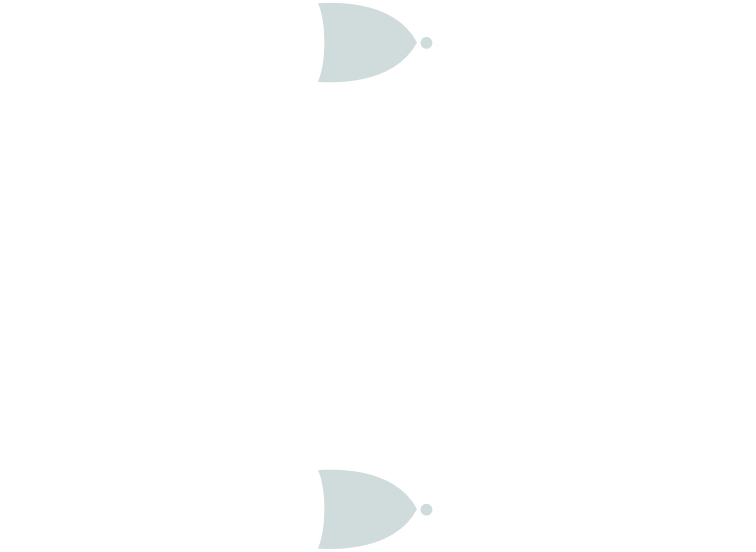
## 11.4 Sequential Circuits

To be able to store state information or data in memory, we require the more complex sequential circuitry, where the current output depends on the current input as well as the past history of inputs or the current state of the circuit.

### Flip-Flops

Flip-flops are the simplest sequential circuits. They exist in one of two states, and in the absence of input, they remain in that state. They also have two outputs, which are always the complement of each other.

A common configuration is the SR flip flop or SR latch.

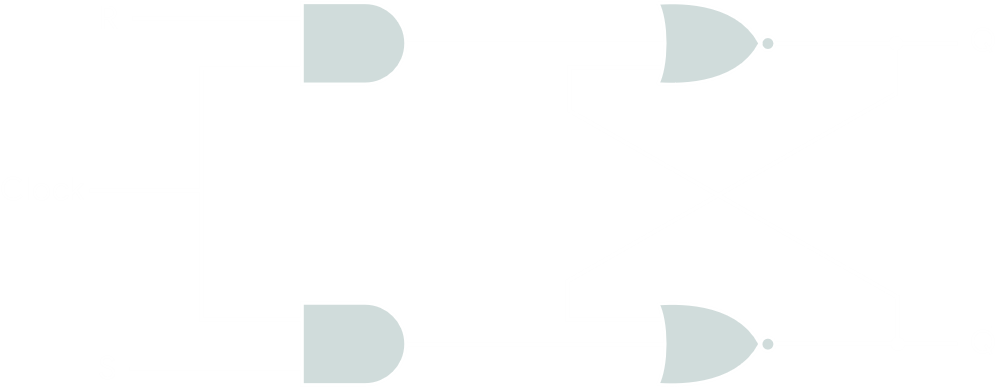


If and are both , the output does not change. This means the circuit can work as a -bit memory. If is switched to , and . This stays stable as well. If alone is set to , the opposite occurs. Note that both and cannot be set to at the same time, since it will cause both outputs to become .

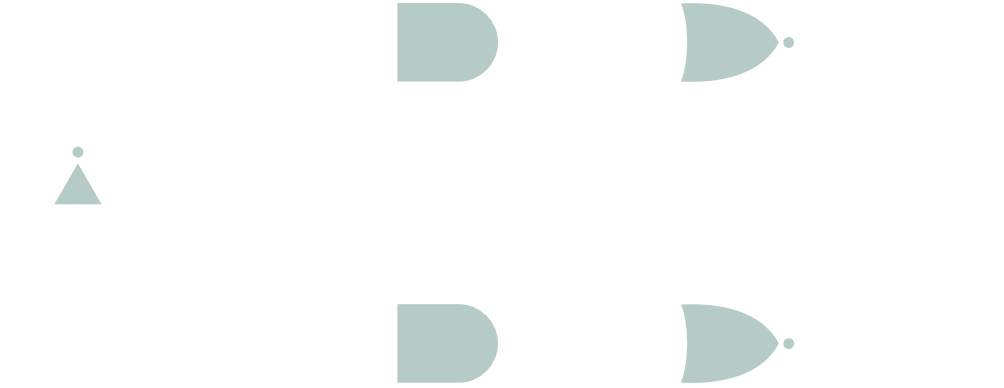
From this, we can derive a characteristics table to show how output changes depending on input and the current state.

|  |  |  |
| --- | --- | --- |
| **Current Inputs** | **Current State** | **Next State** |
|  |  |  |
| 00 | 0 | 0 |
| 00 | 1 | 1 |
| 01 | 0 | 0 |
| 01 | 1 | 0 |
| 10 | 0 | 1 |
| 10 | 1 | 1 |
| 11 | 0 | - |
| 11 | 1 | - |

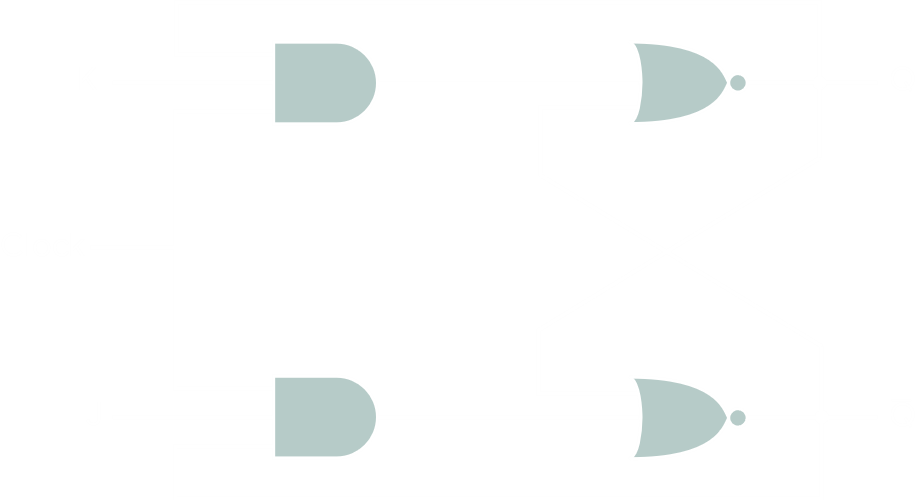
The operation we have seen above is called asynchronous operation. More typically, the input changes only when a clock pulse occurs, thus syncing things up.



D flip flops are the same as SR latches, except that only one input is given and the complement of that input is taken as the second input. This allows us to avoid the - input condition of SR flip flops. The output of a D flip flop is always the most recent input.



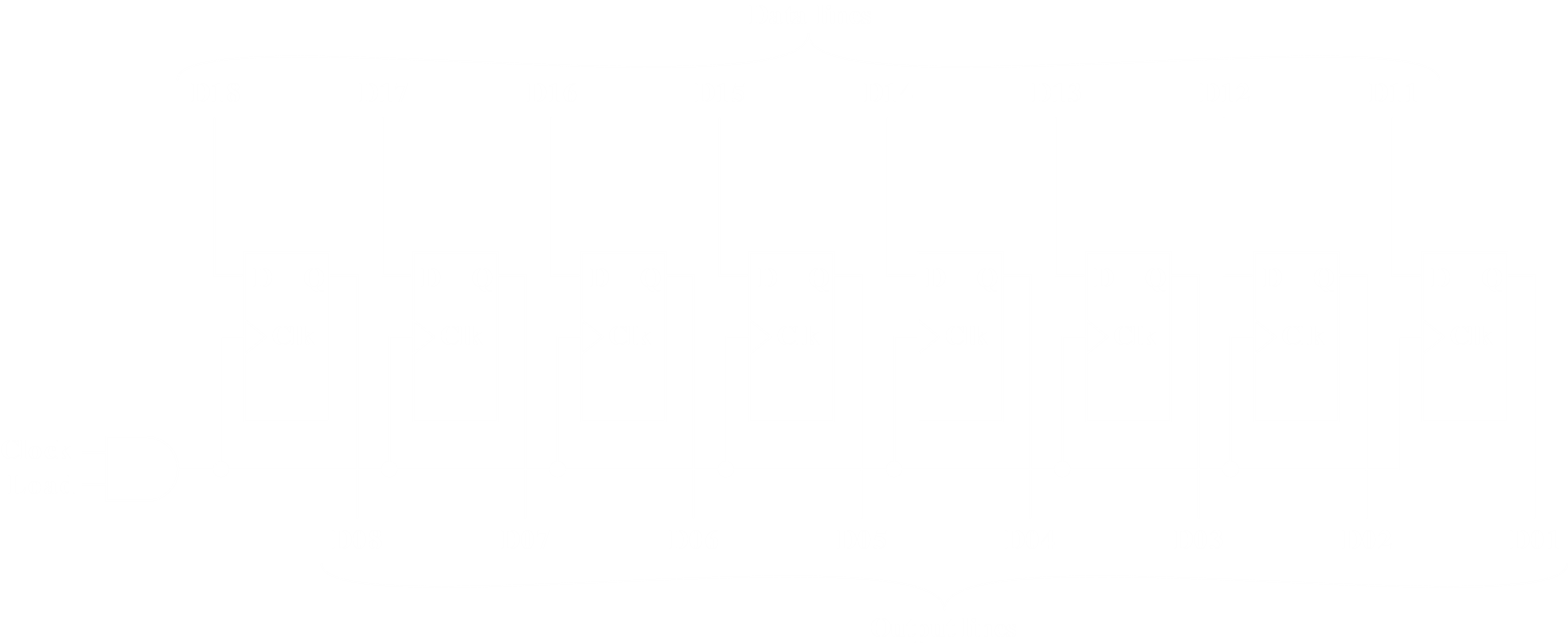
JK flip flops are a little more complicated. If no input is given, the output is stable. If only is set to , the output is . If only is set to , the output is . When both and are set to , the output is flipped.



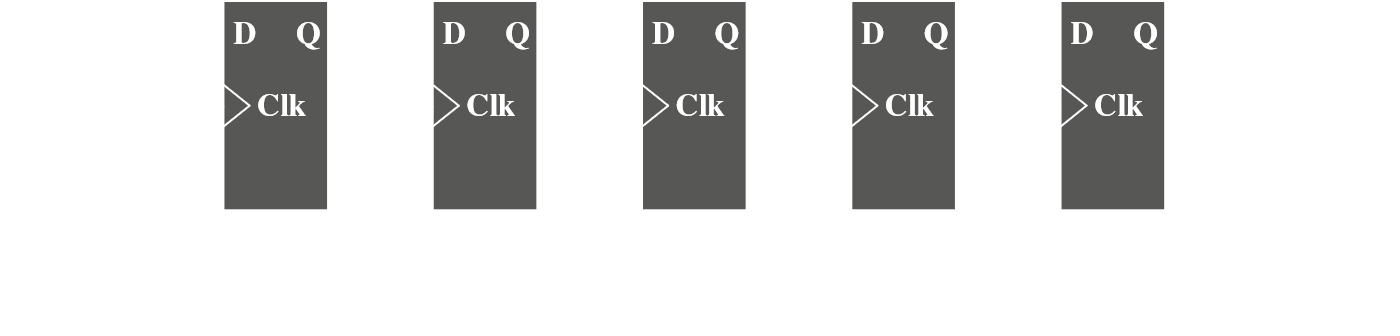
### Registers

Registers are made using flip flops. There are two types of registers, parallel registers and shift registers.

Parallel registers are a set of -bit memories that can be read or written simultaneously.



Shift registers accept and/or transfer information serially. They are used to interface to serial I/O devices. They can also be used within the ALU to perform logic shit and rotate functions.

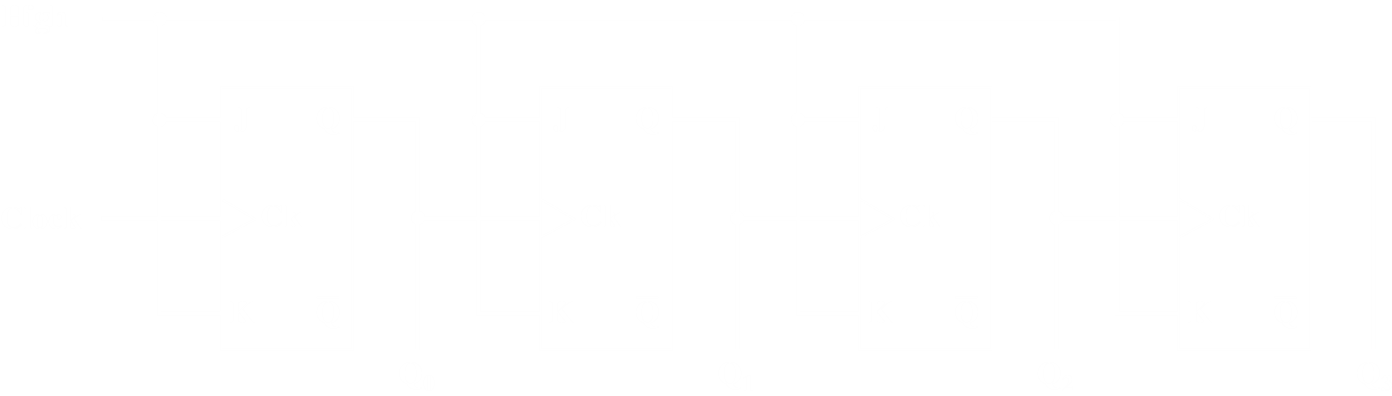


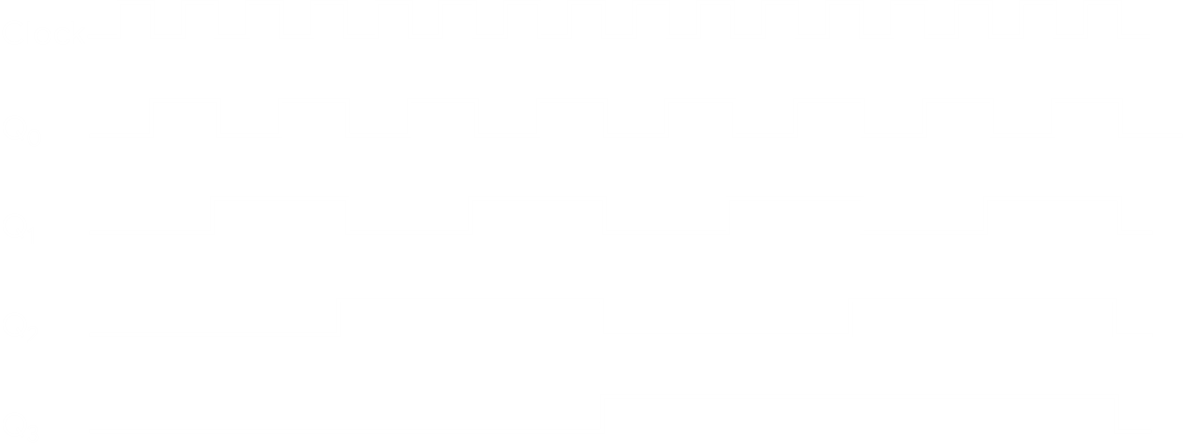
### Counters

A counter is a register whose value is easily incremented by 1 modulo the capacity of the register, meaning the counter returns to 0 and starts again when the maximum value is achieved. The CPU’s program counter works like this.

Counters may be asynchronous or synchronous, with the former being slower since it changes state one flip-flop at a time while the latter changes state all at once. The latter is thus used in CPUs.

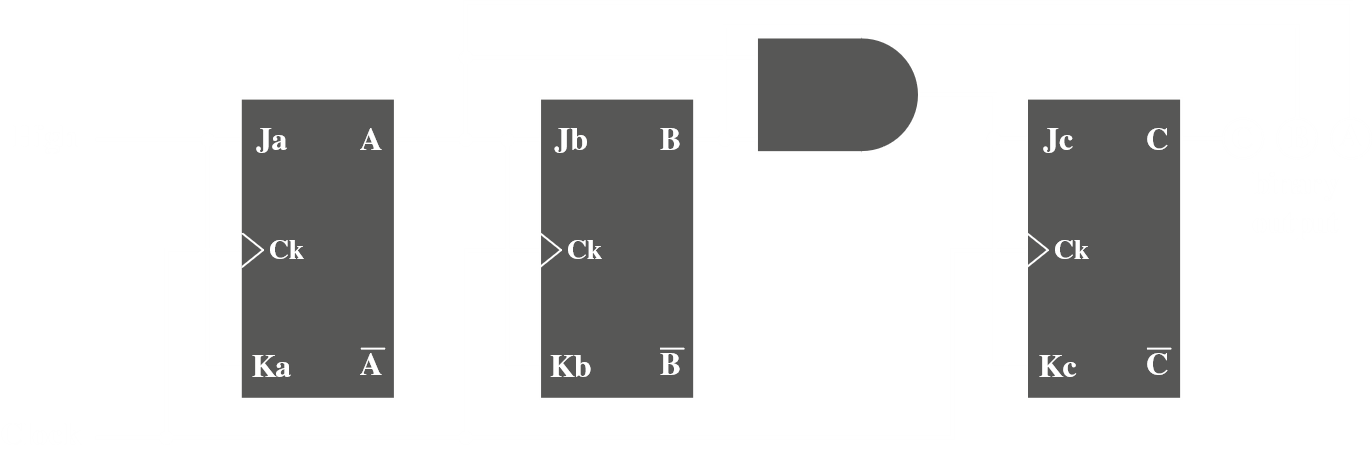
Ripple counters are asynchronous counters for which the change that occurs to increment the counter starts at one end and ripples to the other.





Thus, the output of the leftmost flip flop is the least significant bit. Note that changes occur at the falling edge of the clock-pulse meaning these are edge-triggered flip flops, which provide better timing control.

Ripple counters have delays in changing values, and synchronous counters solve this. They are used in CPUs.



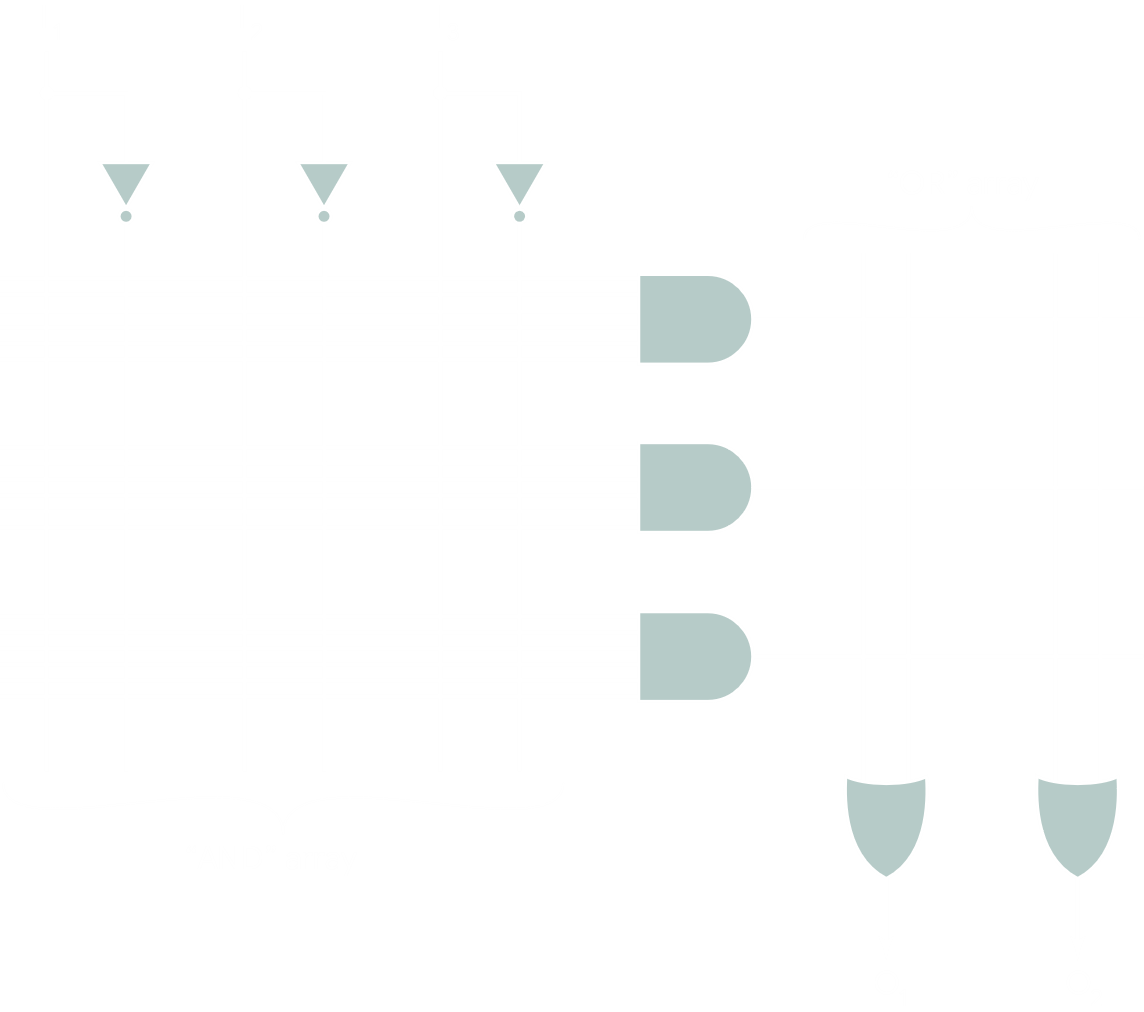
Using an analysis procedure involving the excitation table of JK flip flops, a truth table for the system and Karnaugh maps to simplify it will lead us to the above diagram. Going over it visually also seems to make sense. is toggled by the initial high input. is toggled by and , the most significant bit, is toggled by and together.

## 11.5 Programmable Logic Devices

Even though it eventually became increasingly cheaper to actually produce chips of smaller sizes with larger numbers of gates, the cost of developing custom chips that work for particular logic functions remained high. This is where programmable devices, which are general purpose chips that can be adapted to specific purposes, come in.

### Programmable Logic Array

Any Boolean function can be expressed in a SOP form. The PLA consists of a regular arrangement of NOT, AND and OR gates on a chip. Each input is passed through a NOT gate so every input and its complement is available to each AND gate. The output of each AND gate is available to each OR gate and the output of each OR gate is a chip output. By making the appropriate connections, any SOP expression can be implemented.



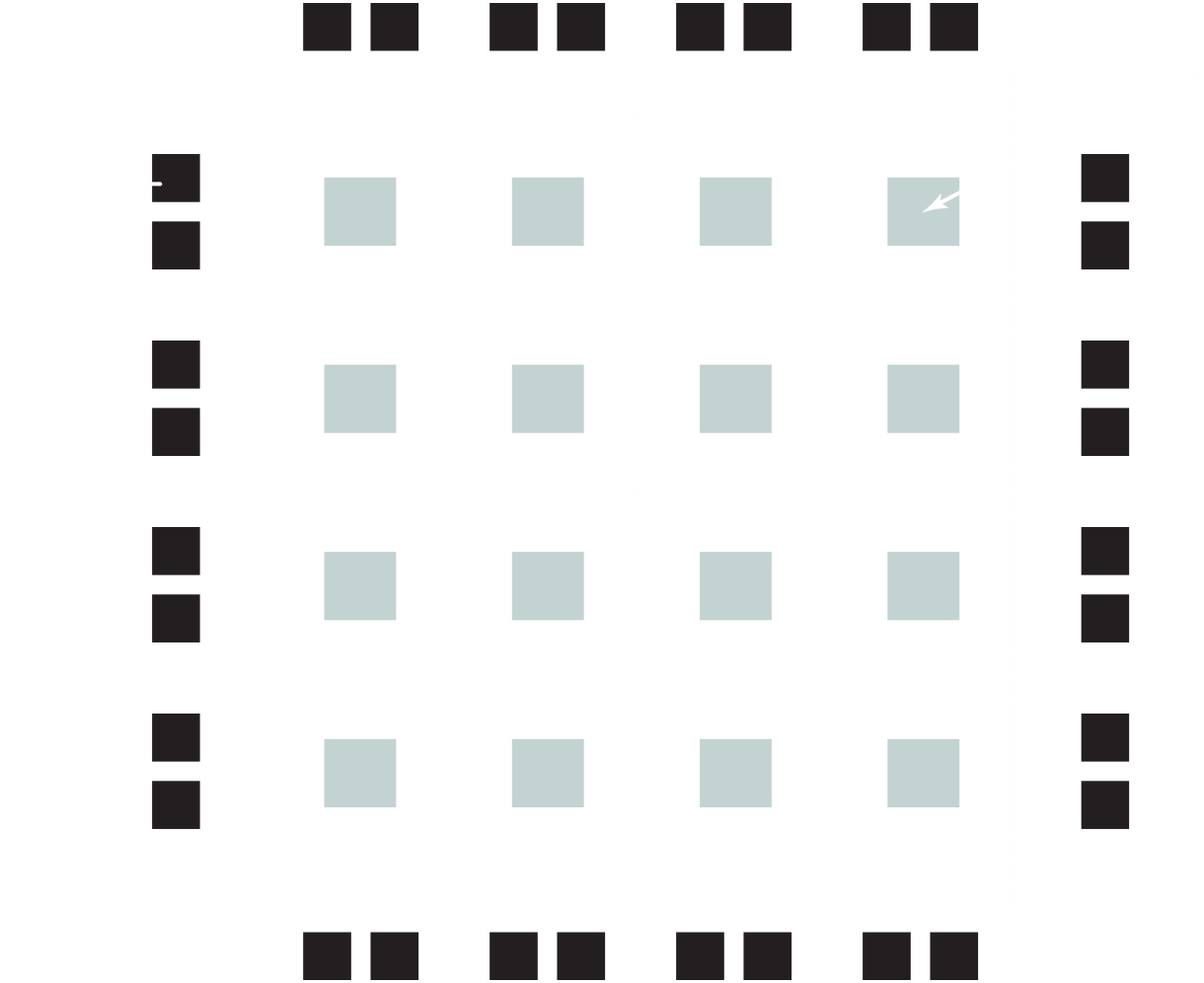
The AND array is programmed by connecting the required lines from the main input and the input to the AND gates at their point of intersection. The OR array is programmed in a similar manner. Most larger PLAs contain several hundred gates and the connections are not specified until programming time.

PLAs are manufactured in two ways. In the first, every possible connection is made with a fuse and the undesired ones are removed by blowing the fuse. This is called a field-programmable logic array (FPLA). The other way is to make the proper connections during chip fabrication using an appropriate mask supplied for a particular interconnection pattern.

### Field-Programmable Gate Array

The PLA is a simple PLD (SPLD). It is difficult to increase the capacity of a SPLD since the size becomes too large too quickly. To provide large capacity devices based on SPLD architecture, we need to integrate multiple SPLDs onto a single chip and provide an interconnect to programmably connect the SPLD blocks together. Such devices are called Complex PLDs (CPLDs). The FPGA is the most important among these.

An FPGA consists of an array of uncommited circuit elements, called logic blocks, and interconnect resources.



* **Logic Block** – The configurable logic blocks are where the computation of the user’s circuit takes place. They use combinational or sequential circuits.
* **I/O Block** – Connect I/O pins to the circuitry on the chip.
* **Interconnect** – Signal paths to establish connections between I/O blocks and logic blocks.