Digital Systems (COE 328): Notes

Adam Szava

Fall 2021

Introduction

This is my compilation of notes from Digital Systems (COE 328) from Ryerson University. All information comes from my professor's lectures, the textbook *Fundamentals of Digital Logic with VHDL*, and online resources.

Chapter 2: Introduction to Logic Circuits

2.1 Variables and Functions

Logic Circuits

This course concerns itself with the study of circuits which can perform a logical operation. This is the basis for all computing. A **logic circuit** is a circuit which contains component which can perform a logical operation.

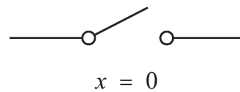
Switches

The most basic component in logic circuits is the switch. A switch is a segment of wire which can be in one of two states: *on*, or *off*, as shown below:

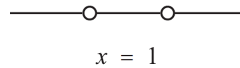
In the *on* state, current can pass through the wire which may complete a circuit. In the *off* state, current cannot pass through the wire which may stop a circuit. Typically, we give the *on* state a value of 1, and the *off* state a value of 0, and we give each switch a variable which can take on one of those two values, as in...

Say S is a switch controlled by x, then:

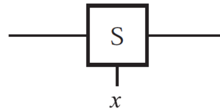
1. If $x = 0$ then the switch is *open* (current cannot pass).



2. If $x = 1$ then the switch is *closed* (current can pass).



The symbol used for a switch controlled by a variable x in a circuit looks like the following:



Switches can be physically constructed using transistors, which will be discussed later on in this course.

Digital Meaning

Note that $x \notin \mathbb{R}$ (x is not a real number), in fact the following is true:

If x is the control variable of a switch, then:

$$x \in \{0, 1\}$$

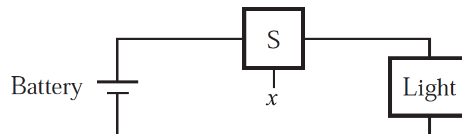
This property of our fundamental control variables being only one of a select few digits at any given time leads to the definition of *digital*. A *digital* system is a complex circuit who's switches can take on one value of a set of digits. In contrast with *analog* systems whose control variables may take on any number in a continuum.

For the purposes of this course, we consider *digital* systems who's switches can take on one of two values (0 and 1) which are called binary systems.

In these kinds of systems 0 has the meaning of *nonexistence/invalidity*, while 1 has the meaning of *existence/validity*.

Functions

Switch control variables can affect other elements of a circuit and their current. In a sense, the state of an element is a *function* of the switch control variables in the circuit. In the following example, the status of the light (on or off) is a function of the state of the switch. We can write this as:

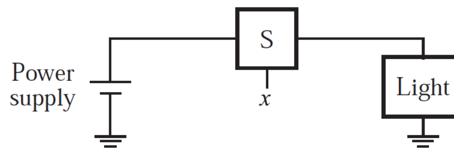


$$L(x) = x$$

... where x is the control variable of the switch, and L is the state of the light. If $x = 1$, then $L(1) = 1$, and so if the switch is closed and the light turns on. We say that $L(x)$ is a **logic function**.

Functions serve as the output of a logic circuit. Note that the functions are **not** defined on the real line, they are defined on $x \in \{0, 1\}$.

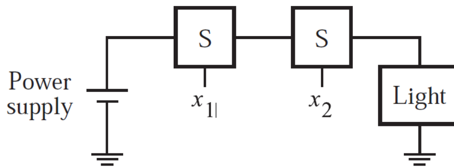
I would also just like to note that the previous circuit can be simplified by putting a ground on the bottom wire, and then redrawing it as the following:



... this is common practice in this course.

AND Operation

Consider the following circuit:



... as you can see the light will only turn on if both switches are in the *closed* state to let the current through. The functions would thus be written as:

$$L(x_1, x_2) = x_1 \ AND \ x_2$$

... we have shorthand for this:

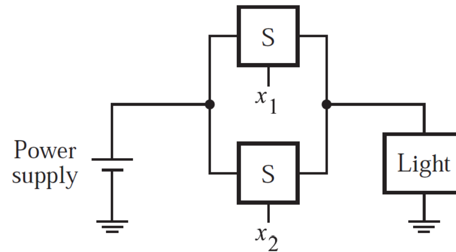
$$L(x_1, x_2) = x_1 \cdot x_2$$

... in fact any multiplication of control variables can be interpreted as the *AND* operation. Multiplication in this context should always be said as: "*and*".

The *AND* operation is the series connections of the switches since if any one of the switches in series are off, then the whole circuit does not complete.

OR Operation

Consider the following circuit:



... as you can see the light would turn on if either the top switch or the bottom switch were in the closed position to let the current through. The functions would thus be written as:

$$L(x_1, x_2) = x_1 \ OR \ x_2$$

... we have shorthand for this:

$$L(x_1, x_2) = x_1 + x_2$$

... in fact any addition of control variables can be interpreted as the *OR* operation. Addition in this context should always be said as: "*or*".

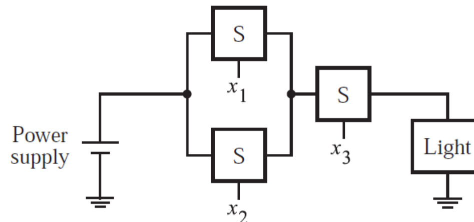
The *or* operation is the parallel connections of the switches since the circuit will be complete if any of the switches are closed.

Combinations of Operations

The *AND* and *OR* operations can be combined into more complex functions, for example the function:

$$L(x) = (x_1 + x_2) \cdot x_3$$

... can be interpreted as the following circuit where x_1 *OR* x_2 and then also x_3 is the logical function:



2.2 Inversion

Consider the following function:

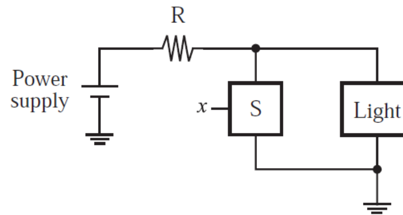
$$L(x) = \begin{cases} 1 & x = 0 \\ 0 & x = 1 \end{cases}$$

... meaning if $x = 1 \implies L(x) = 0$, or if $x = 0 \implies L(x) = 1$. In a sense this function would return to you whatever the opposite of x is. There are many ways to notate this, the following is a few:

$$L(x) = NOT\ x = !x = \bar{x} = \ x$$

The most common one used in the course is $\bar{x}$, and it is usually described as x *compliment*.

The following circuit describes the function:



... notice that if the circuit is open ($x = 0$) then the circuit will be complete and the light will turn on ($L(x) = 1$). If the circuit is closed ($x = 1$) then that branch will create a short circuit and the light will turn off ($L(x) = 1$), which achieves what we want.

With the three operations of *AND*, *OR*, and *NOT*, we have the three basic building blocks of logical circuits.

2.3 Truth Tables

Truth tables are another way to define the *AND*, *OR*, and *NOT* operations. The circuit definition is a good way to get a concrete understanding of that they mean, however you must also have a theoretical understanding.

To begin, lets combine all possible combinations in a table (another word for this is *valuations* of logic values):

x_1	x_2
0	0
0	1
1	0
1	1

Every row represents a possible combination of x_1 and x_2. Generally, n variables have 2^n possible combinations.

We can then add another column which represents what $x_1 \cdot x_2$ and $x_1 + x_2$ would be:

x_1	x_2	$x_1 \cdot x_2$	$x_1 + x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

This is an example of a **truth table**. A truth table is a table where the rows begin with n entries of a possible combination of n input variables, and then the following entries to the row are the original input variables with some logical operation applied to them.

The *AND* and *OR* operations can be extended to taking in n possible input variables, by the following:

- *AND* of n variables is 1 if all $x_1 ... x_n$ are all 1.
- *OR* of n variables is 1 if at least one of $x_1 ... x_n$ is 1.

Example

Consider three input variables x_1, x_2, and x_3, then the corresponding truth table would be (with the extended *AND* and *OR* operations applied):

x_1	x_2	x_3	$x_1 \cdot x_2 \cdot x_3$	$x_1 + x_2 + x_3$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

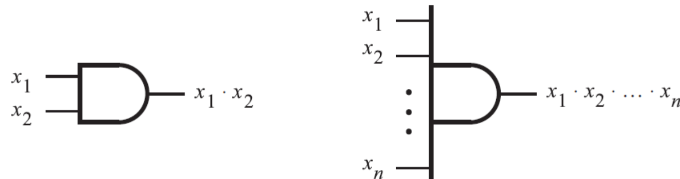
Notice that each row of x_1, x_2, x_3 valuations are the numbers $0, 1, 2, ..., 2^n - 1$ in binary.

2.4 Logic Gates and Networks

Logical operations can be implemented into a circuit using transistors, this circuit is called a *logic gate*. A logic circuit may need dozens of logic gates, and so shorthand notation is used to denote the operations.

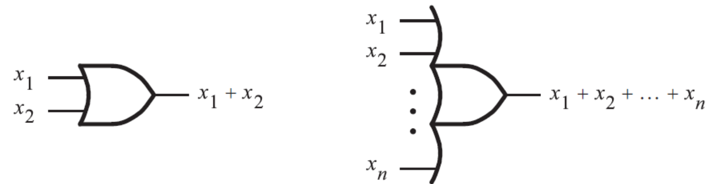
For all of the symbols, some signal or signals are coming in with values of 1 or 0, and out of the gate comes a signal signal which is processed in the appropriate way as 1 or 0.

AND Symbol



On the left, you see the basic *AND* operation, on the right you see the extended *AND* operation which can have n inputs.

OR Symbol



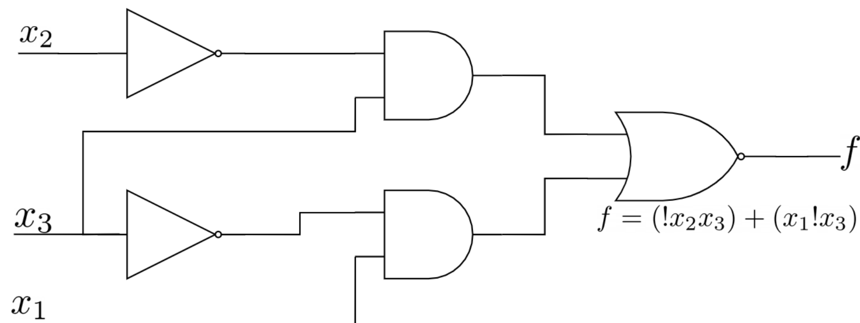
On the left, you see the basic *OR* operation, on the right you see the extended *OR* operation which can have n inputs. It is important to draw the curves base of the gate.

NOT Symbol

A larger circuit is a *network* of gates, also called a *logic network*, or a *logic circuit*.

Example

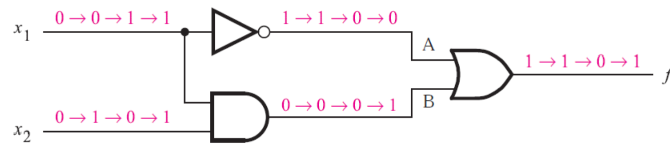
The following circuit contains multiple logic gates, and has an output variable of f (like L with the light examples):



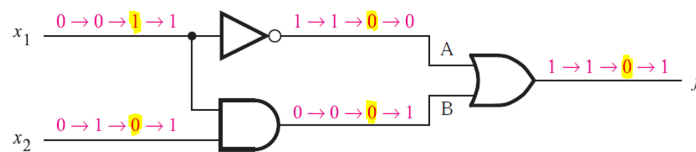
Analysis of a Logic Network

Given some existing logic network, finding the equation which describes its output is called the *analysis* process, which is simple compared to the reverse process of *synthesis*.

To achieve the *analysis* process, you can just write the 2^n possible values along each wire based off of the n starting variable values. For example in the following:



The position of the number in the list matters, so the third position numbers all refer to each other, as in:



This position refers to the fact that:

$$f(1, 0) = 0$$

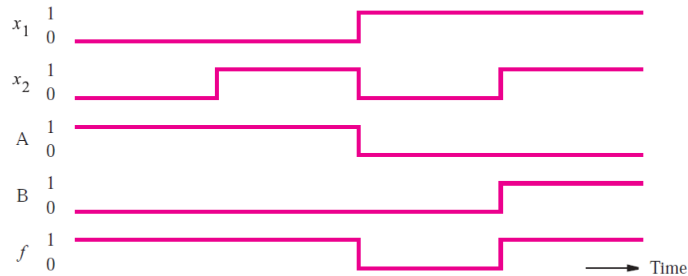
... in our corresponding truth table:

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	1
1	1	1

Note also that intermediate points like A and B in the diagrams also take on some values depending on x_1 and x_2. The *valuations* of x_1 and x_2 are also called our *primary inputs*, while $f(x_1, x_2)$ is called our *primary outputs*. You can think of these as like the pins of the chip.

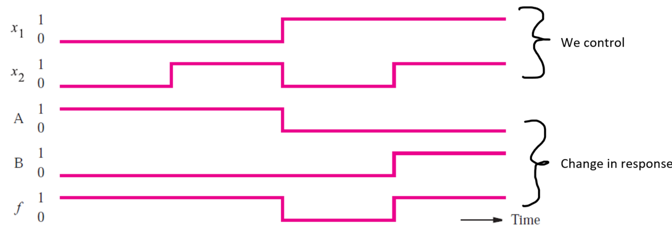
Timing Diagrams

A timing diagram summarizes the same information as a truth table, but now there is an element of time. Each row represents a variable which can switch between its two states. The first two rows are our primary input variables and so you can think of it like we are in control of those two, while the rest just change in response.



You can think of it as the logic network begins in the state with primary input as $x_1 = 0 = x_2$ and the remainder of the lines below represent the corresponding states of the variables. As time passes we switch the values of the first two rows and the remaining values also change.

Note that changes are assumed to be happening instantaneously in an ideal gate. This however is not true in practical applications however we will use a timing diagram to solve issues with timing in later chapters.



2.5 Boolean Algebra

Boolean Algebra gives us a framework of mathematical rules that allows us to *minimize* logic circuits, meaning convert our logic functions into simpler ones to reduce cost. All mathematical frameworks begin with a list of axioms taken as true by default, *Boolean Algebra* is no different:

1. Define two operations + and $\cdot$ by the following table.

Axioms
$1a$ $0 \cdot 0 = 0$
$1b$ $1 + 1 = 1$
$2a$ $1 \cdot 1 = 1$
$2b$ $0 + 0 = 0$
$3a$ $0 \cdot 1 = 1 \cdot 0 = 0$
$3b$ $0 + 1 = 1 + 0 = 1$
$4a$ If $x = 0$, then $\bar{x} = 1$
$4a$ If $x = 1$, then $\bar{x} = 0$

In this algebra, we only define the numbers 1 and 0 and their relations to each other. While considering boolean algebra it is very convenient to consider 1 as

true and 0 as *false*. This gives more meaning to $1 \cdot 1 = 1$ which can be read as *true and true is true*, or $0 \cdot 1 = 0$ which can be read as *false and true* is false.

The following list of theorems can be easily proven from the axioms.

Single Variable Theorems
$5a\ x \cdot 0 = 0$
$5b\ x + 1 = 1$
$6a\ x \cdot 1 = x$
$6b\ x + 0 = x$
$7a\ x \cdot x = x$
$7b\ x + x = x$
$8a\ x \cdot \bar{x} = 0$
$8b\ x + \bar{x} = 1$
$9\ \bar{\bar{x}} = x$

The following list of theorems can also be proven from the axioms, although this now goes without saying.

Multi-variable Theorems
$10a\ x \cdot y = y \cdot x$
$10b\ x + y = y + x$
$11a\ x \cdot (y \cdot z) = (x \cdot y) \cdot z$
$11b\ x + (y + z) = (x + y) + z$
$12a\ x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
$12a\ x + (y \cdot z) = (x + y) \cdot (x + z)$
$13a\ x + (x \cdot y) = x$
$13b\ x \cdot (x + y) = x$
$14a\ (x \cdot y) + (x \cdot \bar{y} = x$
$14b\ (x + y) \cdot (x + \bar{y}) = x$
$15a\ !(x \cdot y) = \bar{x} + \bar{y}$
$15b\ !(x + y) = \bar{x} \cdot \bar{y}$
$16a\ x + (\bar{x} \cdot y) = x + y$
$16b\ x \cdot (\bar{x} + y) = x \cdot y$
$17a\ (x \cdot y) + (y \cdot z) + (\bar{x} \cdot z) = (x \cdot y) + (\bar{x} \cdot z)$
$17b\ (x + y) \cdot (y + z) \cdot (\bar{x} + z) = (x + y) \cdot (\bar{x} + z)$

Another important concept is the *dual* of a statement. This is obtained by replacing all $\cdot$ with $+$ and vice-versa, along with replacing all 0 with 1 and vice versa. The dual of a true statement is also a true statement in boolean algebra.

Parenthesis can be used to indicate precedence of operations, and in fact should always be used. The custom is that in the absence of parenthesis, the order is NOT, AND, OR.

2.6 Synthesis

The process of creating a function to model predetermined output behavior is called *synthesis*.

Sum-of-Products (SOP)

Given some truth table, the *minterms* are product terms (terms made up of a product) in which each variable appears, either as x_i if it's value is 1 or $\bar{x}_i$ if its value is 0. We refer to the nth row's minterm of a truth table as m_n.

Given:

Row number	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

A function can be *synthesized* using the *Sum-of-Products Method* by taking the logical sum of all the minterms which correspond to a functional output of 1. This is essentially saying *either this case, or this case, or this case, ... , or this case make the function output 1.* This gives you a function with the correct output and you can then simplify using Boolean Algebra.

Row number	x_1	x_2	x_3	Minterm	Maxterm
0	0	0	0	$m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$	$M_0 = x_1 + x_2 + x_3$
1	0	0	1	$m_1 = \overline{x}_1\overline{x}_2x_3$	$M_1 = x_1 + x_2 + \overline{x}_3$
2	0	1	0	$m_2 = \overline{x}_1x_2\overline{x}_3$	$M_2 = x_1 + \overline{x}_2 + x_3$
3	0	1	1	$m_3 = \overline{x}_1x_2x_3$	$M_3 = x_1 + \overline{x}_2 + \overline{x}_3$
4	1	0	0	$m_4 = x_1\overline{x}_2\overline{x}_3$	$M_4 = \overline{x}_1 + x_2 + x_3$
5	1	0	1	$m_5 = x_1\overline{x}_2x_3$	$M_5 = \overline{x}_1 + x_2 + \overline{x}_3$
6	1	1	0	$m_6 = x_1x_2\overline{x}_3$	$M_6 = \overline{x}_1 + \overline{x}_2 + x_3$
7	1	1	1	$m_7 = x_1x_2x_3$	$M_7 = \overline{x}_1 + \overline{x}_2 + \overline{x}_3$

In our example, you would sum the *minterms* for row $1, 4, 5,$ and 6, this can be written as:

$$f(x_1, x_2, x_3) = \sum(m_1, m_4, m_5, m_6) = \sum m(1, 4, 5, 6)$$

This is why it's called a sum of products, since each minterm is a product of variables.

In general the sum of products method can be defined as:

$$f(x_1, x_2, \dots, x_n) = \sum m(i_1, i_2, \dots i_k)$$

... where m_n denotes the minterm of row n given that $f(x_1, x_2, \dots x_1) = 1$ for the valuation on row n.

Product-of-Sums (POS)

A function's logical inverse (it's dual) can be synthesized using the same *Sum-of-Products* method but only choosing the rows where the function is equal to 0. You can then complement this new function to find f since:

$$\bar{\bar{f}} = f$$

This can be done more practically using the *Product-of-Sums method* where,

$$\bar{m_j} = M_j$$

... where M_j is called a maxterm. Recall *DeMorgan's Theorem* while solving for the complements. f is then equal to a product of all the max terms corresponding to all the rows where $f = 0$. To do this just sum together all the variables in the valuations as x_i if it is 0 and as $\bar{x_i}$ is it is 1 (opposite of the previous method). Take the logical product of all those rows and you get another function which represents f. This can also be minimized.

In general the product of sums method can be defined as:

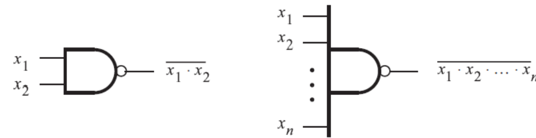
$$f(x_1, x_2, \dots, x_n) = \prod M(i_1, i_2, \dots i_k)$$

... where M_n denotes the maxterm of row n given that $f(x_1, x_2, \dots x_1) = 0$ for the valuation on row n.

2.7 NAND and NOR Gates

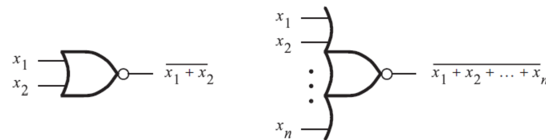
NAND Gate

The NAND gate is the complement to the AND gate:



NOR Gate

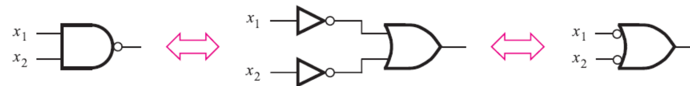
The NOR gate is the complement to the OR gate:



AND/OR Conversion

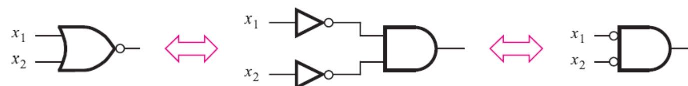
You can convert a NAND gate into an OR gate with both inputs inverted because of DeMorgan's theorem:

$$!(x \cdot y) = \bar{x} + \bar{y}$$



You can also convert a NOR gate into an AND gate with both inputs inverted because of DeMorgan's theorem:

$$!(x + y) = \bar{x} \cdot \bar{y}$$



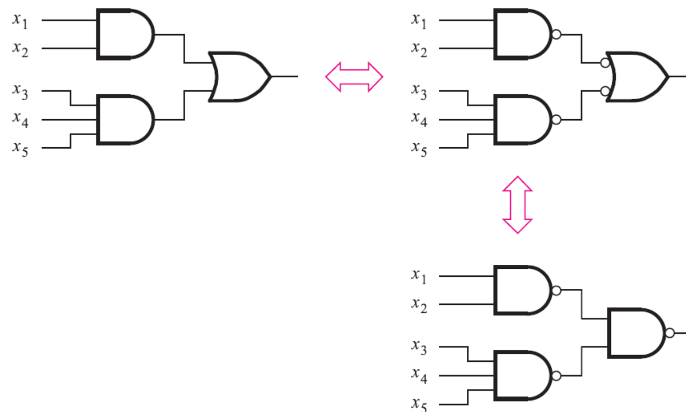
Synthesis using NAND and NOR

We will learn in *Chapter 3* that NAND and NOR gates are cheaper to implement than AND and OR gates and so it is attractive to covert all AND/OR gates into NAND/NOR gates.

Remember that you are allowed to double invert a segment of wire since:

$$\bar{\bar{x}} = x$$

By this logic, and what we just learned about the NAND and NOR gates you can make a simplification like the following:

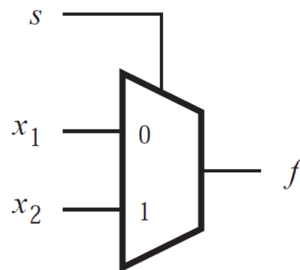


2.8 Design Examples

Multiplexers

A multiplexer is a circuit element which takes in two input signals (x_1, x_2), and exactly outputs one of the two input signals depending on a third control input signal (s).

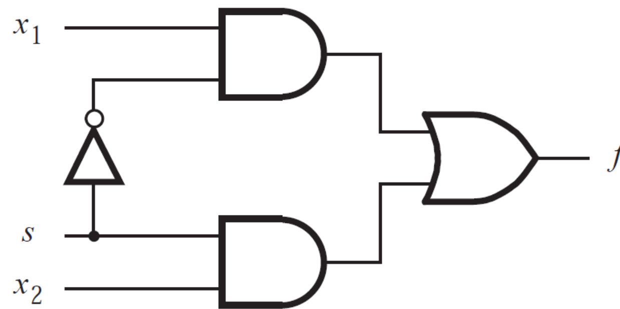
The following is a symbol to denote the circuit described above:



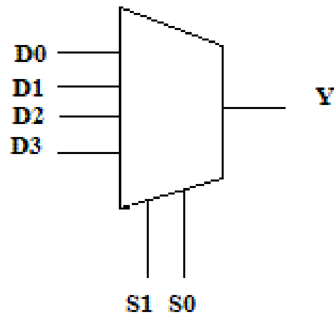
If $s = 0$, then $f = x_1$. If $s = 1$, then $f = x_2$. This can be summarized in the following truth table:

$s\ x_1\ x_2$	$f(s, x_1, x_2)$
0 0 0	0
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	1

The circuit can be implemented using the POS or SOP methods, and then minimized to obtain the following circuit:



The previous examples were of $2 - to - 1$ multiplexers. This is a multiplexer which converts two signals into one, by selecting one from a control signal. A $4 - to - 1$ multiplexer would require two control inputs s_1 and s_2 since they would together have 4 states, and so each state represents a selection of one of the inputs. Such as the following image:



In general, if you have p inputs to your multiplexer you will need at least the smallest number n of inputs which satisfies:

$$n \geq \log_2(p)$$

2.9 Introduction to CAD Tools

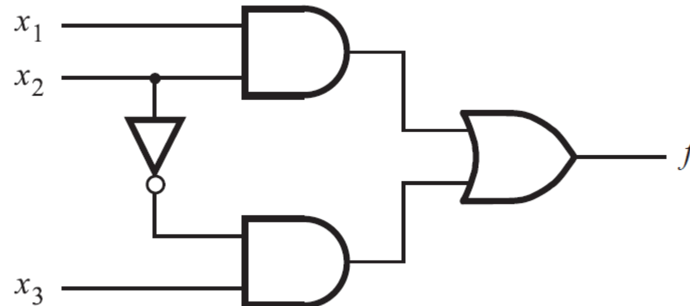
Logic circuits can be made in a CAD tool by the method of *schematic design*, which is where the circuit is drawn in the CAD tool with boxes of abstraction. It can also be made by the aide of a *hardware description language* (HLD) which is a programming language to describe hardware. The two most common HDL languages are:

1. *Very High Speed Integrated Circuit Hardware Description Language* (VHDL)
2. *Verilog HDL*

This course uses VHDL. *Functional simulation* is the process of simulating the outputs of a logic function by outputting a timing diagram to ensure the results are accurate. Timings are assumed to be instantaneous. *Propagation delay* of a circuit is the actual time it takes for change to occur in the output given a change in the input. A *timing simulator* takes into account the timing constraints of each component in the circuit while simulating the outputs.

2.10 Introduction to VHDL

Let's say you want to code the following circuit using VHDL:



Then the code would be:

```
ENTITY example1 IS
    PORT ( x1, x2, x3  : IN    BIT ;
           f           : OUT   BIT ) ;
END example1 ;

ARCHITECTURE LogicFunc OF example1 IS
BEGIN
    f <= (x1 AND x2) OR (NOT x2 AND x3) ;
END LogicFunc ;
```

Let's understand this line by line:

1. ENTITY is the keyword which means that *example1* is a circuit.
2. PORT refers to the inputs and the outputs from the circuit, namely x1, x2, x3 which are all of type BIT. Bit is a data type which can take on a value of 1 or 0.
3. f is an output variable of type BIT.
4. END the framework of *example1*
5. ARCHITECTURE is a keyword which says that *LogicFunc* is a set of logical functions which pertain to the inputs of *example1* which IS:
6. BEGINS starts the logic definition.
7. f is assigned (<=) the logical operation. Parenthesis are *required.*
8. END the definition of the logical function *LogicFunc.*

Chapter 3: Implementation Technology

We begin this chapter with a study of transistors. Switches are implemented into circuits by the means of either voltage or current control. Both are used but we will focus on voltage control.

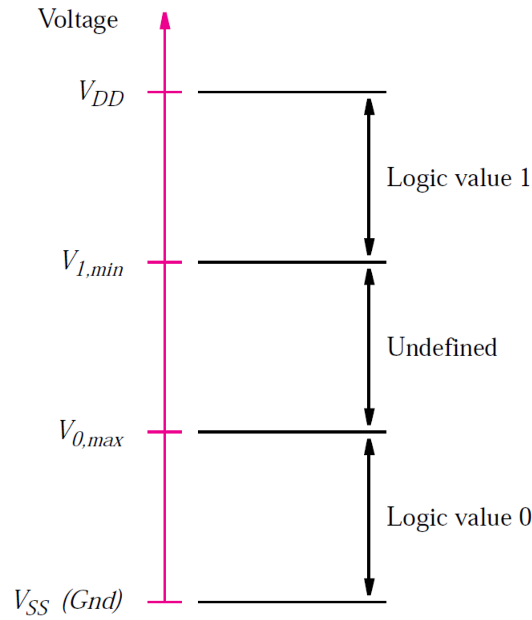
In a positive logic system, a low voltage is a logical 0, while a high voltage is a logical 1. The opposite is true in a negative logic system.

1. Any voltage below some threshold is equivalent to a logical 0, this threshold is denoted ($V_{0,max}$). The voltage must always be above 0 which is denoted V_{SS} and is the lower limit of logical 0.

2. Any voltage above some threshold is equivalent to a logical 1, this threshold is denoted ($V_{1,min}$). The voltage must always be below some upper bound (typically $5V$) which is denoted V_{DD}

3. Typically $V_{0,max} \neq V_{1,min}$. In the range $[V_{0,max}, V_{1,min}]$ the logical value is *undefined.*

The following image summarizes this:



Typically:

1. $V_{1,min} \approx 0.6V_{DD}$

2. $V_{0,max} \approx 0.4V_{DD}$

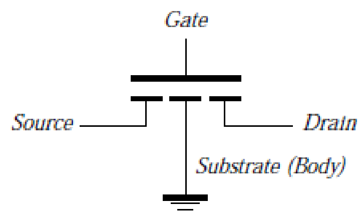
3.1 Transistor Switches

The most popular type of transistor is the *metal oxide semiconductor field-effect transistor* (MOSFET). There are two subtypes of MOSFETs:

1. *n-channel* (NMOS)
2. *p-channel* (PMOS)

3.2 NMOS Transistors

A NMOS transistor is illustrated below:

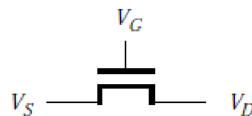


It has *four* electrical terminals called the:

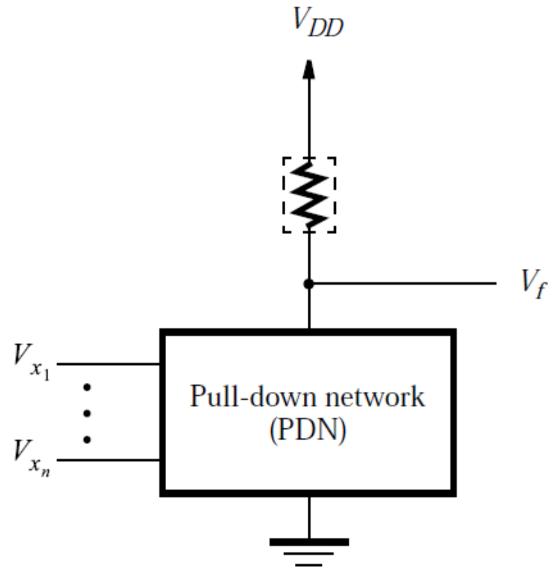
1. Source is the terminal with the lower voltage)
2. Drain
3. Gate is where V_G is applied to control the transistor.
4. Substrate (also called body) is connected to ground. This is typically omitted.

In a NMOS transistor, if V_G is low then there is no connection between the source and the drain terminals, we say the NMOS is *turned off*. If V_G is high then it acts as a closed switch and connects the source and the drain terminals, we say the NMOS is *turned on*. For now assume that there is 0Ω of resistance between the source and the drain. This is summarized in the image below:

The simplified electronic symbol for the NMOS transistor is:

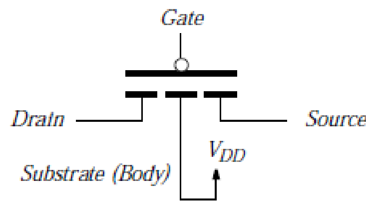


When the *NMOS* transistor is turned on, its drain voltage is pulled down to ground. This gets the name *Pull-Down Network*, illustrated in the following image:



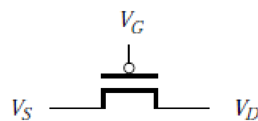
PMOS Transistors

PMOS transistors work essentially opposite to NMOS transistors. It is illustrated below:



The voltage and switch position relation is as below:

The simplified circuit diagram for the PMOS transistor is:

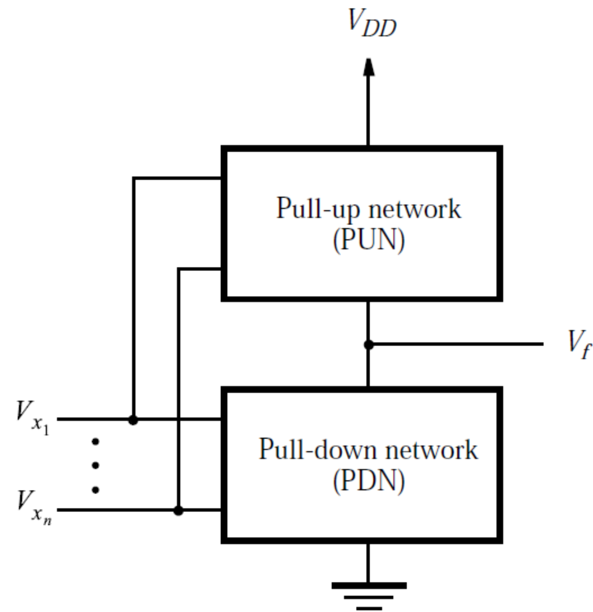


As you can see the symbol indicates that the output of the PMOS is the inverse of the output of the NMOS transistor.

When the *PMOS* transistor is turned on, its drain voltage is pulled up to V_{DD}. This gets the name *Pull-Up Network*.

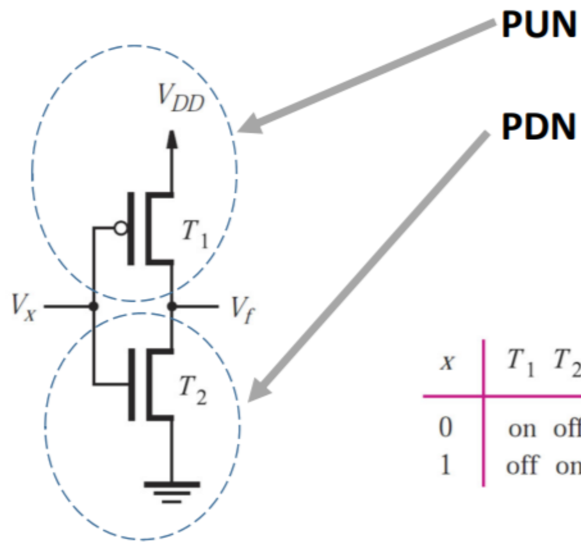
3.3 CMOS Circuits

Complementary MOS or CMOS circuits use both NMOS and PMOS transistors to implement a function. Say V_f is a voltage at a node which represents the output of the function. Then the following general circuit can represent any CMOS implementation of the function f:



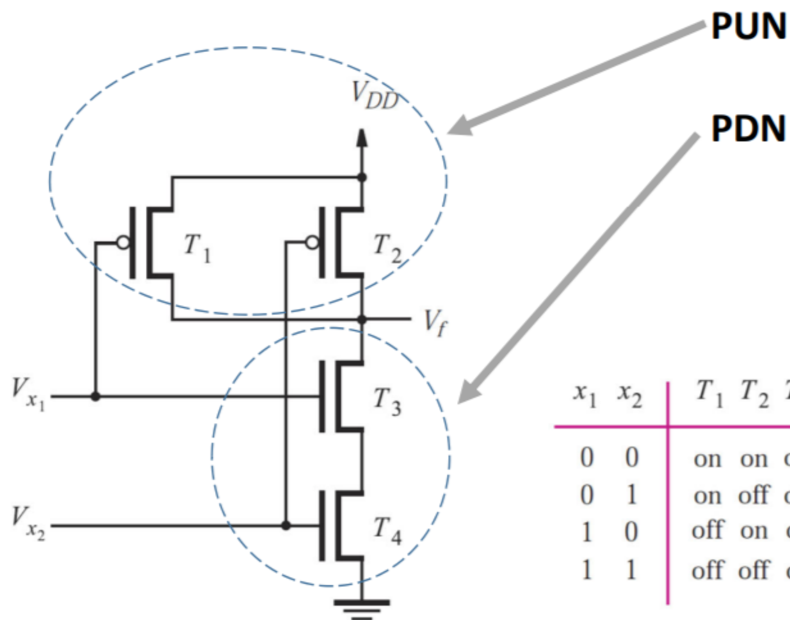
Where the pull-up network implements the function f using switches, and the pull down network implements the function $\bar{f}$ using switches. This is because if the output should be a logical 1, then the output voltage should be brought up to V_{DD} by creating a short circuit between V_f and V_{DD}. If the output should be a logical 0, then the output voltage should be brought down to 0 by creating a short circuit between V_f and ground.

The following two images are examples of CMOS implementation of a function f:



x	T_1	T_2	f
0	on	off	1
1	off	on	0

$$f = \bar{x}$$

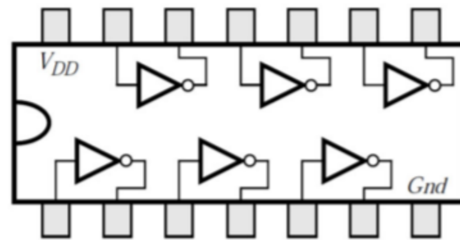


x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	1
1	0	off	on	on	off	1
1	1	off	off	on	on	0

$$f = !(x_1 x_2)$$

3.5 Standard Chips

Standard chips are fabricated circuit elements which contains pins. Each pin corresponds to an input into a circuit. Take for example the *7404 chip*:



(b) Structure of 7404 chip

In this chip, not gates have been implemented into it, and so after connecting the V_{DD} and ground to a power source, the pins will function as not gates.

You could implement an entire function f using standard chips in the following way:

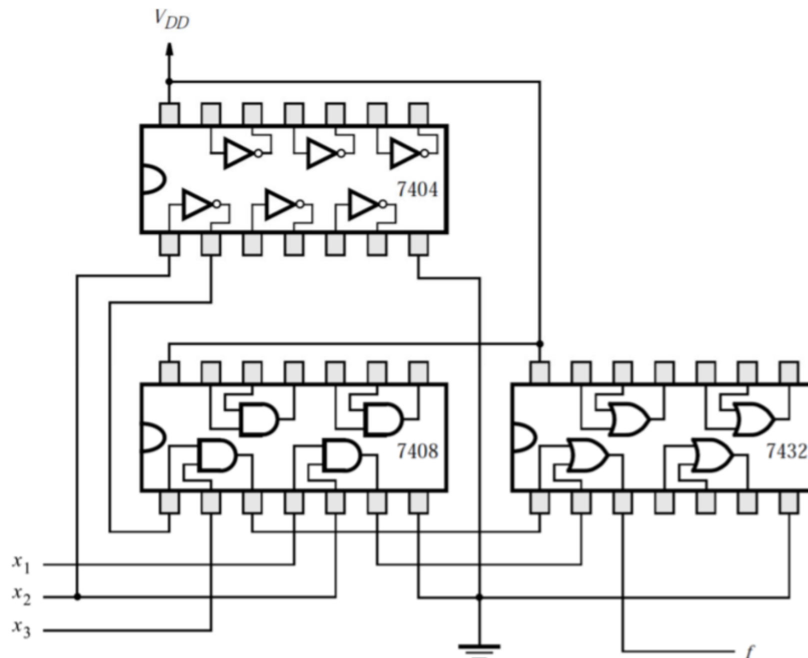
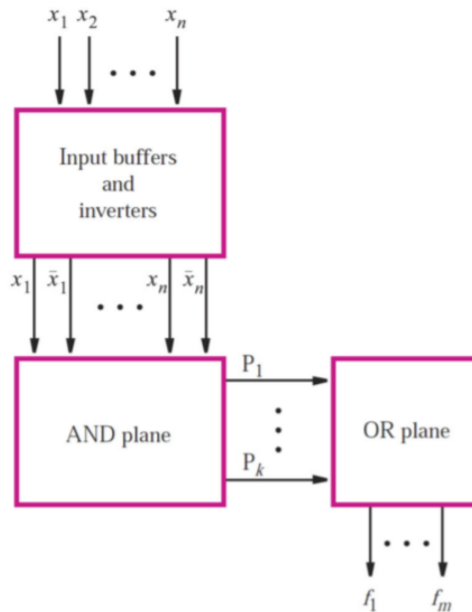


Figure 3.22 An implementation of $f = x_1x_2 + \overline{x}_2x_3$.

3.6 Programmable Logic Devices

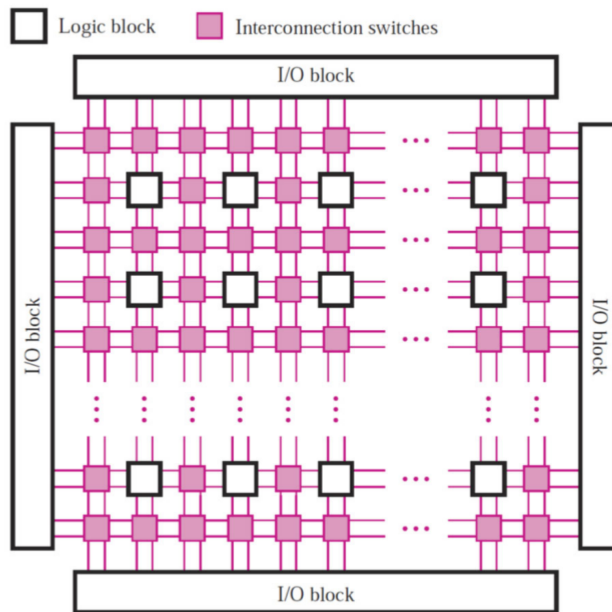
A programmable logic array is a method to implement the SOP form of a logic function. Logically, to implement that kind of function you need to invert some signals, AND some signals, and then OR some signals. This can be illustrated by the following image:



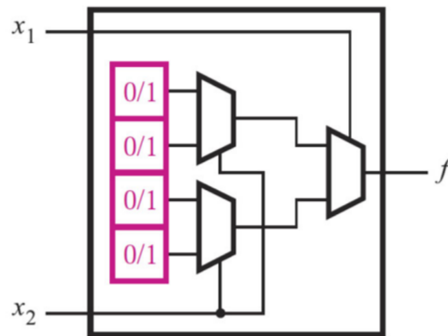
The reason this type of chip is programmable is because you can program exactly which signals go from the first box to the second, and second box to the third, implementing whatever function you want.

3.6.5 Field-Programmable Gate Arrays

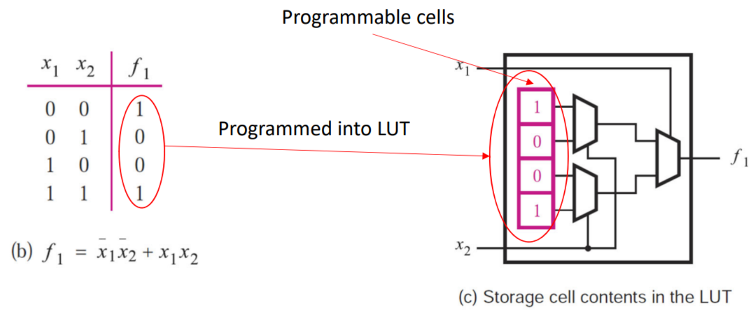
FPGAs are programmable logic devices which support implementation of large logic circuits. They do not contain AND or OR planes, instead they are comprised of logic blocks and interconnection switches between the logic blocks.



In a logic block, truth tables are implemented using the following circuit:

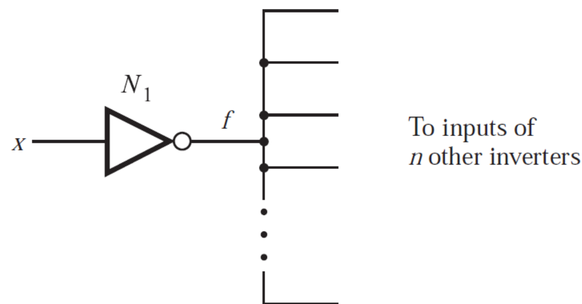


Where each 0/1 block either outputs a constant 1 or 0 and can be programmed to match the output of the function in the truth table. This function is implemented with now AND/OR gates, and only multiplexers. These are called look up tables(LUT), and are used for 2-4 variable functions. More than 2 variables requires subfunctions. The following is an implementation of a function using a LUT:

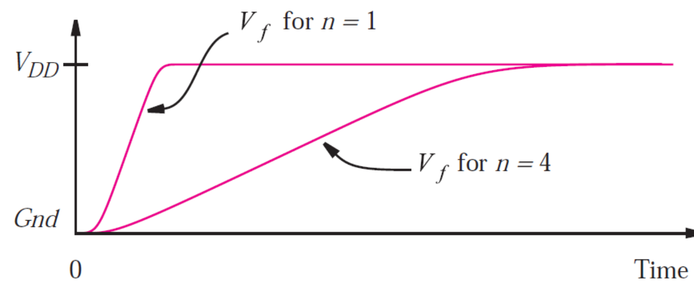


Fan-out and Timing Delays

In real logic gates, one may be required to suppoy voltage to many other gates, like the following image:



The signal strength may weaken and the time for the logic value to change from $1 \to 0$ or $0 \to 1$ will increase, as you can see in the following graph:



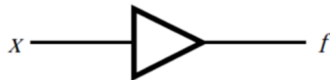
We say that the voltage from the NOT gate is driving n gates and is *fanning out*. So that is causing a *timing delay*.

Buffer

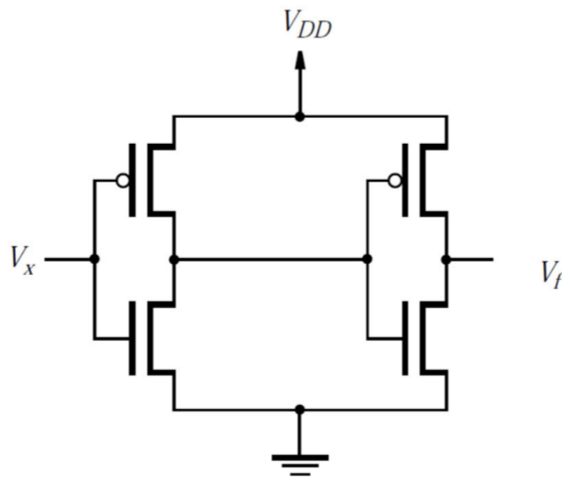
A buffer is a circuit element used to improve timing performance. A non-inverting buffer is a logic gate which implements the function:

$$f = x$$

... and is depicted by the following circuit diagram:



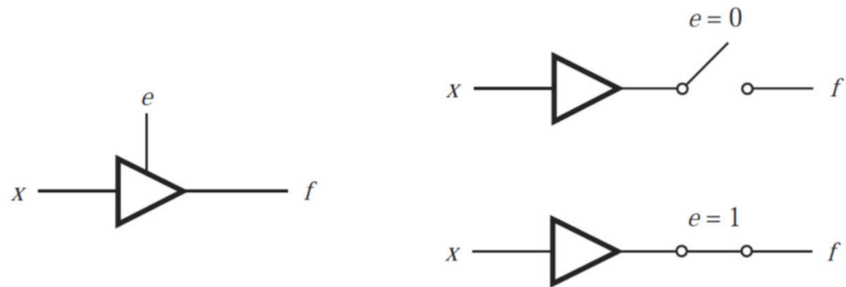
The point of the buffer is to make the voltage as close to V_{DD} or $0V$ as possible depending on the input voltage which may be weakening. The CMOS implementation of the non-inverting buffer is as follows:



Tri-state Buffers

A tri-state buffer is a buffer whose current can be enabled or disabled by a third input current e for *enable*.

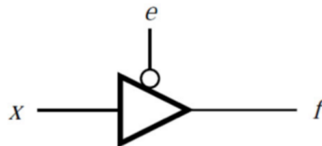
The following diagrams illustrate the circuit symbol and the enable/disable logic:



The truth table for the tri-state buffer is:

e	x	f
0	0	Z
0	1	Z
1	0	0
1	1	1

Z represents high impedance and has no logical value. We say that the previous tri-state buffer is high activated since a logical 1 enables the current. The following circuit is a low activated tri-state buffer meaning a logical 0 enables the current:



XOR and XNOR Gates

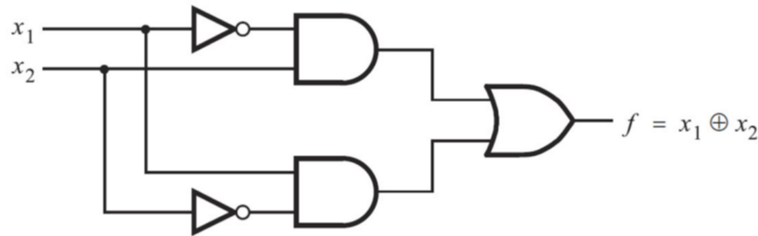
The exclusive OR gate or XOR for short is a complex gate that essentially returns 1 if either input is 1 but **not** if both inputs are 1. In a truth table that is:

x_1	x_2	$f = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

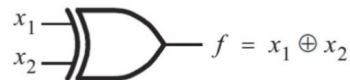
As a function that is:

$$f = \bar{x_1}x_2 + x_1\bar{x_2} = x_1 \oplus x_2$$

The circuit implementation of this is:



... and it's circuit symbol is:



Chapter 4: Optimized Implementation of Logic Functions

4.1 Karnaugh Maps

Karnaugh maps (K-maps) are diagrams used to implement a function based off of it's minterms. Generally, we use the K-map method for functions of $2, 3, 4,$ or 5 variables. Above that the method becomes too confusing to draw.

The method takes advantage of the theorem:

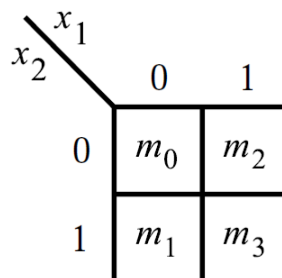
$$x \cdot y + x \cdot \bar{y} = x$$

As you can see, in a sum of minterms in which some variables stay the same (x) and one other differs by its sign (y), we can write this sum as just the variables which stay constant.

Given some function of 2 variables, it's K-map is defined as the following from its truth table:

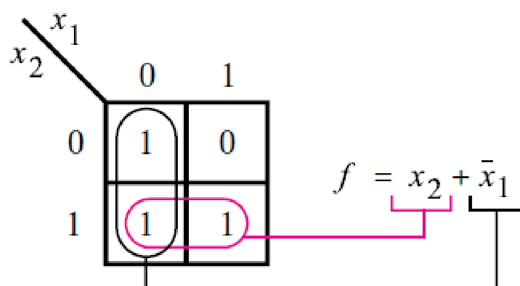
x_1	x_2	
0	0	m_0
0	1	m_1
1	0	m_2
1	1	m_3

(a) Truth table



(b) Karnaugh map

Then, given some specific valuations of the minterms, you can form groups of minterms (evalutating to 1). Groups must be horizontal or vertical groupings of 2^n minterms. This is essentially doing SOP but with multiple terms at the same time. From each grouping, you gain a term equal to the variable which stays constant in that term. The following is an example of this process:



x_2 is a term in our function representation since in the pink group, x_2 stays constant at 1. $\bar{x_1}$ is another term in our sum since in the black grouping $\bar{x}$ stays constant at 0. We do not consider variables which change sign as they will cancel out from the theorem mentioned before.

Notice that groups can overlap each other, this is thanks to the theorem:

$$x + x = x$$

For a **three variable K-map**, the following image is how it's defined. Notice that the minterms are **not** in order. This is because the last two columns have to be switched. This is because it is important that between any two adjacent boxes, only one valuation changes between the variables. If two were to change, then we could not apply the previous theorem. You want the groups to be as large as possible in all situation, but must be in the form 2^n.

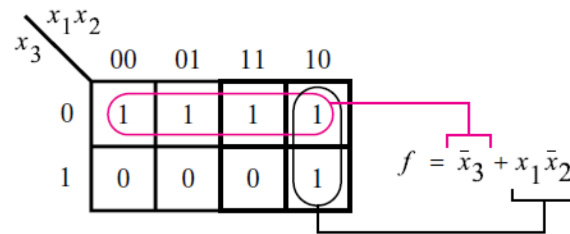
x_1	x_2	x_3	
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

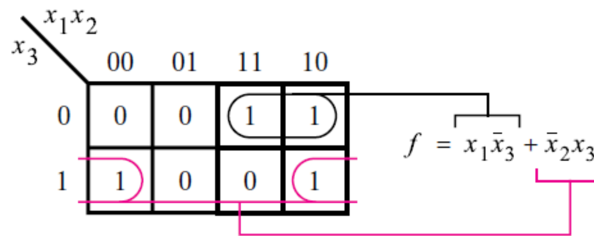
(a) Truth table

x_3 \ x_1x_2	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

(b) Karnaugh map

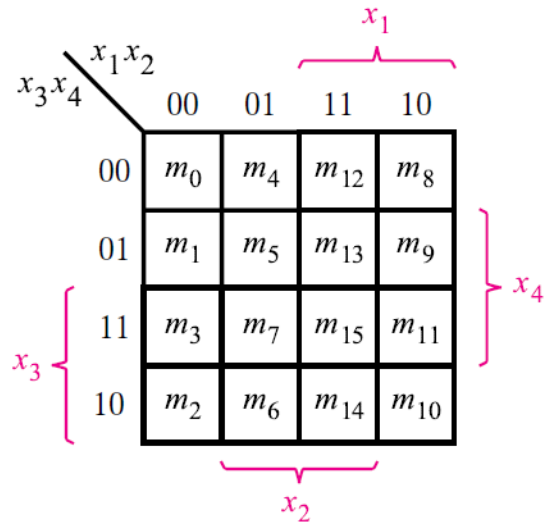
Two examples of it begin filled in and then it's minterms grouped would be as follows:



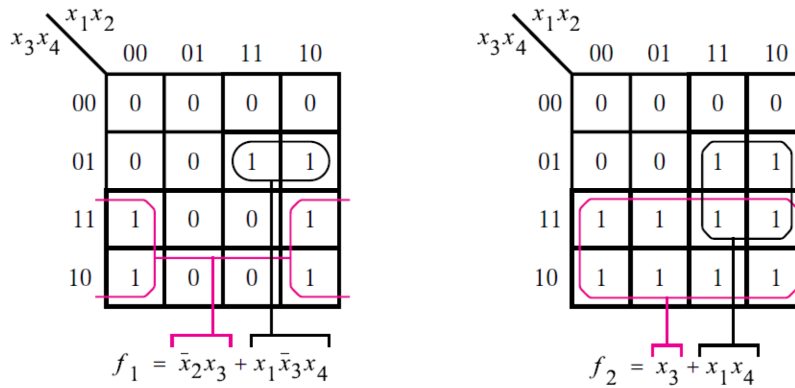


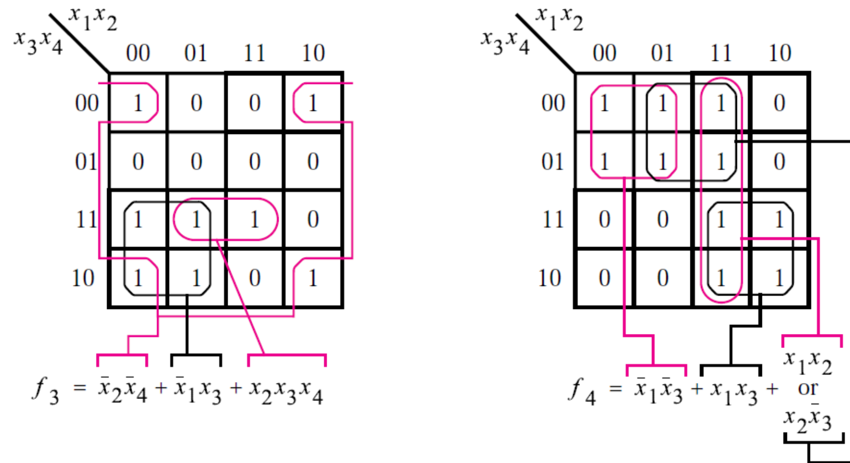
Notice groups are always of size 2^n. Also notice that groups can wrap around the map like in the second example. It is useful to imaging the K-map morphing into two different cylinders by meeting any two opposite edges. *Edges are always connected.*

Four variable K-maps are the biggest we go in this course, the map would look like the following in relation to it's truth table:



A couple examples of some filled in and grouped 4 variable K-maps. Notice that in the last example there is some choice as to how you want to group the terms with the same efficiency. Choice would depend on other factors.





4.2 Strategy for Minimization

The following is a list of terminology:

1. In a product term, each appearance of a variable is called a literal. $x_1\bar{x_2}x_3$ has three literals.

2. A product term that indicated the input valuations for which a given function is equal to 1 is called an *implicant* of the function. Minterms are examples of implicants, but also pairs of minterms are implicants (like groupings in the K-maps).

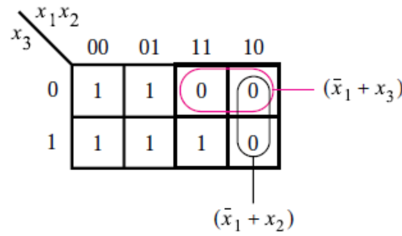
3. Prime implicants are the largest groupings of implicants you can make. They cannoy be combined into another implicant that has fewer terms. It is impossible to delete any literal in a prime implicant and still have a valid implicant.

4. A collection of implicants that account for all valuations for which a function is equal to 1 is called a *cover* for that function. The set of all minterms is a cover, but a number of different covers exist for a given function.

5. The *cost* of a circuit is a relative measure of complexity. Cost can be calculated by adding together all gates, and all the inputs to all those gate. You do **not** include the initial inputs and final outputs to the function as part of the cost.

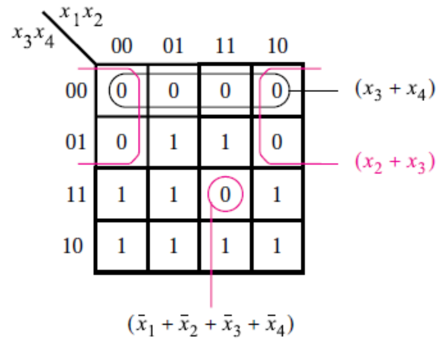
4.3 Minimization of POS Forms

Given some truth table, you can find it's K-map using the same method as before and then group the maxterms instead. The difference comes from the fact that you need to sum up all the constant literals, and then take the product of all those collections of literals. The following example illustrates this process:



$$f = (\bar{x_1} + x_2)(\bar{x_1} + x_3)$$

Or the following 4 variable example:



$$f = (x_2 + x_3)(x_3 + x_4)(\bar{x_1} + \bar{x_2} + \bar{x_3} + \bar{x_4})$$

4.4 Incompletely Specified Functions

Sometimes in a logic function, some valuations can never occur. Say for example you have a two variable function, and the valuation $(x_1, x_2) = (1, 1)$ will never occur for whatever reason. We can then assign $f(1, 1)$ to be whatever we want it to be, as it will never occur. We call this a *don't care condition* or a *don't care* for short. A function which contains *don't cares* is called an incompletely specified function.

In the K-map, we assign that minterm the value of d and while grouping either $1s$ or $0s$, we can consider the d boxes to be whatever we want it to be to achieve the best minimum-cost implementation. For example:

x_3x_4 \ x_1x_2	00	01	11	10
00	0	1	d	0
01	0	1	d	0
11	0	0	d	0
10	1	1	d	1

$x_2\bar{x}_3$

$x_3\bar{x}_4$

Along with the special notation for this kind of function:

$$f = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$$

Where D represents all the terms which have a value of *don't care.*

Chapter 5: Number Representation and Arithmetic Circuits

Binary Numbers

Unsigned Integers

Numbers that are only positive are called *unsigned*, while numbers which can be positive or negative are called *signed.*

The digits of a numbers are written side by side, in increasing order. The $(n+1)$th digit is to the left of the nth digit. For some n digit number D, which has digits $d_0 - d_{n-1}$, it can be written as:

$$D = d_{n-1}d_{n-2}\dots d_1d_0$$

We say that the value of D ($V(D)$) is the quantity obtained by multiplying each digit with the chosen *base* raised to the corresponding power. In base 10 that is:

$$V(D) = d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + \dots + d_1 \times 10^1 + d_0 \times 10^0$$

It is important we start counting at 0 since it allows for the final power of 10 to be equal to 1, this makes us able to count any integer digit we want in any

base. A binary number is *base* 2, and so the base 10 value of a binary number (B) can be written as:

$$V(B) = \sum_{i=0}^{n-1} b_i 2^i$$

When we write a number we assume it to be in base 10, to denote another base, such as base n we denote it:

$$(d_{n-1}d_{n-2}\ldots d_2 d_1 d_0)_n$$

For example:

$$(1101)_2 = (13)_{10}$$

Conversion from binary to decimal is done by definition of *value* above. The opposite is done by dividing by 2, writing down the remainder, and repeating until you are at 1 or 0. You write the number from left to right. The leftmost digit is called the *most significant bit* (MSB) while the rightmost but is the *least significant bit* (LSB)

5.1 Number Representations in Digital Systems

The four number systems we use in this course are *decimal* (Base 10), *binary* (Base 2), *octal* (Base 8), and *Hexadecimal* (Base 16). This is done because decimal is what we are used to, binary is for digital logic, octal is just groupings of three binary bits and hexadecimal is just groupings of four binary bits, as in:

$$\underbrace{1\,0\,1}_{5}\quad \underbrace{0\,1\,1}_{3}\quad \underbrace{0\,1\,0}_{2}\quad \underbrace{1\,1\,1}_{7}$$

$$(101011010111)_2 = (5327)_8$$

$$\underbrace{1\,0\,1\,0}_{\text{A}}\quad \underbrace{1\,1\,1\,1}_{\text{F}}\quad \underbrace{0\,0\,1\,0}_{2}\quad \underbrace{0\,1\,0\,1}_{5}$$

$$(1010111100100101)_2 = (AF25)_{16}$$

5.2 Addition of Unsigned Numbers

The following table represents what we expect the output to be in a 1*bit* adder which ads a single bit of data to another bit of data:

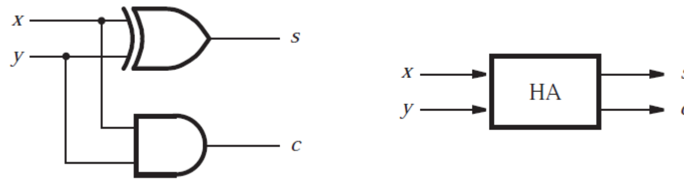
$$\begin{array}{r} x \\ +\,y \\ \hline c\ s \end{array}\qquad \begin{array}{r} 0 \\ +\,0 \\ \hline 0\ 0 \end{array}\quad \begin{array}{r} 0 \\ +\,1 \\ \hline 0\ 1 \end{array}\quad \begin{array}{r} 1 \\ +\,0 \\ \hline 0\ 1 \end{array}\quad \begin{array}{r} 1 \\ +\,1 \\ \hline 1\ 0 \end{array}$$

Carry (c) Sum (s)

The sum is the first digit of the answer, and the carry is the second digit. The second digit occurs when you need your answer to be a 2 digit number. The truth table that would implement this for two inputs x, y and two outputs c, s would be:

x	y	Carry c	Sum s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The minimized form of it's implementation is the following, along with it's abstracted symbol:



This circuit is called the *half-adder*, because it has half the functionality of the *full-adder* which we will discuss next.

We are interested in building a circuit which can add n bit numbers. To do this, you would need to have an adder circuit which can take in two digits of an n but number, and add them along with any *in carry* (c_i) from the previous bit, and then output the singe digit sum, and single digit *carry out* (c_{i+1}) which would go into the adder circuit for the next digit. We call this type of adder a **full adder**.

The following is the truth table for the full adder:

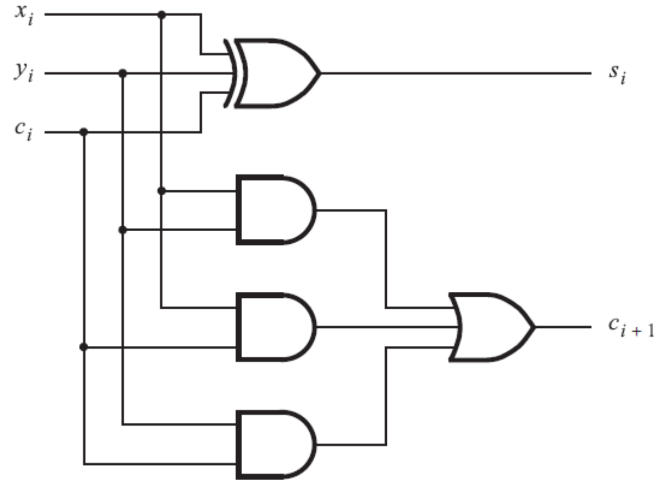
c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Using any method we can derive that the outputs are as follows:

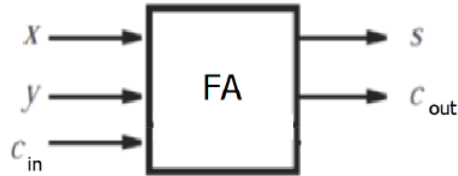
$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

$$s_i = x_i \oplus y_i \oplus c_i$$

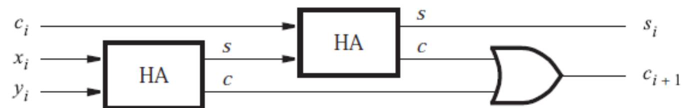
The implemented circuit can be draw like this:



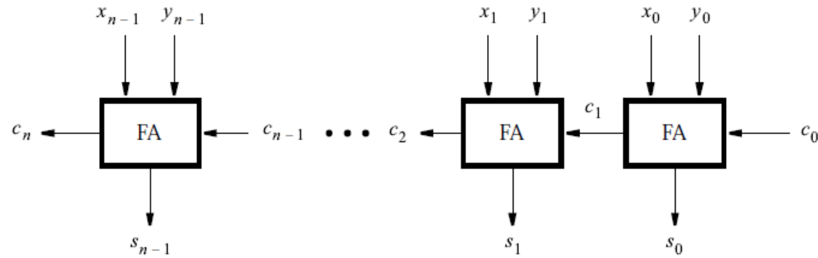
... and the block diagram can be drawn like this:



... additionally you can think of the full adder in terms of half adders like follows:



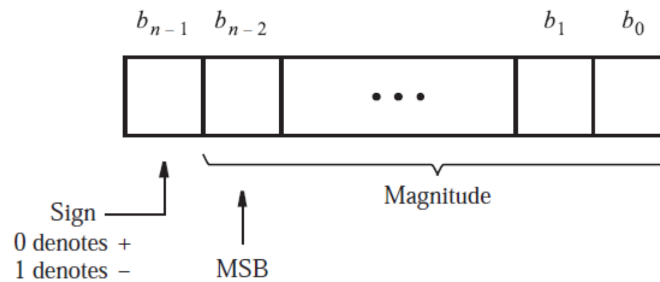
Ripple carry adders are adder circuits which implement the above discussion of linking full adders together, and will form a basis for the *adder-subtracter unit* (ASU).



In the above diagram we are adding two $n - bit$ numbers with the nth digit denoted x_n and y_n. Each full adder adds a single digit, takes in the correct carry and then outputs a new carry out and a sum.

Signed Numbers

In digital logic, the bit that represents the sign is 0 for a positive number and 1 is for a negative number.



Signed numbers can be represented in three ways:

1. **Sign-and-Magnitude** is where the MSB represents the sign of the number, and the remaining bits represent the magnitude.

$$+5 = 0101 \quad -5 = 1101$$

2. **1s Complement** is where to negate a number, you flip all of its bits, once again leaving one bit on the left hand side only for the sign .

$$+5 = 0101 \qquad -5 = 1010$$

3. **2s Complement** is where to negate a number, you start from the LSB, and work your way along until you reach a 1, then **after** that 1 start flipping all the bits. once again leaving one bit on the left for the sign.

$$+2 = 0010 \qquad -2 = 1110$$

   Additionally, you can add 1 to the 1s complement of a number to get the 2s complement.

Table 5.1 Interpretation of four-bit signed integers.

$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	−0	−7	−8
1001	−1	−6	−7
1010	−2	−5	−6
1011	−3	−4	−5
1100	−4	−3	−4
1101	−5	−2	−3
1110	−6	−1	−2
1111	−7	−0	−1

We almost never use sign and magnitude because addition of integers is easy with it (you would need additional circuitry to compare their magnitudes first). We do sometimes use *1s complement* for addition. To do this you:

- Add column by column.
- Carry when needed.
- If the last addition has a carry, add 1 to your sum to get the final answer.

$$\begin{array}{r} (+5) \\ +\ (+2) \\ \hline (+7) \end{array} \qquad \begin{array}{r} 0101 \\ +\ 0010 \\ \hline 0111 \end{array} \qquad\qquad \begin{array}{r} (-5) \\ +\ (+2) \\ \hline (-3) \end{array} \qquad \begin{array}{r} 1010 \\ +\ 0010 \\ \hline 1100 \end{array}$$

$$\begin{array}{r} (+5) \\ +\ (-2) \\ \hline (+3) \end{array} \qquad \begin{array}{r} 0101 \\ +\ 1101 \\ \hline 1\ 0010 \\ \to 1 \\ \hline 0011 \end{array} \qquad\qquad \begin{array}{r} (-5) \\ +\ (-2) \\ \hline (-7) \end{array} \qquad \begin{array}{r} 1010 \\ +\ 1101 \\ \hline 1\ 0111 \\ \to 1 \\ \hline 1000 \end{array}$$

In *2s complement* the method is almost the same except we can completely ignore the final carry if there is one, and the sum will be correct. This is illustrated below:

$$\begin{array}{r} (+5) \\ -\ (+2) \\ \hline (+3) \end{array} \qquad \begin{array}{r} 0101 \\ -\ 0010 \\ \hline \end{array} \Rightarrow \begin{array}{r} 0101 \\ +\ 1110 \\ \hline 10011 \end{array}$$

ignore

$$\begin{array}{r} (-5) \\ -\ (+2) \\ \hline (-7) \end{array} \qquad \begin{array}{r} 1011 \\ -\ 0010 \\ \hline \end{array} \Rightarrow \begin{array}{r} 1011 \\ +\ 1110 \\ \hline 11001 \end{array}$$

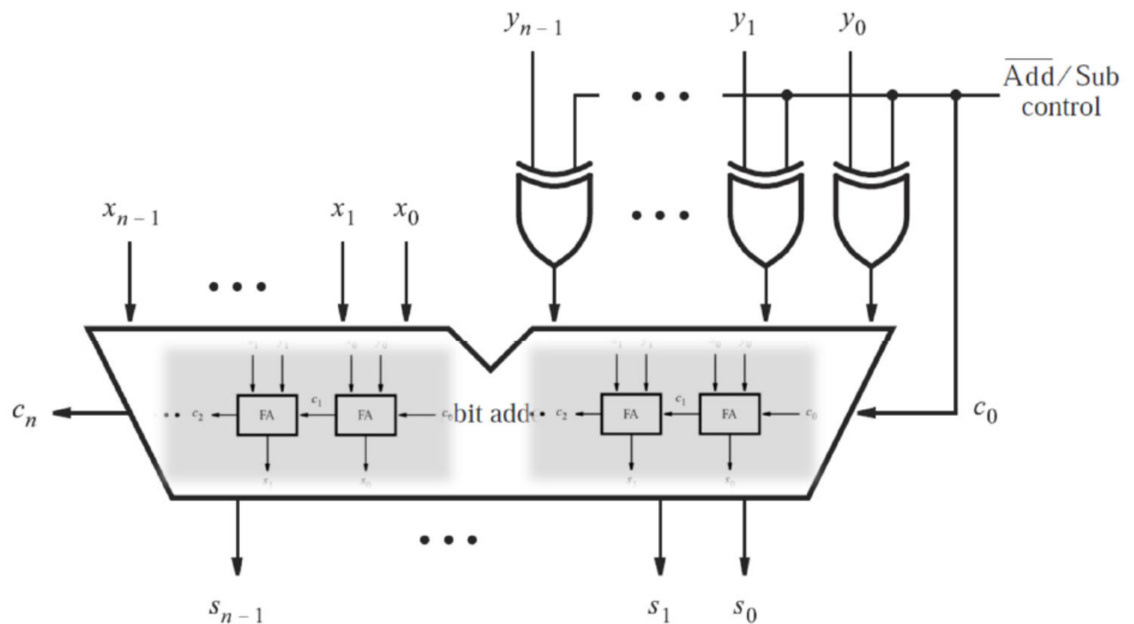
ignore

$$\begin{array}{r} (+5) \\ -\ (-2) \\ \hline (+7) \end{array} \qquad \begin{array}{r} 0101 \\ -\ 1110 \\ \hline \end{array} \Rightarrow \begin{array}{r} 0101 \\ +\ 0010 \\ \hline 0111 \end{array}$$

$$\begin{array}{r} (-5) \\ -\ (-2) \\ \hline (-3) \end{array} \qquad \begin{array}{r} 1011 \\ -\ 1110 \\ \hline \end{array} \Rightarrow \begin{array}{r} 1011 \\ +\ 0010 \\ \hline 1101 \end{array}$$

Adder Subtracter Unit (ASU)

The adder subtracter unit is a circuit block element which can add or subtract two n-bit numbers. It takes in all the digits of both numbers in binary, as well as a carry in, and it outputs the digits of the sum as well as a carry out. The carry in is used to control the operation, if carry in is 1 then 1 is being added to all the inverted inputs of the second number (by the XOR gates) which is exactly how you find the *2s complement* of a number. Then all these inputs are fed into a ripple carry adder, and the summation of the signed numbers is complete.



In use here is the idea what the XOR gate can be configured in a way like a not gate with an enable/disable signal.

Arithmetic Overflow

Sometimes the sum of two n but numbers requires $n+1$ bits to represent. But if your adder is only setup to output n bits, the result will be incorrect. We need to be able to detect this happening so that the circuit can report what's called an *overflow error*.

The signed sum must be in the range of:

$$-(2^{n-1}) \rightarrow 2^{n-1} - 1$$

To detect an overflow error when adding two n-bit numbers you check the last

two carries:

$$\text{If } c_n = c_{n-1} \text{ then there is no overflow.}$$

$$\text{If } c_n \neq c_{n-1} \text{ then there is overflow.}$$

... or in other words:

$$\text{Overflow } = c_n \oplus c_{n-1}$$

BCD Representation

Binary-coded-decimal representation of a number is a way to represent decimal numbers by encoding each digit into it's binary form.

The benefit of this is that you also get the BCD form of the answer, even if in decimal the number has multiple digits. Each set of n bits in the sum of an n-bit BCD adder, corresponds to a decimal digit.

If the sum exceeds 9, then you need to correct the answer by adding 6. For example:

If the addition > 9

	7	0111
	+ 5	+ 0101
	12	1100
correction	+ 6	+ 0110
		1 0010
carry in BCD		0001 0010
		1 2

Parity

The parity bit is sometimes added to the end of string of bits to check for errors in transmission. The parity bit has two types, which change its interpretation. In both cases, the parity bit is calculated by XORing all the data bits (for example in 3 bits of data):

$$p = x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

The receiver would then check to see if the parity is correct by doing:

$$c = p \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

If $c = 0$ then the parity is correct, if $c = 1$, then there is an error. Note that $c = 0$ does not guarantee no error has occurred.

Even-Parity

In even parity, the parity bit is 0 if the number of $1s$ is even. Otherwise the parity bit is 1.

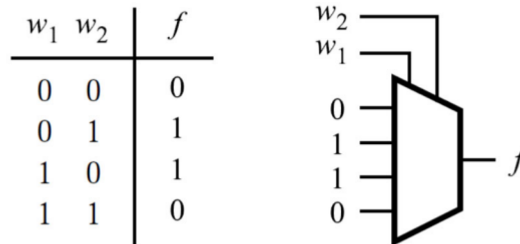
Odd-Parity

In odd parity, the parity bit is 0 if the number of $1s$ is odd. Otherwise the parity bit is 1.

Chapter 6: Combinational-Circuit Building Blocks

6.1: Implementation of Logic Functions using Multiplexers

Using the same logic as LUT, a function can be implemented using multiplexers with input variables being used as the control, and the inputs to the multiplexer is the truth table of the function. For example:



Shannon's Theorem

Any logic function $f(w_1, \ldots, w_n)$ can be written in the form:

$$f(w_1, \ldots, w_n) = \bar{w_p} \cdot f(w_1, \ldots, w_{p-1}, 0, w_{p+1}, \ldots, w_n) + w_p \cdot f(w_1, \ldots, w_{p-1}, 1, w_{p+1}, \ldots, w_n)$$

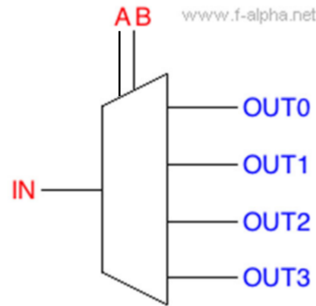
Where p is between 1 and n.

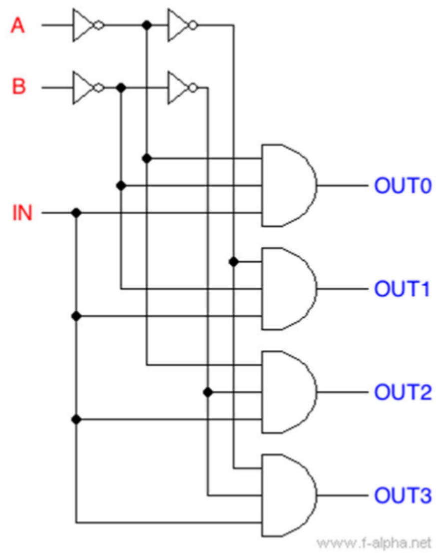
This decomposition can be done using any of the variables, and using different ones will end up with different simplicities. Use the variable which appears in the most terms.

Demultiplexer

A demultiplexer serves the opposite purpose of a multiplexer. Taking one signal and then setting it to go to one of n different outputs based on the control variables.

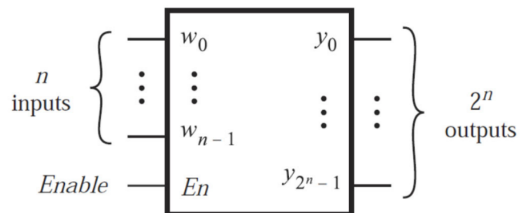
The following is it's symbol, and then implementation using logic gates:





6.2: Decoders

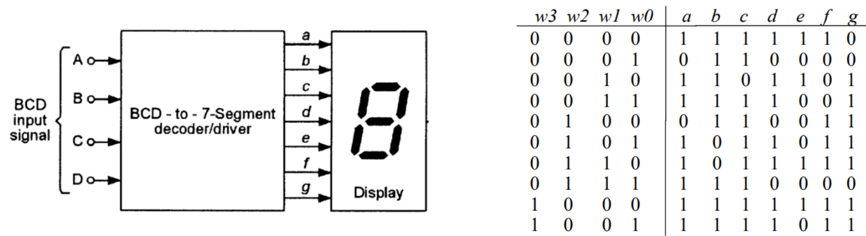
A decoder decodes encoded information. Encoded information is information which was compressed to express the same data in less bits. By this logic a decoder would make the data larger, which is why the decoder has n inputs and 2^n outputs.



Decoders also have an enable signal which controls whether the block should

actually process the data.

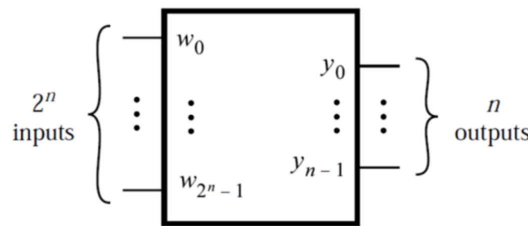
One example of a decoder is a BCD sever segment display decoder which takes in a 4 bit number and outputs 7 signals which each correspond to a led in the display:



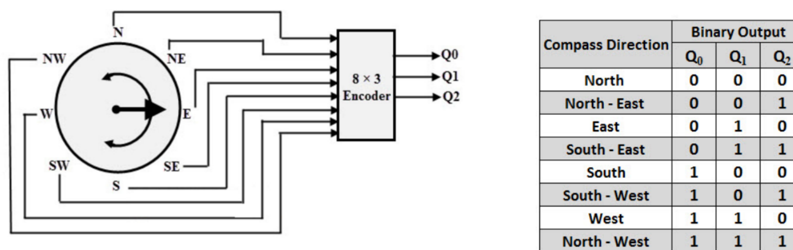
w3	w2	w1	w0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

6.3 Encoder

An encoder encodes information to be decoded by a decoder. This means the information is being compressed into a simpler number.



One particular example of an encoder is the direction a compass is facing converted into a 3-bit binary signal:

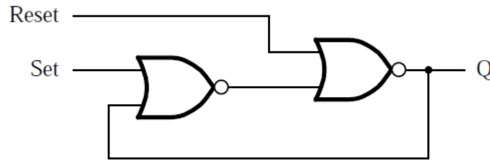


Compass Direction	Binary Output		
	Q_0	Q_1	Q_2
North	0	0	0
North - East	0	0	1
East	0	1	0
South - East	0	1	1
South	1	0	0
South - West	1	0	1
West	1	1	0
North - West	1	1	1

Chapter 7: Flip-Flops, Registers, Counters, and a Simple Processor

So far we have considered combinational circuits whose outputs depend on the input variables. We now consider *sequential* circuits whose outputs depend on

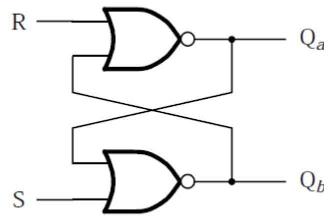
the input variables as well as the previous state of the circuit. Take the following circuit for example:



Here, if $RESET = 0$ and $SET = 1$ then the line segment between the NOR gates will be 0, and so $Q = 1$. If I then turn off SET, the output will still be 1 since the line segment between the NOR gates is still 0. The circuit remembers the effect of the SET signal even after it has turned off. Then, if $RESET = 1$, the Q value will reset to 0. This is an example of a basic latch. Typically however, the latch is rearranged so that the line segment between the NOR gates gets its own name Q_b, while our main output is Q_a, note that $Q_a = \bar{Q_b}$.

7.1 Basic (SR) Latch

The basic set-reset latch:



The following is the truth table for the basic latch:

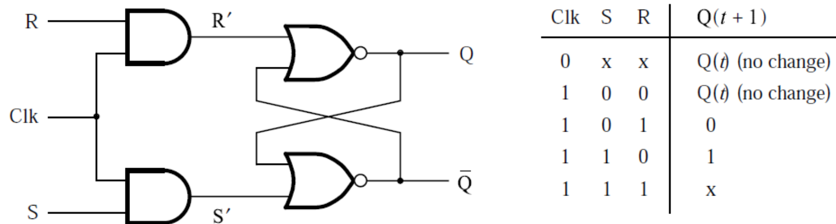
S	R	Q_a	Q_b	
0	0	0/1	1/0	(no change)
0	1	0	1	
1	0	1	0	
1	1	0	0	

The set signal changes Q_a to 1 when it is on, and the Reset signal changes Q_a to 0 when it is off. Note nothing happens when either turn off, that is the memory of the circuit.

7.2 Gated SR Latch

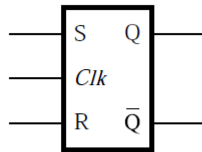
The gated SR latch provides us with a way to enable or disable the latch. This allows us to know when the latch is being updated by the use of the *Clock* or

Clk signal. This latch also has a set and Reset which function the same way as the basic latch. The following is it's implementation and truth table:



Clk	S	R	Q(*t* + 1)
0	x	x	Q(*t*) (no change)
1	0	0	Q(*t*) (no change)
1	0	1	0
1	1	0	1
1	1	1	x

... and the following is it's block diagram:

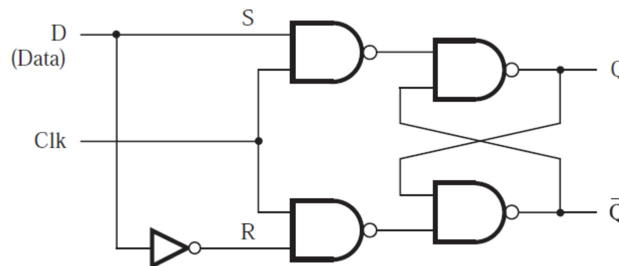


If the clock signal is 1, then the circuit will respond to changes in S and R, otherwise it will not. It is called clock to reflect the fact that in many circuits, latches are enabled and disabled in a complex arrangement of timings. Circuits which use a control signal are called gated latches.

7.3 Gated D Latch

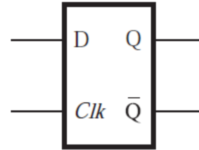
A gated D latch is a useful way to store bits. This means the circuit remembers the signal coming in for an indefinite amount of time. Think of the outputs of an ASU, which need to be remembered regardless of if they are a 1 or a 0. This is why we need a *Data* latch or a D-latch.

The following is its implementation:



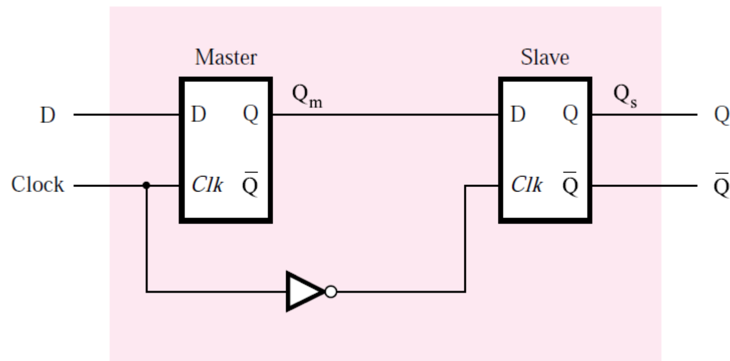
... and this is it's truth table, and graphical symbol:

Clk	D	Q(t + 1)
0	x	Q(t)
1	0	0
1	1	1



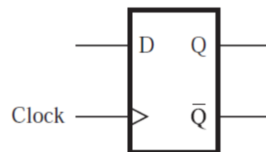
7.4 Master-Slave and Edge-Triggered D Flip-Flops

If you align two gated D latches, in the following circuit:



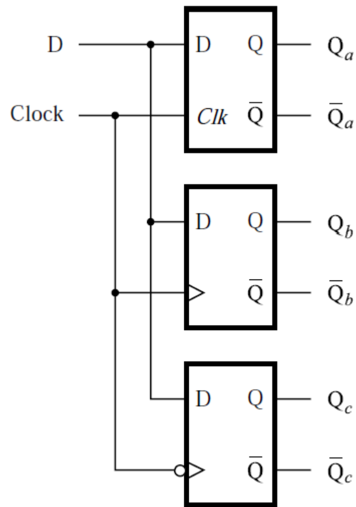
You can see that if $D = 1$, and $Clk = 1$, then $Q_m = D$. If we then lower Clk to 0, Q_m will no longer follow D, but Q_s will follow D, changing our output only once. We say that these kinds of circuits are *edge-triggered* since the circuit only updates when the clock either goes from 1 to 0 (negative edge triggered), or from 0 to 1 (positive edge triggered).

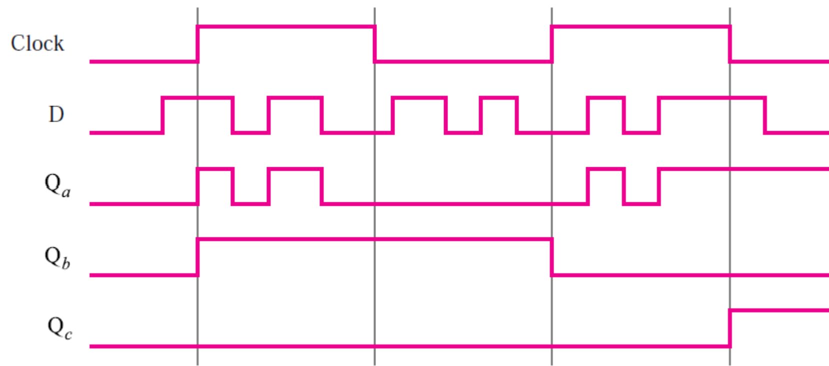
We call edge-triggered memory elements: *flip-flops*, a positive edge triggered D flip-flop is denoted:



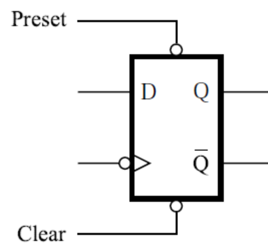
When we clock the circuit (meaning provide an edge to the clock), then the circuit updates and Q follows D. At any other moment the circuit is unresponsive to changes in D.

The following is three of the possibilities of D flip-flip/latches, and their timing diagrams. The latch is *positive-level activated*:



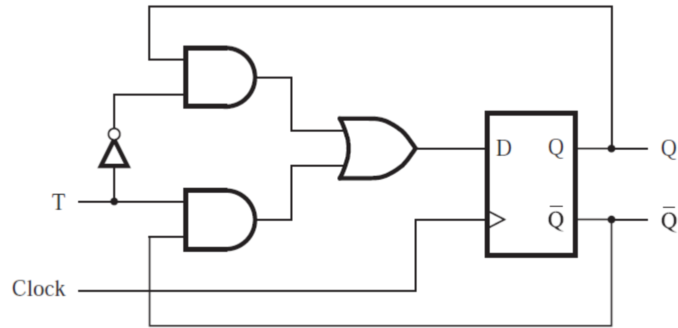


We can advance the *D* flip-flop somewhat by adding ina *Preset* and *Clear* signal which are both low active (to save resources). If *Preset* = 0 then the initial output of the circuit = 1. If *Reset* = 0 then the initial output of the circuit = 0. Typically *Preset* and *Clear* are asynchronous to the clock. The following is it's circuit diagram:



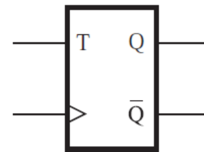
7.5 T Flip-Flop

Toggle flip-flops (or T flip-flops) are similar to D flip-flops, except they use an input T to either invert or not invert the output, whatever the initial output was. The following is it's implementation:



... and the following is it's truth table and circuit symbol:

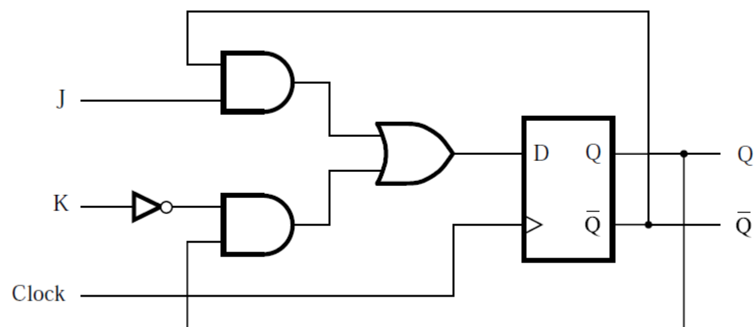
T	$Q(t+1)$
0	$Q(t)$
1	$\overline{Q}(t)$



Note that it is positive edge-triggered.

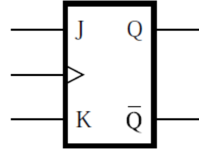
7.6 JK Flip-Flop

JK flip-flops are versatile circuit elements, which can act as either an SR-latch or a T-flip flop in once circuit. It's implementation is below:



... along with its truth table and circuit symbol:

J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\bar{Q}(t)$



Note that $Q(t+1)$ denotes the state of the circuit after the next clock (edge).

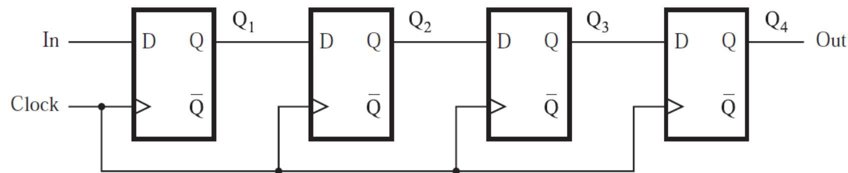
When only one is active at a time, J acts as a *Set*, and K acts as a *Reset*. When both are active or inactive at the same time, the circuit acts like a T flip-flop.

7.8 Registers

A flip flop stores one bit over a clock cycle. If n flip-flops are lined up to store n bits of data, we say they form a *register*.

7.8.1 Shift Registers

A shift register allows us to shift the bits of a n bit number over by 1 every clock cycle. The following is it's diagram:



As you can see, whenever the circuit is clocked, it happens simultaneously, and so $Q_n \to Q_{n-1}$. You can feed in a number by changing In every clock, and take the outputs of the register at Q_n. This allows us to store a n bit number in this register with n flip-flops. The following is a sample sequence:

	In	Q_1	Q_2	Q_3	Q_4 = Out
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

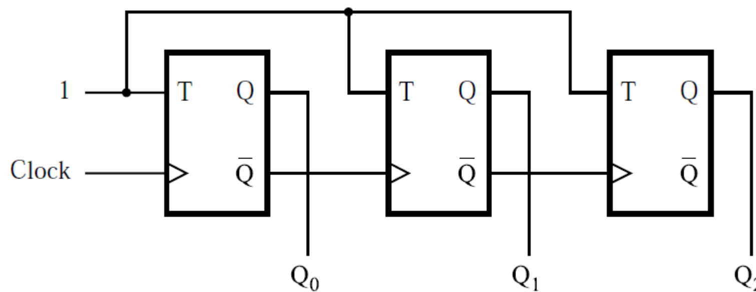
7.9 & 7.11 Counters

A counter circuit increments or decrements a stored value by 1 at a time. This is very useful in a lot of circuits and so we want to be able to make it using simpler circuits than what we had in *Chapter 5*.

For now we just learn about these different types of counters, we learn how to actually design them from scratch in *Chapter 8*.

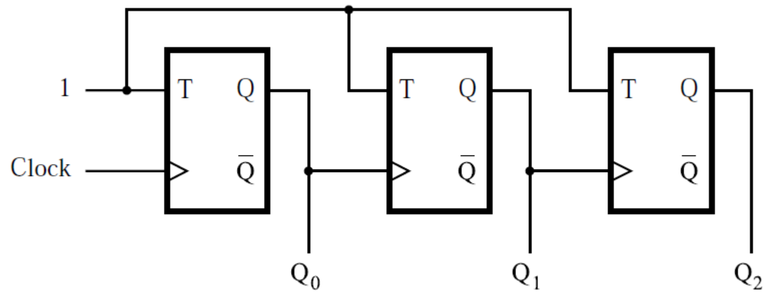
7.9.1 Asynchronous Counters

Asynchronous counters have flip-flops which are not all connected to the same clock signal. The following is a *modulo-8 up-counter* meaning it cycles between 0 and 7 counting up one at a time. You can read the output as $Q_2Q_1Q_0$:



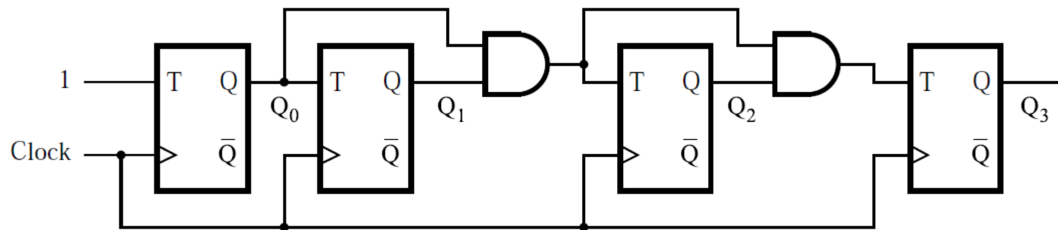
Note that the input to each T flip-flop is always 1 which means the output will always toggle when clocked. Also remember that these are positive edge triggered flip-flops.

After some slight modification we have a *modulo-8 down-counter*:



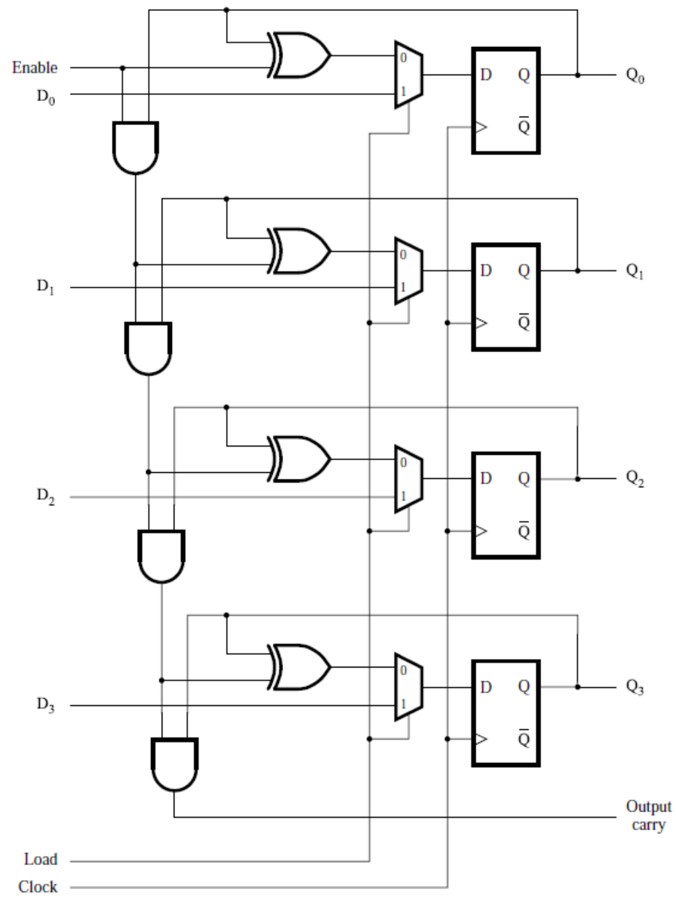
7.9.2 Synchronous Counters

A synchronous counter has all the flip-flops connected to the same clock, this is more efficient. The following is a *four-bit synchronous up-counter*:



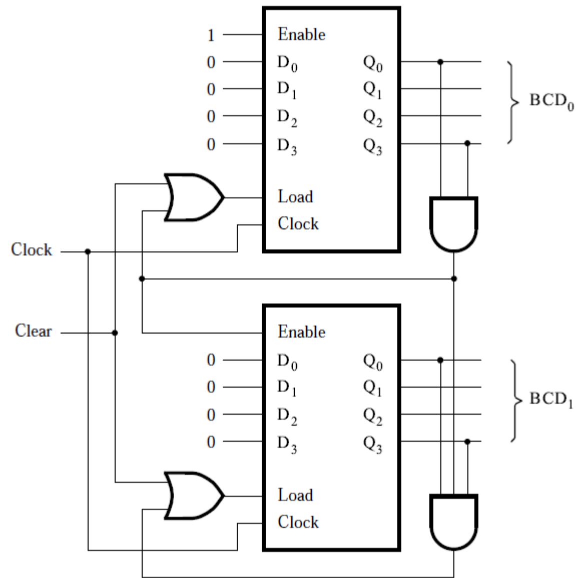
7.9.3 Counter with Parallel Load

If you want a counter to start at a value that isn't 0, then you need to load in data. Say you want to load in a 4 bit number: $D_3D_2D_1D_0$ only when another signal *Load* is active. Then you could use the following circuit and take the outputs at $Q_3Q_2Q_1Q_0$:



7.11.1 BCD Counter

The following circuit is used to count up using two BCD outputs which could be run into a seven segment display. Note that if the output is a 9, then $Load = 1$ which means the counter will be reset.



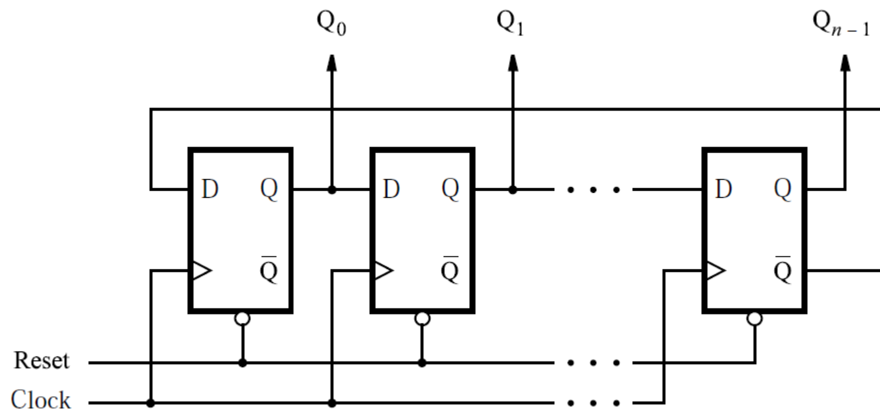
This circuit uses two modulo-10 counters.

7.11.3 Johnson Counter

The Johnson counter outputs a string of outputs which vary only by 1. For example the 4 bit Johnson counter outputs the following in a sequence of 8 clocks if initialized to 0:

$$0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000$$

... and so on. The following is the circuit:



Chapter 8: Synchronous Sequential Circuits

In this chapter we study the design on circuit using circuit elements from chapter 7. These circuits are called *synchronous* because all elements rely on the same clock signal.

Before designing circuits with these elements we must understand some basic design steps.

8.1 Basic Design Steps

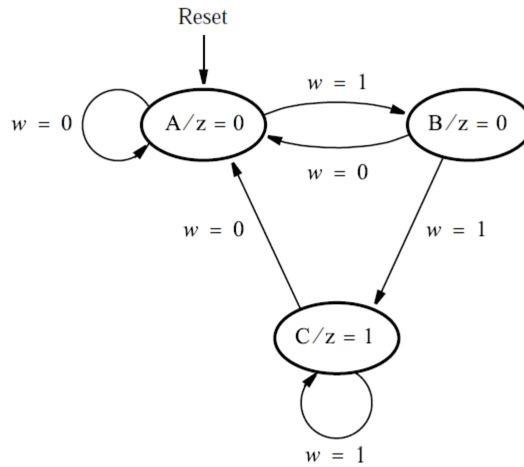
The example used in this text is a circuit with the following criteria:

1. It has one input w, and one output z.
2. All changes in the circuit happen at the positive edge of the clock.
3. The output is equal to 1 if for two clocks in a row the input is equal to 1.

We can think of the circuit as a logical tool called a *Finite State Machine* which is a set states, and instructions of how to get from state to state. In this circuit:

- **State A** represents the starting place. The output of the circuit is 0 since the input has not been 1 for two clocks yet. From here we could either move to state B if the input becomes 1, or we could stay in state A if the input stays 0.
- **State B** is where the circuit moves to if the input has been 1 for 1 clock cycle. The output is still 0 since the criteria for the output being 1 was that the input should be 1 for *two* clock cycles. But the circuit remembers that the input has been 1 for one clock by being in this state. From here we could either move to state C if the input stays 1, or we could go back to state A if the input goes back to 0.
- **State C** is where our output is 1! This only happens when the circuit is in state B and the input is 1 for another clock. This means that the input has been 1 for a total of two clock ($A \to B \to C$) and so we output 1 as per the specifications. From here we could either stay in C if the input stays 1 or we could go back to A if the output becomes 0.

Now that's a **lot** of words which could have been easily represented by a *state diagram.*



In this diagram, the big bubbles represent *states* the circuit can be in. Within the state you can see the output of the circuit. Between states there are arrows which show what state the circuit will move to if the input is what is specified. Reset simply refers to the initial state of the circuit.

To begin the design of this circuit, we could translate this state diagram into a **state table**, as follows:

Present state	Next state $w = 0$	Next state $w = 1$	Output z
A	A	B	0
B	A	C	0
C	A	C	1

Which is typically just an intermediate step before the **state-assign table**:

	Present state $y_2 y_1$	Next state $w = 0$ $Y_2 Y_1$	Next state $w = 1$ $Y_2 Y_1$	Output z
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	*dd*	*dd*	*d*

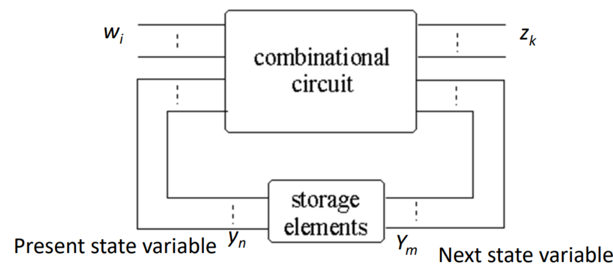
This table will be a key piece of information used in creating the circuit. Just like truth tables were key in sequential circuits.

As you can see here:

- We have assigned a 2-bit binary number to each state, and we called the 11 state nothing. This is because the circuit will never be in this state, and so all its values can be don't cares. The variables y_1 and y_2 are typically used for present state. In general if you have n present states, then you will need the lowest power of 2 which is still greater than n.
- Focusing on the first row, if we are in state A, and the input $w = 0$, then we want to stay in the first row. This is why under next state $w = 0$, we have 00.
- The output of the circuit depends entirely on the present state. So there is just the output column which represents that data. This is called a *Moore Model* of a finite state machine (FSM). More on that later.

The General Model of a Sequential Circuit

The following is the general schematic for every sequential circuit:



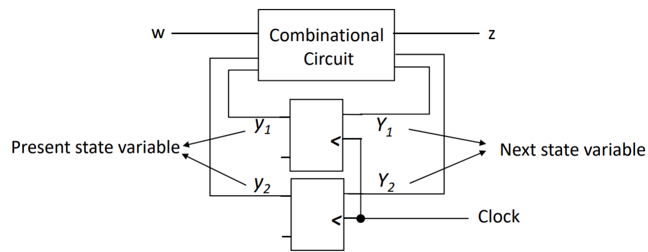
Every sequential circuit has primary inputs w, and primary outputs z. The primary outputs are calculated as logic functions based on:

- Both the primary inputs and the current state (*Mealy State Model*).
- Only the current state (*Moore State Model*).

The combinational circuit will output the next state and the circuit's output. The next state variables go into a *memory block* which is typically constructed using flip-flops. Those flips flops then get clocked and the next state variables become the present state variables, and new outputs/next states are calculated in the combinational part. The directions of the arrows in the diagram above are meaningful.

Designing the Circuit

To continue designing the circuit we had earlier, we note that the circuit will be in the following general form:



We now use K-maps to derive the combinational part of the combinational circuit:

$w \backslash y_2y_1$	00	01	11	10
0	0	0	d	0
1	1	0	d	0

$w \backslash y_2y_1$	00	01	11	10
0	0	0	d	0
1	0	1	d	1

$y_2 \backslash y_1$	0	1
0	0	0
1	1	d

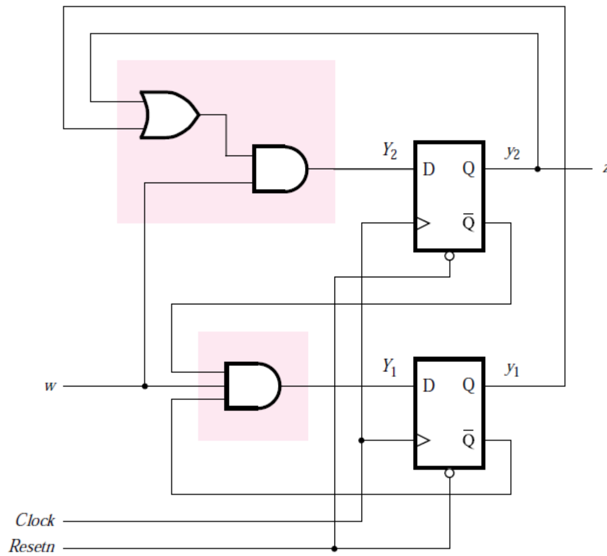
From here you can see:

$$Y_1 = w\bar{y_1}\bar{y_2}$$

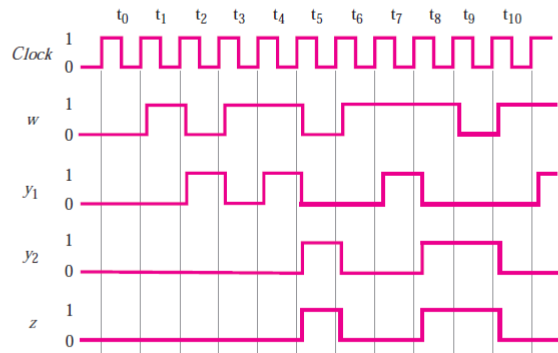
$$Y_2 = w(y_1 + y_2)$$

$$z = y_2$$

We can then implement this circuit using $D-$flip-flops as our storage element in the following way:



... and get it's timing diagram:



This concludes the example which we began this section with.

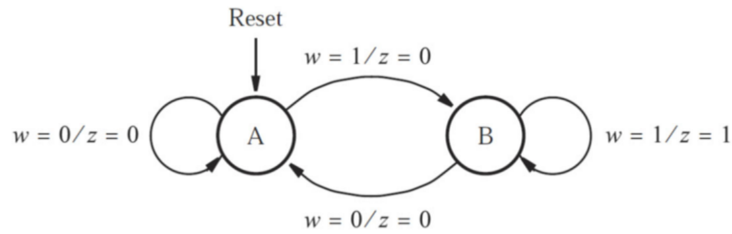
In general the following is true to determine the number of possible states given some number of flip flops:

$$\#States = 2^{\#FF}$$

8.3 The Mealy State Model

The mealy state model is a model of a FSM where the primary output depends on the current state **and** the primary inputs. The following is an example of a:

- **State Diagram**:



- **State Table**:

Present state	Next state		Output z	
	$w=0$	$w=1$	$w=0$	$w=1$
A	A	B	0	0
B	A	B	0	1

- **State-Assign Table**:

	Present state	Next state		Output	
		$w=0$	$w=1$	$w=0$	$w=1$
	y	Y	Y	z	z
A	0	0	1	0	0
B	1	0	1	0	1

- **K-Map Process**:

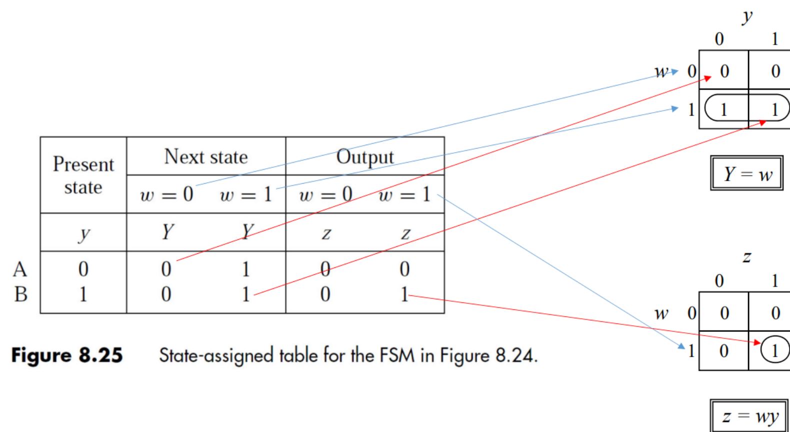
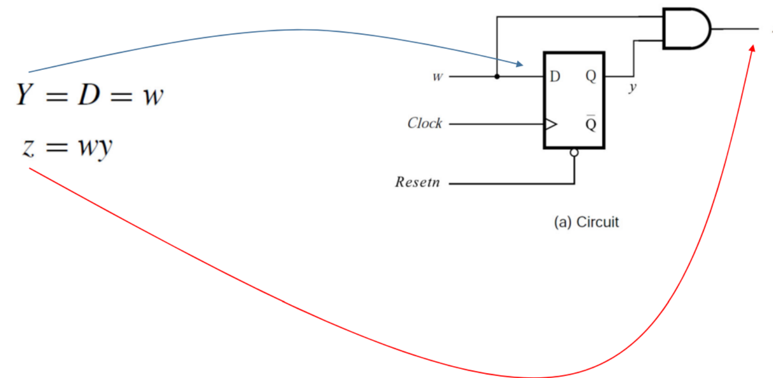


Figure 8.25 State-assigned table for the FSM in Figure 8.24.

- **Final Circuit**:



(a) Circuit

Notice how the output depends on the primary input as well as the current state in all the diagrams, including the final circuit.

General Design Procedure

The following is a general list of steps to be used to derive a circuit given some specification:

- Obtain the specification of the desired circuit.
- Select a starting state and then derive the other states of the circuit, and the conditions needed to move between states.
- Make a state diagram.
- Create a state table.
- Decide on the number of state variables needed to represent all states, and assign each state a binary number.
- Choose the type of flip-flop in the circuit. Derive the next-state logic expressions and the output logic expressions.
- Implement the circuit as indicated by the logic expressions and choice of flip-flops.

Implementation using JK-Flip-Flops and T-Flip-Flops

Initial implementation is always done in this course using $D-$flip-flops. From there you can convert the design to be implemented using JK-flip-flops as well as T-flip-flops. The following table is all you need to do this conversion:

Present State	Next State D-FF	J K-FF	T-FF
0	0	0 d	0
0	1	1 d	1
1	1	d 0	0
1	0	d 1	1

Once you have created the state-assign table for the D-flip-flops, you can convert it to an *excitation table* using the table above. Here is an example of an standard state-assign table:

	Present state	Next state		Count
		$w = 0$	$w = 1$	
	$y_2 y_1 y_0$	$Y_2 Y_1 Y_0$	$Y_2 Y_1 Y_0$	$z_2 z_1 z_0$
A	000	000	001	000
B	001	001	010	001
C	010	010	011	010
D	011	011	100	011
E	100	100	101	100
F	101	101	110	101
G	110	110	111	110
H	111	111	000	111

Below is an example of it begin done for a JK flip-flop:

Once the table in Figure 8.65 has been derived, it provides a truth table with inputs y_2, y_1, y_0, and w, and outputs J_2, K_2, J_1, K_1, J_0, and K_0. We can then derive expressions for

Present State	Next State D-FF	J K-FF	T-FF
0	0	0 d	0
0	1	1 d	1
1	1	d 0	0
1	0	d 1	1

	Present state	Flip-flop inputs **(Next State)**								Count
		$w = 0$				$w = 1$				
	$y_2 y_1 y_0$	$Y_2 Y_1 Y_0$	$J_2 K_2$	$J_1 K_1$	$J_0 K_0$	$Y_2 Y_1 Y_0$	$J_2 K_2$	$J_1 K_1$	$J_0 K_0$	$z_2 z_1 z_0$
A	000	000	$0d$	$0d$	$0d$	001	$0d$	$0d$	$1d$	000
B	001	001	$0d$	$0d$	$d0$	010	$0d$	$1d$	$d1$	001
C	010	010	$0d$	$d0$	$0d$	011	$0d$	$d0$	$1d$	010
D	011	011	$0d$	$d0$	$d0$	100	$1d$	$d1$	$d1$	011
E	100	100	$d0$	$0d$	$0d$	101	$d0$	$0d$	$1d$	100
F	101	101	$d0$	$0d$	$d0$	110	$d0$	$1d$	$d1$	101
G	110	110	$d0$	$d0$	$0d$	111	$d0$	$d0$	$1d$	110
H	111	111	$d0$	$d0$	$d0$	000	$d1$	$d1$	$d1$	111

Conclusion

This concludes the content in this course. I hope these notes were helpful! Good luck in the exam!

- Adam Szava