Notes for ECE 29595PD - Principles of Digital System Design

Shubham Saluja Kumar Agarwal

March 20, 2024

These are lecture notes for spring 2024 ECE 29595PD at Purdue as taught by Professor Irith Pomeranz. Modify, use, and distribute as you please.

Contents

Course Introduction

This course serves as an introduction to digital system design, with an emphasis on principles of digital hardware and embedded system design. It is an alternate class to ECE 27000.
Learning Outcomes:

1. Ability to analyze and design combinational logic circuits.

2. Ability to analyze and design sequential logic circuits.

3. Ability to analyze and design computer logic circuits.

4. Ability to realize, test, and debug practical digital circuits.

Introduction

Digital design entails creating hardware that can conduct an operation or set of operations within a computer system. For example, adding two numbers.

```
┌─────────┐   ┌─────────┐
│    a    │   │    b    │
└─────────┘   └─────────┘
┌───────────────────────┐
│         adder         │
└───────────────────────┘
        ┌─────────┐
        │    c    │
        └─────────┘
```
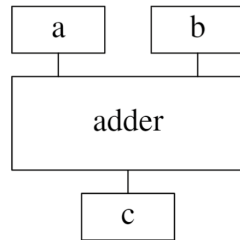
This hardware can add two numbers, that is, conduct the operation $c = a + b$. It could also perform $f = d + e$ or $i = g - h = g + (-h)$ as it is not restricted to the sole values of a and b as inputs. This process fits into the logic design and switching algebra portions of chip manufacturing.

The creation of systems like these is based on the fact that voltage and current are time-varying and can assume any value in a continuous range of real numbers, but are mapped to only two values.

Digital Logic Signals

A digital signal is modeled assuming that at anytime, it can have only one of two discrete values, which represent:

0	1
LOW	HIGH
FALSE	TRUE

This is called positive logic. It maps the infinite values of voltage and current to these two values. An example of this is CMOS 2-Volt logic:

0	1
0-0.5V	1.5-2.0V

These completely separated ranges of values allow for 0 and 1 to be completely separate, with noise and other possible errors being ignored. In this way, all physical values can be partitioned into the two values, though, for intents and purposes of digital system design, voltage is the most relevant.

Additionally, circuits known as buffer circuits can be used to restore logic values. That is, if a voltage is not sufficiently close to the values

of 0 or 2 (in the case of the CMOS), they can push these values far closer to the desired values, reducing the possibility of error.

Digital circuits have replaced analog circuits becuse they are far easier to design. They can be made using Hardware Description Languages (HDLs), softwares that are similar to programming languages. A logic value is called a binary digit, or bit. A set of n bits, represent 2^n values. For example:

b_0
0
1

b_0	b_1
0	0
1	0
0	1
1	1

Logic Circuits

At the highest level, a logic circuit is a *black box* with n inputs and m outputs. Only zeros and ones are required to represent inputs and outputs.
Combinational circuits are circuits that have outputs that solely depend on the inputs. It can be represented by a truth table.
An adder can be represented by a truth table in this manner.

x_1	x_2	x_3	z_2	z_1
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

On the other hand, a sequential circuit has memory. That is, the output is dependent on both the inputs and the current state of the circuit itself. This is representable through a state table.

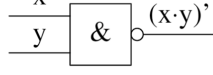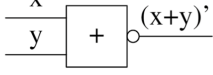| input	present state	next state	output
0	00	00	0
1	00	01	0
0	01	01	0
1	01	10	0
0	10	10	0
1	10	00	1

Gates

Basic digital devices are called gates. These implement basic logic functions. These are AND, OR, NOT. However, AND, NOT and OR, NOT are sufficient to make any combination, as the third gate can be formed as a combination of the other two.

Each of the gates is represented symbolically like the following, and have their truth tables shown below:

AND		OR		NOT	

x	y	$x \wedge y$	x	y	$x \vee y$	x	$\neg x$
0	0	0	0	0	0		
0	1	0	0	1	1	0	1
1	0	0	1	0	1	1	0
1	1	1	1	1	1		

This can be done for more inputs, by combining multiple gates. There are two more often used gates that can be created through a combination of these two.

NAND		NOR	

x	y	$\neg(x \wedge y)$	x	y	$\neg(x \vee y)$
0	0	1	0	0	1
0	1	1	0	1	0
1	0	1	1	0	0
1	1	0	1	1	0

It is a good exercise to see how two of the basic gates (AND, NOT or OR, NOT) can be used to create the rest, as it is possible to connect gates to form complex circuits and thus, complex functions.

Note: By convention, signal flows from left to right. Thus, output comes out the right and input goes in at left.

Timing

When the values of an input changes, it takes time for the outputs to change as well, which is known as *propagation delay*, which can be represented in a timing diagram. However, for most cases, it is possible ot ignore these, as the behavior is well defined regardless of delay.

The following is a timing diagram:

Propagation delay, represented by pd in the timing diagram, is irrelevant if there are no changes, regardless of the fact that there is a new set of inputs. The values at the bottom of the diagram are the values that each of the elements hold after all the necessary transitions have happened for that set of inputs.

Transition time is the time it takes for a signal to change from 0 to 1, and is represented by "rt".

Software Aspects of Digital Design

Software is not technically essential to digital system design. People used to draw symbold by hand, and could make technically equivalent systems. However, the availability, utility, and ease of use of HDLs has made the use of software prevalent in the current technological landscape.

Electronic Design Automation tools are useful in improving designer productivity. The following are examples of these:

- Schematic entry: Allows for fully detailed digital diagrams to be created digitally.

- HDLs: Can be used to design anythong from individual function modules to multichip digital systems. This course will involve extensive use of VHDL.

- HDL text editors, compilers, synthesizers: text editor to define, compiler to debug syntax, synthesizer to create corresponding circuit or chip.

- Simulator: It is virtually impossible to debug a synthesized chip, so simulators are used before synthesis. They allow for verification of functionality prior to the actual tedious process of synthesis. Among simulators, there are also PCB simulators, and FPGAs, which are like programmable chips. They are very important.

- Test benches: Designs are simulated and tested using these. They run a series of checks to ensure that nothing stopped working due to changes.

- Timing analyzers and verifiers: Correct timing of inputs and reactiosn are paramount in digital systems. This facet of simulation allows for the automation of timing functionality verification.

- Word Processors: Allow for pretty documentation to be created.

Software is important, but the understanding of what it actually does is even more so.

Integrated Circuits

A collection of gates on a single silicon chip is called an integrated circuit or IC.
An IC originates from a wafer, which is segmented into many identical ICs. The wafer is divided into rectangles, called *dies* which in turn have *pads* which allow for wire connections. Microscopic probes are used to debug the wafers, and defects are discarded, and the succeses, cut out.
For this class, an IC, is a packaged die.
ICs were divided by size, small SSIs with 1 to 20 gates, medium MSIs with 20 to 200, large LSIs with upto 1000, and vely large VLSIs fro everything above that. The largest VLSIs now have over 10 billion transistors.
An IC process is the steps taken to create an IC, and are categorized by the transistor density within the chip.

Logic Families

There are many different ways of implementing logic circuits. Connectable logic circuits are of the same logic family. So, the technology controlling them is different. Chips need to be from the same family to be connected to each other. If they are from different families, they cannot be connected to each other.
For example TTL (Transistor-Transistor Logic) is a logic family based on bipolar junction transistors.
CMOS (Complementary Metal-Oxide Semiconductor) is based on MOSFETs (MOS Field-Effect Transistors). These are more commonly used, and the ones we will be analyzing in this class.

CMOS Logic Circuits

A MOS transistor is modellable as a voltage controlled resistor. This means that the input voltage controls the resistance of the MOSFET,

allowing it to act as a kind of switch. This is because it has only two effective states: very high resistance and very low resistance.

This is an n-channel (NMOS) transistor:
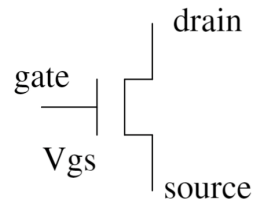
When $V_{gs} = 0$, the resistance is very high.
When V_{gs} is increased, the resistance is very low.
On the other hand, this is the PMOS transistor:

When $V_{sg} = 0$, the resistance is very high.
When V_{sg} is increased, the resistance is very low.
NMOS and PMOS together allow us to form CMOS logic. For example, this is a CMOS inverter, in which the output will be the opposite of the input.
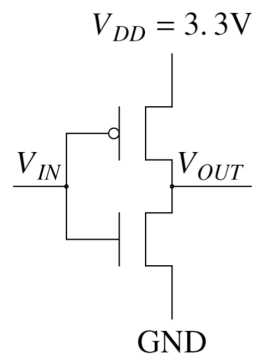
If $V_{in} = 3.3V$, or high, the PMOS transistor be off, and the NMOS would be on. This would mean that the output and ground are shorted, causing the output to be low.
If $V_{in} = 0V$, or low, the PMOS transistor be on, and the NMOS would be off. This would mean that the output and V_{DD} are shorted, causing the output to be high.

This allows this to be act as an inverter.

Now we will observe the CMOS arrangement of a NAND gate.

If A and B are both HIGH, both PMOS are HIGH, and thus V_{DD} is shorted to Z.

If they are both LOW, the exact opposite happens, with Z shorting to ground.

If A or B are HIGH, than one of the PMOS will be high, and one of the NMOS will be LOW, causing the output to be HIGH.

The NOR gate can be constructed easily from the same thought process, but NMOS in parallel, and PMOS in series.

These can be expanded to more inputs by adding more transistors.

Preview of future topics

The following is a multiplexer:
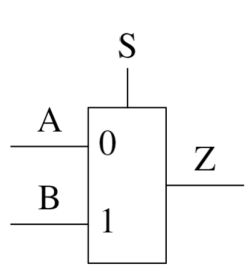
It outputs A when S is 0, and B when S is 1. It allows us to select between different functions, such as adding and subtracting.

Truth tables are another form of representation of a digital system. We will later learn how to analyze and derive equations from truth tables, as well as the gate implementation.

Note: CMOS logic doesn't have AND, OR gates, so the actual implementation of these two is NAND, NOR gates with an inverter.

Gates require 4 transistors, and inverters need two transistors. Verilog
Structural model:

```
module Ch1mux_s(A,B,S,Z);
    input A,B,S;
    output Z;
    wire SN,ASN,SB;
    not U1(SN,S);
    and U2(ASN,A,SN);
    and U3(SB,B,S);
    or U4(Z,ASN,SB);
endmodule
```

Which is equivalent to:

*Note: the order of these statements is irrelevant, they will result in the same
digital system.* Verilog Behavioral Model:

```
module Ch1mux_b(A,B,S,Z);
    input A,B,S; \\input declaration
    output reg Z; \\output declaration
    always @(A,B,S) \\if input changes, recompute
    if (S==1) Z=B;
    else Z=A; \\this is saying what the multiplexer above did
endmodule
```

Number Systems and Codes

A digital circuit processes binary digits in the form of bits.
It is impoortant to see how these relate to real life values.
Numbers in the decimal system:

$$abcd = 1000a + 100b + 10c + d = 10^3a + 10^2b + 10^1c + 10^0d$$

The base, or radix is 10. For a general radix r, it would be:

$$abcd = r^3a + r^2b + r^1c + r^0d$$

For this course, we care about binary, or radix 2.

base 2	base 8	base 16
0	0	0
1	1	1
10	2	2
11	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	10	8
1001	11	9
1010	12	A
1011	13	B
1100	14	C
1101	15	D
1110	16	E
1111	17	F

Occasionally, base 8 (octal) and base 16 (hexadecimal) are also relevant. To convert from octal to binary, replace every octal with three bits starting from the least significant bit. For hexadecimal, replace each digit with 4 bits.

The least significant bit(LSB) is the last bit, and the first is the most significant bit (MSB).

To convert from decimal to binary by dividing the number by 2, and assigning the remainder to the least siginificant unassigned bit. Do this continuously to the quotient until done.

Adding and Subtracting

To add and subtract in binary:

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

To add signs to numbers, it has been standardized that the first bit defines the sign. If the leftmost bit is 0, the number is posititve. and if it is 1, the number is negative.

This allows an n-bit signed integer to be from the range $-2^{n-1} + 1$ to $2^{n-1} - 1$.

To actually add and subtract numbers, we first check if the sign is the same, if it is, we add directly, and append the sign bit. If they have opposing signs, we find the larger number, subtract the smaller, and use the sign of the larger.

Using the complement to conduct operations

In the complement number system, we can add or subtract directly, the operations can be done directly. The complementation process is more complicated than sign checking, but it simplifies addition.

The 2-complement system is the difference between 2^n and the n-bit integer. To compute $-B$, we first need to calculate $2^n - B$, which we can compute as $(2^n - 1) - B + 1$, which is essentially inverting each bit, and adding 1 to the result. (Carries outside the bit length are ignored.)

The most significant bit of the complement acts as a sign bit. This can be seen in the following table: The magnitude of a number can be

number	binary	negative	binary
0	0000	-0	0000
1	0001	-1	1111
2	0010	-2	1110
3	0011	-3	1101
4	0100	-4	1100
5	0101	-5	1011
6	0110	-6	1010
7	0111	-7	1001
-	-	-8	1000

computed as for an unsigned number, except that the weight of the MSB is -2^{n-1} instead of 2^{n-1}. So, to subtract 2^n, replace the weight of the MSB with -2^{n-1}.

To add a bit to a complement number, we can simply duplicate the MSB at the beginning of the MSB. As proof of this actually working, we can calculate the complement of both the complement and the modified complement as uncomplemented numbers, and we can observe that they will be the same.

That is, the complement of 11001 and 111001 is the same number, the same happens for 011 and 0011.

We can also reverse this process if the digits corresponding to the removed bits of the given complement are all the same.

Now we will analyze how the complement can be used to add and subtract.

+6	0110	+
-3	1101	
3 1	0011	

This process may not work if overflow occurs. That is, if the number of necessary bits to represent the result of an operation are more than

the available bits. This can only occur if the numbers being operated on have the same sign.

To subtract two numbers, we instead use the following property: $X - Y = X + (-Y) = X + Y' + 1$ where Y' is the complement of Y.

$$
\begin{array}{ccll}
6 & & 0110 & - \\
2 & & 0010 & \\
\hline
& & 0001 & \\
6 & & 0110 & + \\
2' & & 1101 & \\
\hline
& 1 & 0100 & \\
\hline
4 & & 0100 &
\end{array}
$$

Binary Codes for Decimal Numbers

A set of n-bit strings in which differnt bit strings represent diffferent elements of a set is called a code. A combination on these is a code word.

o through 9 requires at least 4 bits, but there are many methods of doing so.

digit	BCD	2421	excess-3	1-out-of-10
0	0000	0000	0011	1000000000
1	0001	0001	0100	0100000000
2	0010	0010	0101	0010000000
3	0011	0011	0110	0001000000
4	0100	0100	0111	0000100000
5	0101	1011	1000	0000010000
6	0110	1100	1001	0000001000
7	0111	1101	1010	0000000100
8	1000	1110	1011	0000000010
9	1001	1111	1100	0000000001

Another very important representation is the gray code.
By partitioning a disk into eight regions, with n bits with sensors in each, any number between o and $2^n - 1$ can be represented.
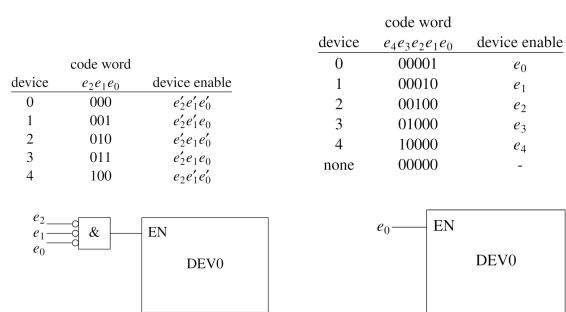
Note: if the disk stops between two sections, there is a possibility of error. Using standard binary representation will cause the error to be very large, as it can stop between 000 and 111, which would imply that any of the bits can have error, and thus any of the numbers between 0 and 7 can be represented. This is where the gray code comes in:

number	gray code
0	000
1	001
2	011
3	010
4	110
5	111
6	101
7	100

This encoding allows there to be only one bit of change between any two consecutive digits, allowing us to reduce the margin of error. Codes can also be used to represent text, as the ASCII does, which uses a 7-bit code word.

The selection of code and code words is very important, as it can drastically change the complexity of the designed circuit:

device	code word $e_2e_1e_0$	device enable
0	000	$e_2'e_1'e_0'$
1	001	$e_2'e_1'e_0$
2	010	$e_2'e_1e_0'$
3	011	$e_2'e_1e_0$
4	100	$e_2e_1'e_0'$

device	code word $e_4e_3e_2e_1e_0$	device enable
0	00001	e_0
1	00010	e_1
2	00100	e_2
3	01000	e_3
4	10000	e_4
none	00000	-

Switching Algebra and Combinational Logic

A combinational circuit depends only on the inputs. We will begin by considering single ouptut combinational circuits.

Switching algebra is a mathematical tool used for circuit design. It is a subset of boolean algebra. A variable is used to represent a signal.

Axioms

- (A1) $X = 0$ if $X \neq 1$

- (A1D) $X = 1$ if $X \neq 0$

There is a principle of duality in this axiom, as do most properties of switching algebra.

- (A2) $X = 0 \implies X' = 1$

- (A2D) $X = 1 \implies X' = 0$

- (A3) $0 \wedge 0 = 0$

- (A4) $1 \wedge 1 = 1$

- (A5) $0 \wedge 1 = 1 \wedge 0 = 0$

- (A6) $1 \vee 1 = 1$

- (A7) $0 \vee 0 = 0$

- (A8) $1 \vee 0 = 0 \vee 1 = 1$

AND has a higher precedence than OR in logical expressions"

$$W \wedge Y \vee Y \wedge Z = (W \wedge X) \vee (Y \wedge Z)$$

Theorems

There are 5 theorems that can be easily be proven through this:

1. $X \vee 0 = X$

2. $X \vee 1 = 1$

3. $X \vee X = X$

4. $(X')' = X$

5. $X \vee X' = 1$

Which have the following dualities:

1. $X \wedge 1 = X$

2. $X \wedge 0 = 0$

3. $X \wedge X = X$

4. No duality

5. $X \wedge X' = 0$

There are also two variable theorems:

- Commutativity:

    1. $X \vee Y = Y \vee X$

    2. $X \wedge Y = Y \wedge X$

- Associativity:

    1. $(X \vee Y) \vee Z = X \vee (Y \vee Z)$

    2. $(X \wedge Y) \wedge Z = X \wedge (Y \wedge Z)$

    This allows for the representation of three or more input gates with ease.

- Distributivity:

    1. $X \wedge Y \vee X \wedge Z = X \wedge (Y \vee Z)$

    2. $(X \vee Y) \wedge (X \vee Z) = X \vee Y \wedge Z$

- Covering:

    1. $X \vee X \wedge Y = X$

    2. $X \wedge (X \vee Y) = X$

- Combining:

    1. $X \wedge Y \vee X \wedge Y' = X$

    2. $(X \vee Y) \wedge (X \vee Y') = X$

- Consensus:

    1. $X \wedge Y \vee X' \wedge Z \vee Y \wedge Z = X \wedge Y \vee X' \wedge Z$

    2. $(X \vee Y) \wedge (X' \vee Z) \wedge (Y \vee Z) = (X \vee Y) \wedge (X' \vee Z)$

All these theorems can be proven using perfect induction, that is, through building the complete truth table for the inputs, and showing that the output is always true.
Some of these can also be proven using the axioms and priorly proven theorems. *Note: The proof of one theorem, can be applied to the dual, but using the duals of the theorems used in the proof of the first.*

- Variable Theorems:

1. $X \vee X \vee X \ldots \vee X = X$

2. $X \wedge X \wedge X \ldots \wedge X = X$

- De Morgan's Theorems:

    1. $(X_1 \wedge X_2 \wedge X_3 \ldots \wedge X_n)' = X_1' \vee X_2' \vee X_3' \ldots \vee X_n'$

    2. $(X_1 \vee X_2 \vee X_3 \ldots \vee X_n)' = X_1' \wedge X_2' \wedge X_3' \ldots \wedge X_n'$

Using De Morgan's Theorems allows us to change combinations of NANDs and NORs into more efficient NAND only or NOR only gate functions.

De Morgan's Theorem Generalization:

$$[F(X_1, X_2, ..., X_n, \vee, \wedge)]' = [F(X_1', X_2', ..., X_n', \wedge, \vee)]$$

- Shannon's Expansion theorems:

    1. $F(X_1, X_2, ..., X_n) = X_1 \wedge F(1, X_2, ..., X_n) \vee X_1' \wedge F(0, X_2, ..., X_n)$

    2. $F(X_1, X_2, ..., X_n) = X_1 \vee F(1, X_2, ..., X_n) \wedge X_1' \vee F(0, X_2, ..., X_n)$

These can be proven through finite induction.

Connecting Switching Algebra and Combinational Logic

We will start by defining some things:

- A *literal* is a variable that can be complemented or uncomplemented, for example X, X', Y, Y'

- A *product term* is a literal or a product (AND) of two or more literals, for example $X, X', X \wedge Y' \wedge Z$

- A *sum-of-products* is a sum (OR) of multiple products.

- A *sum term* is a single literal or the sum (OR) of multiple literals.

- A *product-of-sums* is a product of multiple sum terms.

- A *normal term* is a product or sum in which no variable appears more than once.

- An *n-variable minterm* is a normal product term with n literals. There are 2^n such terms. That is, each variable appears exactly once in the term, in either complemented or uncomplemented form. It is 1 in only one row of the truth table.

- An *n-variable maxterm* is a normal sum term with n literals. Once again, there are 2^n such terms. It is 0 in only one row of the truth table.

- A *canonical sum* of a function is the sum of the minterms corresponding to the rows where the truth-table of the function is 1.

	X	Y	Z	F	minterm	maxterm
0	0	0	0	$F(0,0,0)$	$X' \cdot Y' \cdot Z'$	$X + Y + Z$
1	0	0	1	$F(0,0,1)$	$X' \cdot Y' \cdot Z$	$X + Y + Z'$
2	0	1	0	$F(0,1,0)$	$X' \cdot Y \cdot Z'$	$X + Y' + Z$
3	0	1	1	$F(0,1,1)$	$X' \cdot Y \cdot Z$	$X + Y' + Z'$
4	1	0	0	$F(1,0,0)$	$X \cdot Y' \cdot Z'$	$X' + Y + Z$
5	1	0	1	$F(1,0,1)$	$X \cdot Y' \cdot Z$	$X' + Y + Z'$
6	1	1	0	$F(1,1,0)$	$X \cdot Y \cdot Z'$	$X' + Y' + Z$
7	1	1	1	$F(1,1,1)$	$X \cdot Y \cdot Z$	$X' + Y' + Z'$

This has another notation, in which each row that is true is added to the sum:

$$F = (1,0,0,1,1,1,0,1) = \sum_{X,Y,Z} (0,3,4,5,7)$$

- A *canonical product* of a function is the product of the maxterms corresponding to the truth table rows where the function is 0.

	X	Y	Z	F	minterm	maxterm
0	0	0	0	$F(0,0,0)$	$X' \cdot Y' \cdot Z'$	$X + Y + Z$
1	0	0	1	$F(0,0,1)$	$X' \cdot Y' \cdot Z$	$X + Y + Z'$
2	0	1	0	$F(0,1,0)$	$X' \cdot Y \cdot Z'$	$X + Y' + Z$
3	0	1	1	$F(0,1,1)$	$X' \cdot Y \cdot Z$	$X + Y' + Z'$
4	1	0	0	$F(1,0,0)$	$X \cdot Y' \cdot Z'$	$X' + Y + Z$
5	1	0	1	$F(1,0,1)$	$X \cdot Y' \cdot Z$	$X' + Y + Z'$
6	1	1	0	$F(1,1,0)$	$X \cdot Y \cdot Z'$	$X' + Y' + Z$
7	1	1	1	$F(1,1,1)$	$X \cdot Y \cdot Z$	$X' + Y' + Z'$

This also has another notation:

$$F = (1,0,0,1,1,1,0,1) = \prod_{X,Y,Z} (1,2,6)$$

Note: the union of the terms of the canonical product and the canonical sum, is the full truth table, or the numbers from $0 \rightarrow 2^n - 1$

A function can be written in Verilog by doing the following:

```
case {(X,Y,Z)}
    0,3,4,5,7:F=1;
    default:F=0;
endcase
```

Or it can be written like this:

```
case {(X,Y,Z)}
    1,2,6:F=;
    default:F=1;
endcase
```

Now that we have all the necessary definitions, we will go back to Shannon's Expansion Theorem.

$$F(X_1, X_2, ..., X_n) = X_1 \wedge F(1, X_2, ..., X_n) \vee X_1' \wedge F(0, X_2, ..., X_n)$$

This theorem allows us to extract a variable from a function and divide it into two halves, in which the term is true or false, while the remains have a smaller minterm, maxterm truth table.

Combinational Circuit Analysis

Circuit analysis is defined as locating the logic function that defines it. It is the transition from a logic diagram to a truth table or an algebraic expression.
Once this function is available, it can be used to determine the behavior, or manipulate the circuit for more efficiency, or for different elements. For example, it may be necessary to modify a circuit to transition from PLA to FPGA.
To do so, all functions to be compared are reduced to the canonical form, as it is unique.
One method of running analysis on a logic circuit, we can analyze its output for each of the 2^n inputs it can get, and find the output. However, this is no longer viable for large values of n.
The other method would be to find the output of each gate, starting from the left towards the right of the circuit, each output is joined to find the end function of the circuit. The function can then be minimized and made more efficient.
DeMorgan's theorem can be used to simplify circuits, for example, it can allow us to replace a NAND gate with an OR gate with inverted inputs, or NOR gates with AND with inverted inputs. This is usually done from right to left, replacing gates that we believe would allow us to simplify the circuit. In most cases, the goal would be to reduce the number of inverters within the circuit, as they may complicate the analysis once we have the function.

Combinational Synthesis

This is the process that goes form a truth table or function, and want to get the circuit that defines it.
The goal is to make a circuit as small as possible. This can be done through analyzing how many gates and gate inputs there are.
We also want to minimize delay (gate levels) and power dissipation.
In a design process, the software synthesizes a circuit. However, knowledge is required to make sure the synthesized circuit is efficient, or whether it can be improved.

The process starts from a specification, in words, and leads to a function. However, in this class, we will usually start from the function or truth table.

We want a minimal sum-of-products or product-of-sums. After which, the next goal would be to minimize the delay.
Canonical forms are usually larger than what we want.
If we want to find the canonical form, we conjoin each term of the function with the term that does not appear in both the original and negated form. For example, if we have X,Y,Z as inputs:

$$X \vee Y = X \vee Y \vee (Z \wedge Z')$$

This is done for each term, and then expanded, which will result in the canonical form.
However, it is more complicated to go from the canonical form to the original expression.

Karnaugh Map

Most minimization processes are based on the use of the Combining theorems.
We will use Karnaugh Maps to go from the canonical form to the minimal form. A Karnaugh map is a representation of the truth table that allows us to visualize the function's minimal form

	XY			
	00	01	11	10
Z 0	0	2	6	4
1	1	3	7	5

where the numbers represent the minterms of the function. This can be expanded to more variables, by dividing the number of variables by two.
The order of the bits on each side of the rectangle is the gray code for that amount of bits. That is, after dividing the number of bits by two, to get the number of bits on each side of the rectangle, we will assign the bits, and organize the binary values in gray code order. This allows for only one bit to change between two adjacent cells. This will allow for simlification using the $(A \wedge X \vee (A \wedge X')) = A$.
Note: to generate n-bit gray code the following steps can be followed:

- *Write out 0 n times. This will be your first bit.*

- *Change the least significant but to 1. You will now have* 000...01

- *Reflect this pair of bits, but change the 2-bit to* 1*. You will generate a pair* 00...11,00...10*.*

- *Reflect all the bits you have, and change the next most significant bit from* 0 *to* 1*, until you have all* 2^n *combinations.*
  Example for n=4, with each horizontal line being a reflection:

$$
\begin{array}{c}
0000 \\
\underline{0001} \\
0011 \\
\underline{0010} \\
0110 \\
0111 \\
0101 \\
\underline{0100} \\
1100 \\
1101 \\
\vdots
\end{array}
$$

So, the function $F(X, Y, Z) = \sum_{X,Y,Z}(3, 6, 7)$ would be represented as

		XY			
		00	01	11	10
Z	0	0	0	1	0
	1	0	1	1	0

Let us assume we have a completed Karnaugh map. We will now observe adjacent terms, if there are two adjacent ones, be it vertical (6,7), horizontal (3,7), or across borders (4,0), they can be combined using the combining theorems.

		WX			
		00	01	11	10
YZ	00				
	01	1	1	1	
	11	1	1		1
	10	1			

So, in the above Karnaugh map, any of the following can be combined:

0	0	0	0
<u>1</u>	<u>1</u>	1	0
1	1	0	1
1	0	0	0

0	0	0	0
1	1	1	0
<u>1</u>	1	0	<u>1</u>
1	0	0	0

0	0	0	0
1	1	1	0
<u>1</u>	1	0	1
<u>1</u>	0	0	0

Additionally, if you combine two ones, you can then combine an adjacent pair:

0	0	0	0
<u>1</u>	<u>1</u>	1	0
<u>1</u>	<u>1</u>	0	1
1	0	0	0

So, all four of the above ones can be combined into a single term.
Note: each combination will have 2^n terms.
The goal is to combine as many terms as possible.
This leads to the following rules:

- If combinations cover only areas of the map where the variable is 0, the variable is complemented in the product term.

- If combinations cover only areas of the map where the variable is 1, the variable is uncomplemented in the product term.

- If combinations cover all areas of the map where a variable is either 0 or 1, the term does not appear

We want as few combinations as possible, and these combinations to be as large as possible.
The remaining products are then added together.
The above map will have the following combinations:

0	0	0	0
<u>1</u>	<u>1</u>	1	0
<u>1</u>	<u>1</u>	0	1
1	0	0	0

0	0	0	0
1	<u>1</u>	<u>1</u>	0
1	1	0	1
1	0	0	0

0	0	0	0
1	1	1	0
<u>1</u>	1	0	<u>1</u>
1	0	0	0

0	0	0	0
1	1	1	0
<u>1</u>	1	0	1
<u>1</u>	0	0	0

$W' \wedge Z$		$X \wedge Y' \wedge Z$		$X' \wedge Y \wedge Z$		$W' \wedge X' \wedge Y$	
W	00	W	01	W	10	W	0
X	01	X	11	X	00	X	0
Y	10	Y	0	Y	1	Y	11
Z	11	Z	1	Z	1	Z	10

Each of these resulting combinations will result in a single product.
Based on the aforementioned rules, it will be defined as:

$(W' \wedge Z) \vee (X \wedge Y' \wedge Z) \vee (X' \wedge Y \wedge Z) \vee (W' \wedge X' \wedge Y)$

A logic function implies another if for every function in which the first function is 1, the second function is 1 as well. This is called an implicant.

A prime implicant is one such that if any variable is dropped from its product term would stop it from being an implicant. That is, if the combination was once again combined, it no longer only combines 1's. This leads to the prime implicant theorem, which tells us that a minimal sum is a sum of prime implicants. So, to find a minimal sum we only need to consider it's prime implicants.

It is possible for a function to have more prime implicants than the number of necessary terms in the minimal sum. So, we will add another few definitions:

- A distinguished minterm of a logic function is a minterm covered by only one prime implicant.

- An essential prime implicant is a prime implicant that covers one or more distinguished minterms.

These two definitions allow us to find the minimal sum through the following steps:

1. Find all the prime implicants

2. Identify the distinguished minters and essential prime implicants.

3. Include all the essential prime implicants.

But, this will not assure covering all the ones. So, we make an additional rule:

If P and Q are prime implicants and the number of literals in P is not greater than the number in Q and P covers all the remaining minterms in Q, remove Q.

Some of the remaining may be necessary to select. These are secondary prime implicants.

So we are going to create a prime implicant table:

- A row for every prime implicant

- A column for every minterm

- A mark if the corresponding implicant covers the corresponding minterm

So, for

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

the table would be:

	0010	0110	0111	1001	1101	1111
1-01				X	X	
-111			X			X
0-10	X	X				
011-		X	X			
11-1					X	X

If there is a column with only one mark, it's implicant is selected.
All the marks it covers are selected, and these columns and rows are
removed.

	0010	0110	0111	1001	1101	1111
1-01				X	X	
-111			X			X
0-10	X	X				
011-		X	X			
11-1					X	X

The table above is reduced to:

	0111	1111
-111	X	X
011-	X	
11-1		X

Since -111 covers both the others, they can be removed, and we keep -111.

This process can be conducted several times, until it is as reduced as possible. However, the Karnaugh map is not powerful enough to always get the minimal form. That will require another representation. However, for several cases, it will drastically reduce the number of terms in the prime implicant table.

This process can instead be conducted for zeros, or considering F', and using deMorgan's on the result.

Quine-McCluskey Method

This method is also based on the $A \wedge X \vee A \wedge X' = A$.

We start by listing out all the minterms, for example, for $F(W, X, Y, Z) = \sum(2, 6, 7, 9, 13, 15)$:

2	0010
6	0110
9	1001
7	0111
13	1101
15	1111

Note how the terms are divided by the number of ones in the binary of the minterm.

We will then combine each pair of consecutive sections, all terms with all terms:

2,6	0-10
6,7	011-
9,13	1-01
7,15	-111
13,15	11-1

Two minterms are combined if there is only one bit of difference between the two. The process would be continued if this could be done again, that is, if there were terms in consecutive sections with only one bit difference between two items in two consecutive sections. We will mark every expression that has been combined, and thus

appears in a lower table. If there is one that is not selected, and thus unmarked, it will be a part of the final prime implicant table.

Incompletely Specified Functions

These are functions in which we know what the output should be for certain inputs but don't care what it is for other inputs. To deal with them, the prime implicants can optionally cover them, and occasionally make more efficient expressions.

So, the following map:

0	0	0	0
0	0	1	1
d	1	1	d
1	1	0	0

can have any values at the d, that is, it can be 0 or 1.

So, if we were to select them both as one, the final expression would only have two terms instead of three.

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

$\rightarrow$

0	0	0	0
0	0	1	1
<u>1</u>	<u>1</u>	1	1
<u>1</u>	<u>1</u>	0	0

0	0	0	0
0	0	<u>1</u>	<u>1</u>
1	1	<u>1</u>	<u>1</u>
1	1	0	0

The table that is drawn with incompletely specified functions represents terms we don't care about as dashes.

The prime implicant table will also not include the "don't-care" terms. We will conduct the same process as what was done for earlier prime implicant tables.

Timing Hazards

The output of a gate is not instant, and the delay could possibly cause errors in the output if the delay is severe enough.

Suppose there is a change in an input to a gate, through which there is an expected change in the output, which will be delayed, and can thus cause problems.

This erroneous performance is called a glitch, and the probability of it happening, is called the hazard.

Static Hazards

Static Hazards

The static-1 hazard is the possibility that the circuit will have a momentary 0 pulse when it is supposed to be stable at 1.

It is a pair of input combinations that differ in only one input variable and they both give a 1 output, causing a momentary 0.

The reason for these conditions is that it is possible for only one input to change at a time.

The same can happen for 0, which would be called a static-0 hazard. Only one input changes in a static hazard.

K-maps can be used to locate static hazards. We look for prime implicants that are adjacent, and the change between them is not covered by the circuit.
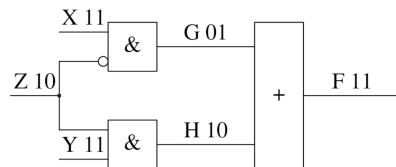
The upper prime implicant and the lower prime implicant, which are each covered by AND gates, have the 11 column as an adjacency. This can cause timing hazards to occur. To fix them, one would be to add the hazard causing adjacency as an additional AND gate to the circuit.

If were to not use a K-map, we would need to compare all pairs of prime implicants and find terms that differ in the value of a single input.

If there are no adjacent K-map terms, or all the items that need to be covered, are already covered, no timing hazard will occur.

Dynamic Hazards

> **Dynamic Hazards**
> They are the possibility of an output changing multiple times based on a single input change.
> This needs to be a multilevel circuit.
> The dynamic hazard occurs because there was a static hazard in one of its inputs. This causes its value to change temporarily before its final output.
> So, in essence, if there is a difference in the beginning output and the end output, and a glitch in the process, it would be a dynamic hazard.

To analyze the hazards on a multilevel circuit, we analyze each of the inputs, and mark where there are changes: The analysis is done with

$$
\begin{array}{c|cc}
W & 0 & 0 \\
X & 0 & 1 \\
Y & 0 & 0 \\
Z & 1 & 1
\end{array}
\rightarrow
\begin{array}{c|ccc}
W & 0 & 0 & 0 \\
X & 0 & x & 1 \\
Y & 0 & 0 & 0 \\
Z & 1 & 1 & 1
\end{array}
$$

the second set of inputs. All the inputs to each gate are analyzed. If there is an x in between a 0 and a 1 at any output, there is no hazard. However, if at some output there is an x in between 1 and 1, or between 0 and 0, there would be a static hazard at that output. Any gate that has this as an input, will then have a dynamic hazard at its output.

Digital Circuits

Static behavior: situations where inputs are not changing.
Dynamic Behavior: situations with changing inputs.
Fan-in: The maximum number of inputs a gate can have in some predefined logic family, is defined as the fan-in of the logic family.
So, for the CMOS family, the fan-in would ideally be ∞, but, because the resistance of the transistors starts mounting, there is a limit.
In the CMOS family, the fan-in for a NAND gate is 6, and for a NOR it is 4.
If we wanted more inputs, we would need to connect multiple gates.

CMOS Static Electrical Behavior

Before, we had: But the graph is actually slanted. This introduces new

V_{in}	V_{out}
0V	5V
5V	0V

parameters that the manufacturer provides.

- V_{OHmin}: The minimum output at HIGH.

- V_{IHmin}: The minimum input that will be recognized HIGH.

- V_{OLmax}: The maximum output at LOW.

- V_{ILmax}: The maximum input to be recognized as LOW.

DC noise margin: amount of noise to corrupt the worst case output to not be recognized properly.

For LOW:

$$V_{ILMax} - V_{OLmax}$$

For HIGH:

$$V_{OHmin} - V_{IHmin}$$

CMOS gates also have something known as leakage current, which is caused by the input consuming current when it would ideally not be doing so.

- I_{IL}: the maximum current flowing when it is LOW.

- I_{IH}: the maximum current flowing when it is HIGH.

If a circuit is not purely CMOS, or if there are high resistances in other parts of the circuit, there would be a need to model the resistance of the output. This causes the outputs to be slightly lower or higher than what it should be at that state.

- I_{OLmax}: Maximum current an output can sink if while maintaining the voltage below V_{OLmax}.

- I_{OHmin}: Minimum current an output needs to maintain the voltage above V_{OHmin}.

Fan-out: Number of inputs the gate can drive without exceeding worst case specs.
It is calculated as: $\min((V_{OLMax}/V_{IL}), (V_{OHmax}/V_{IH}))$.
If the fan-out is surpassed, there are several effects, such as longer delays, noise, etc. that will affect the functionality of the circuit.

CMOS Dynamic Electrical Behavior

This determines the speed and power consumption of the circuit.
Transition Time: amount of time taken to change of one voltage level to another.
Till now, we had modeled circuits with right angles, but transitions are actually slanted, and, in fact, the slope change is also continuous.
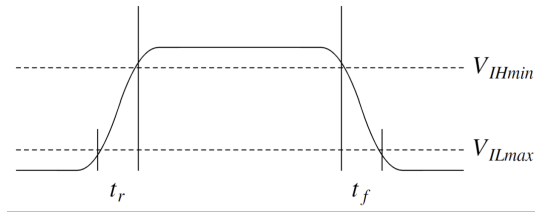
These times depend on the resistances of the transistors, the capacitances in the system, among others.

The following equation will allow us to calculate the delay from low to high.

$$V_{OUT} = V_{CC}e^{-t/(R_n C_L)}$$

The same principle can be used to calculate in the other direction.

Propagation Delay: amount of time it takes for a change in the input signal to cause a change in the output signal.

t_{pHL}: time between an input change and the corresponding output changes from HIGH to LOW.

t_{pLH}: time between an input change and the corresponding output changes from LOW to HIGH.
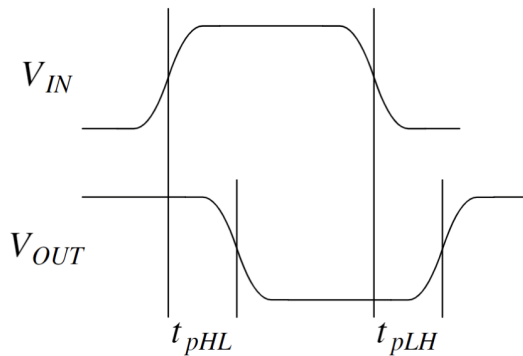
Slight Side topics

Power consumption: If the inputs are not changing, it is called static power consumtpion.

CMOS circuits have very low power consumption.

Decoupling capacitors allow for reduction of current spikes.

Wires or connections may have an inductance, allowing for voltage drops.

Tri-State Outputs

This allows for the creation of a gate that allows for the gate to behave as if it is not in the circuit. That is, it behaves as an open circuit.

This will allow for multiple gates driving a bus, with all but one disconnected, allowing this one to write the output of the bus.

This is essentially adding an enabled/disabled state:
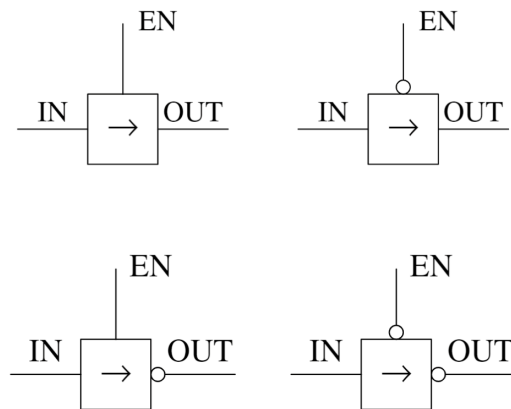
The first of the above gates is created through the following circuit:
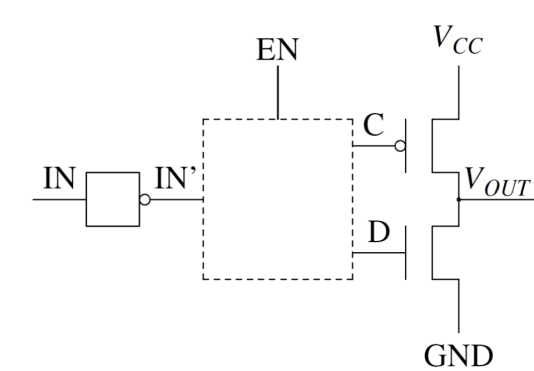
If enable is 0, the resistance of the circuit goes to a very high value, making it not exist to the rest of the circuit.
The "blackbox" in the circuit is the $(C \wedge D') \vee (C \wedge D)'$.
So, in the situation above, one could enable or disable these at will.
When this is disabled, it will act as a third state, that of "nonexistence".

Tri-State outputs are designed so that the output enable delay is longer than the disable delay. This will allow us to avoid there being two or more inputs into a bus to be enabled at the same time.
However, this may not always be possible.
So we create a solution that makes sure there is a disable for everything between any two enables. This will allow us to make sure there is never a conflict due to this.
Bus Transceiver: device used to connect two buses, allowing them to write to each other.
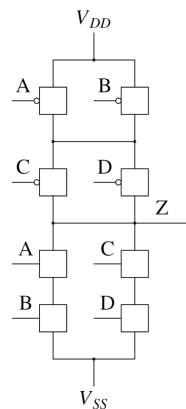The following is an example of transceivers being used:
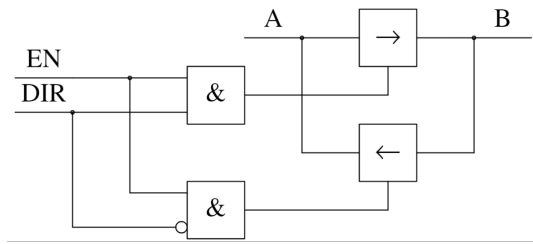
EN-DIR	Mode	Operation
0-x	disconnect	disconnect
1-0	0	A writes to B
1-1	1	B writes to A

There is also a leakage current here, that must be taken into account.

Complex CMOS Gates

If we were to analyze the following circuit:

We must analyze what situations will lead to a connection between Z and V_{DD} or V_{SS}. Series basically represents AND ($\wedge$), while parallel basically represents OR ($\vee$).

So the above function is $(A' \vee B') \wedge (C' \vee D')$, as those are the combinations that will lead to Z to be connected to HIGH, or V_{DD}. Transmission gates are essentially switches controlled by enables.
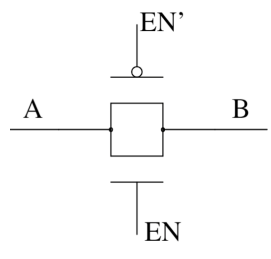
It is a PMOS connected with and NMOS that are connected through complementing control signals. When EN is LOW, both of them are off, and thus the circuit is disconnected. When EN is HIGH, both of them are on, and thus the circuit is connected.

The reason we have two gates, instead of just one to connect A and B, is because NMOS transistors are not good at transmitting HIGH voltages, and PMOS are not good at transmitting LOW voltages. However, together, both HIGH and LOW can be transmitted.

Verilog Hardware Description Language

A Hardware description language (HDL) is the counterpart of a high-level programming language in software design. **It is used for describing hardware**.
It comes with:

- Text Editor: to write, edit and save the program.

- Compiler: Produces intermediate description independent of technology. It also detects syntax errors.

- Synthesis tool: Passes the code into the hardware.

- Libraries: premade importable modules with code to fulfill certain functions used in many places.

Result is a design to be implemented.
If the synthesis is large enough, it is divided into two steps. One is a gate level description, and then the other is the layout description.
There are also:

- Simulators: takes HDL, determines output. Used for design verification. It can also give you intermediate values of the circuit, allowing debugging.

- Timing analyzer: calculates worst case paths and delays of these.

- Power estimation tool: calculates the worst case power consumption of the design.

Verilog Models and Modules

Basic unit of design is a module. It contains declarations and statements.
The declarations describe names and types of inputs, outputs, signals, etc.
Statements are those which manipulate the declarations.

One can specify the module behaviorally, that is it can be given a truth table, and Verilog will figure it out.

It can also be done structurally by describing other modules or individual components, how they are connected, etc. This is equivalent to a logic diagram.

When a module refers to another module it is instantiating it. These modules are defined hierarchically. Higher level modules can instantiate lower level modules as many times as desired. Higher level modules may instantiate the same lower level module.

Every instantiation is a new copy, but if built correctly, the use of a module can be shared.

Definitions inside a module remain local. Modules communicate with each other through declared input and output signals.

Short comments use // and are in effect until newline, longer comments use /* and end at */.

Syntax for module declaration:

```
module module-name(port-name,port-name,...);
    input declarations //list and type of inputs
    output declarations // list and type of outputs
    inout declarations
    net declarations //just a connection
        wire(default), supply0, supply1, tri
    variable declarations //
        reg, integer
    parameter declarations
    function declarations
    task declarations
    concurrent statements // what the module does
endmodule
```

There are words with special meaning, and cannot be used as names, such as module, endmodule, among others.

- **input**: signal that is an input to the module.

- **output**: output signal, cannot be read within module. Can have type net or reg. reg can be assigned values as the code is being processed. That is, it can be modified in the procedural portion of the code. For nets, it is possible to write a reg, modify them, and assign the result to the net.

- **inout**: can be used as both input and output.

- **signal**: single bit number, unless associated with range. Range is defined by [msb:lsb], which is also called a vector.

- **nets**: items to provide connectivity. A signal that is not output or input is assumed to be a wire.
  supply0, supply1 connect to GND and HIGH respectively.
  Tri is used for thee state wires. Can use [msb:lsb] syntax as well.

- **variables**: used during execution, that do not necessarily have physical presence within the circuit. They cannot be changed from outside the module.
  Can be either register (reg), or integer (int). reg can be any of teh below 1-but signal values. int is the 2-complement of a number.
  Numbers can be represented through the following syntax :
  $n'Bdd\ldots d$. The B can be changed for the required base, where:

  - B, b = binary

  - O, o = octal

  - H, h = hexadecimal

  - D, d = decimal

  and the d's are strings of one or more digits in the given base. If the number is longer, the excess is discarded, if it is shorter, the remaining spaces are 0.

- **parameter**: allows constants to be defined The value can be a number or an expression.

A 1-bit signal can have one of four values:

- 0, false

- 1, true

- x, unknown

- z, high impedance

and there are some operators:

- &, AND

- |, OR

- ~, NOT

- ^, XOR
  The above can also be used on buses, performing the operation on all the bits of said bus.

- The [msb:lsb] notation is known as a vector.

- {} is used for concatenation through comma divided signals. One can also multiply inputs and make them appear multiple times, for example {2{byte1}, byte2, byte3}.

- Arithmetic operators:

  - +
  - -
  - *
  - /
  - %
  - **, exponent
  - <<, logical left shift
  - >>, logical right shift
  - <<<, arithmetic left shift
  - >>>, arithmetic right shift

  Each of these will cause a module that can run this operation to be synthesized.

- Logical Operators:

  - &&
  - ||
  - !
  - ==
  - !=
  - >
  - >=
  - <
  - <=

  The bottom 6 create comparator modules.
  Note: if there is one unsigned and one signed number, they will both be considered unsigned, effectively modifying the signed number.

- "Bitwise comparators"

  - ===
  - !==

  Compare the two operands bit by bit. These cannot be synthesized.

- Conditional Operator: <statement> ? <true> : <false>

Compiler directives:

'include <filename> - allows for the file to be immediately read and processed.

'define <identifier text> - allows for repeatedly used expressions to be easily rewritten.

Arrays

Ordered set of variables with the same type. Elements are selected by index. Syntax like the following:

```
reg [7:0] mem [0:255]
```

allows for the creation of a 256 byte array that can be accessed through mem[a][b]

Concurrent Statements

There are three types:

- Instance (structural model):
  Individual gates and other components are instantiated and connected using nets. There are built-in components which need not be instantiated.
  Syntax for instantiation (not declaration):

```
component−name  instance−identifier  (expr,expr,...,expr);
component−name  instance−identifier
     (.port−name(expr),
     .port−name(expr),  ...,
     .port−name(expr));
```

  The first format is the only one allowed for the built-in gates.
  Ports must be given in the order (output, input, input...). If they were three state buffers, the order would be (output, data input, enable input).
  Parameters can be used to parametrize a module to handle inputs and outputs of any width.

```
        module  Maj(OUT, I0 , I1 , I2 );
            parameter WID=1;
            input [WID−1:0] I0 , I1 , I2 ;
            output [WID−1:0] OUT;
            assign OUT = (I0&I1)|(I0&I2)|(I1&I2);
        endmodule
```

  This will turn each of the inputs and outputs into an array of size WID.

When instantiating, one can use the syntax Maj #(8) U1(W,X,Y,Z), and this will change the parameter value to 8.
If we were to have several parameters, we would need to mention which parameter we are changing. For example:

Maj #(.WID(8)) U1(W,X,Y,Z)

- continuous-assignment(dataflow model):
  Syntax:

  ```
  assign net-name: expression;
  assign net-name[msb:lsb] = expression;
  ```

  The right-hand side is evaluated, and then assigned to the left-hand side continuously. That is, it assigns a new value every time the expression changes.
  The order is not important, as they are all re-evaluated. So, all values need to be consistent.

- always, initial (behavioral or procedural):
  Syntax:

  ```
  always @(signal_name or signal_name ...)
      procedural statement
  always @(signal_name, signal_name ...)
      procedural statement
  always @(signal_name, signal_name ...)
      procedural statement
  always @(posedge signal_name)
      procedural statement
  always @*
      procedural statement
  always
      procedural statement
  ```

  These statements execute sequentially, that is, within the statements, the order matters. However, separate statements are evaluated at the same time.
  If any of the signals changes, the statement must be reevaluated. If none of them are changing, the block is not executed and is held suspended.
  If the evaluation of the statement changes any of the input signals, then it will reevaluate the statement. It will keep doing so until none of the inputs have changed.
  When using conditional statements, it is important that every variable is assigned a value in every pass.
  Assignment statements have two syntaxes:

```
variable-name = expression; //combinational
variable-name <= expression; //sequential
```

The first syntax is an instant assignment, while the second does not
assign until the end of the statement.
Another procedural statement syntax is:

```
begin
    procedural statement
    ...
    procedural statement
end
begin block-name
    variable declarations
    parameter declarations
    procedural statement
    ...
    procedural statement
end
```

Begin-end blocks allow us to create procedural statements that
contain several single statements within.
If statements have the following syntax:

```
if(condition)procedural statement
else procedural-statement
```

Any of the procedural statements can have more if statements
within.
There is then a case statement:

```
case(selection-expression)
    choice ,... , choice:procedural statement
    ...
    choice ,... , choice:procedural statement
    default:procedural-statement
endcase
```

Each of the choices is compared with the selection expression, and
the first one that evaluates as true is implemented.
Next are for loops:

```
for(loop-index=first-expr;logical-expression;loop-index=next-expr)
    procedural-statement
```

For synthesis to be possible, a few conditions must be met:

– first-expr must be constant

- logical-expression must be evaluatable

- next-expr must be simple increments or decrements

Other loop statements are repeat, while, forever.

Basic Combinational Logic Elements

There are two perspectives this can be analyzed from:

- Read-only memories (ROM) which allow us to implement any truth table, with a number of inputs and outputs predefined by the ROM.
  When a function has too many inputs, they can be decomposed into smaller functions to make more manageable truth tables. This is the basis for FPGAs.

- Commonly occurring operations (decoding, selecting, comparing) are used to implement functions.

Read-Only Memory

ROM: a combinational circuit with n inputs and b outputs.
It is a two-dimensional array with 2^n rows.
The inputs are stored in a vector A, known as address, and the outputs are stored in the output D, known as data output.
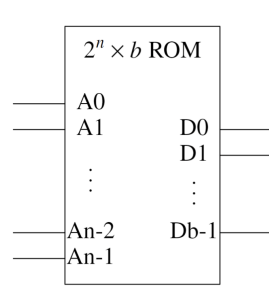
It works by taking in a "number", finding the output stored at the index of the input, and assigning this output to the data outputs.
The ROM needs to be pre-programmed with the truth table.
An FPGA consists of small ROMs, known as lookup tables, which are used to implement logic functions.
A $2^n \times 1$ look-up table can turn any set of n inputs, and give you an output. Smaller values of n allow for this to be viable. So, the actual problem is knowing how to decompose a function into as few LUTs, with as few inputs as possible.

Decoding and Selecting

A decoder allows us to select a component based on its address. n address bits allow us to select certain outputs. The most standard of outputs is to select one from 2^n, known as a binary decoder.

It is, however, far less flexible than a ROM. The specialty of them is that they are not programmable, they are predefined. So, one cannot load a new truth table onto it. The advantage is that it is simpler, using only some gates, and more economic than a ROM.

There are many kinds of codes that can be placed into decoders, such as 1-out-of-10, or gray code, which were mentioned before.

As mentioned before, a binary decoder allows us to select one out of 2^n outputs. It can also have an enable input that will allow us to activate or deactivate the decoder.

Any n-variable function can be represented through an n-to-2^n decoder by collecting all the minterms into an or gate.

It is very wasteful, however, as there are many minterms that are still created, but are left unused.

Decoders can be connected to each other to make higher order decoders. This is done by breaking this into two truth tables of lower order, with all the terms with a selected variable as 0 in one table, and all the ones with that variable as 1 in the other table. Finally, they are both connected to enables, with the 0 truth table's enable connected to the inverse of the selected input, and the 1 truth table's enable connected to the selected variable.

This can be done to increase to even higher orders, with an increase in the inputs to the enables.

This can be implemented in Verilog through the following "code":

```
module VrDecoder(A,EN,Y);
parameter N=3, S=8;
    input [N-1:0] A;
    input EN;
    output reg [S-1:0] Y;
    always @(*)
    begin
        Y=0;
        if(EN==1) Y[A]=1;
    end
endmodule
```

An encoder is the opposite of a decoder. For example, a binary encoder takes 2^n inputs, and turns them into n outputs.

That is, it "enumerates" the 2^n inputs, and the output is the number corresponding to the single input that is turned on, and outputs it in binary.

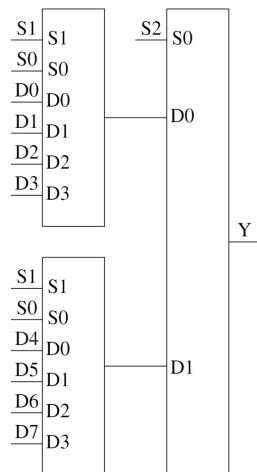*Multiplexers and Demultiplexers*

A multiplexer connects data from one of its inputs to the output. That is, based on the input defined at its select area, it will select one of the inputs to connect to the output.
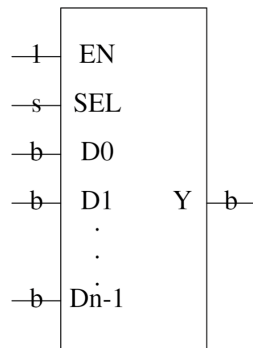
A multiplexer will have nb data inputs (D0, D1, D2, D3, ..., Dn-1), and b outputs(Y). It will have $s = \lceil \log_2(n) \rceil$ SEL inputs to select the input that connects to the output.

This can be simplified if there are less inputs, inputs that relate to each other, or constant inputs. That is, instead of using an nth order multiplexer, we might be able to reduce it to logic gates or to a lower order multiplexer.

Like encoders, multiplexers can be combined, as shown in the following image:

where each of the first layer multiplexers corresponds to S1=1, and S1=0 respectively.

A demultiplexer, on the other hand, is basically a decoder, with the only difference being that, instead of making the output 1, it makes the output equivalent to the input corresponding to the SEL.

Other Combinational Building Blocks

Priority Encoders

Priority encoders are those which "accept requests for service" from multiple devices. As these requests can be made at the same time, it will select those with the highest value, and allow it to write to the bus.

Its equations are defined as following:

$$H_N = I_N$$
$$H_{N-1} = I_N' \wedge I_{N-1}$$
$$H_{N-2} = I_N' \wedge I_{N-1}' \wedge I_{N-2}$$
$$\vdots$$

With I_N being the highest priority input.
Among priority encoders, exist the cascading priority encoders.
If there are groups of priority encoders that we want writing to the same bus we would input each of the devices into their own priority encoder with enables allowing us to decide which of these priority encoders are activated.

XOR gates

An XOR gate can compare two bits for equality. It can be used to add, to compare, to check difference, to count, and to generate random numbers.
It can be implemented through a MUX.
Properties:

- $X \oplus 0 = X$

- $X \oplus 1 = X'$

- $(X \oplus Y)' = X' \oplus Y = X \oplus Y'$

- $X \oplus Y = X' \oplus Y'$

- $X \oplus Y = Y \oplus X$

- $X \oplus Y \oplus Z = (X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)$

The output of an n-input XOR function is 1 iff the number of high inputs is odd.
The k-map of an XOR gate cannot be minimized.
Also:

$$C = A \oplus B \implies A = B \oplus C$$

State Machines

Unlike combinational circuits which depend only on the inputs, sequential circuits depend on current values as well as inputs. Past inputs determine the state of a circuit, which is represented by set variables which have a single bit of **memory** assigned to them.

k state variables correspond to 2^k possible states. However, the memory is limited, and thus, not all combinations will react uniquely. **Synchoronous sequential circuit**: state changes occur at times specified by a clock signal. That is, the changes will occur when the clock goes from LOW to HIGH, of when it goes from HIGH to LOW(negative logic).
There are many types of memory to store values of state varibles. Let us consider positively triggered D flip-flops.
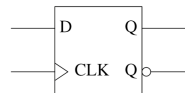
Its operation works as following: A truth table for it can be easily

D	CLK	Q	QN
0	$L \to H$	0	1
1	$L \to H$	1	0
x	0	last Q	last QN
x	1	last Q	last QN

made using current and next state, that is before and after teh rising clock edge.

D	Q	Q*
0	0	0
0	1	0
1	0	1
1	1	1

or

D	Q	Q*
0	x	0
1	x	1

Where Q* is the next state.
Which leads to the following characteristic equation:

$$Q^* = D$$

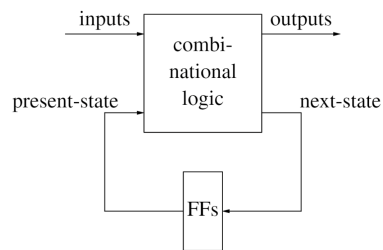Another kind of flip-flop is the S-R flip-flop. which lead to the follow-

S	R	Q	Q*
0	0	0	0
0	0	1	1
0	1	X	0
1	0	X	1
1	1	X	unused

ing characteristic equation:

$$Q^* = S \vee R' \wedge Q$$

State-Machine Structure and Analysis

The general form of a synchronous finite state machine is as follows:

There are kinds of FSMs, for example the Mealy Machine which follows:
next state $= F$(present state, input)
output $= G$(present state, input)

or Moore Machines:
next state $= F$(present state, input)
output $= G$(present state).

Analysis of State Machines

Steps to follow:

1. Determine the next-state and output functions

   (a) Obtain the functions in terms of the flip-flop inputs

   (b) Substitute the functions into the characteristic equations

2. Use these functions to make a transition table that specifies the next state based on the input.

3. Assign names to states and create a step table which has the present state, the next state and the output.

4. (Optional) Draw a state diagram.

A state table allows us to discern the behavior of a circuit directly, without having to analyze the circuit for every possible state and every possible input to every state. All one needs to do is use it as a lookup table based on the input and the state at each moment.

State-Machine Design with State Tables

To instead create a circuit from a state table or function, the following steps are the following.

1. Find a state table that represents the desired function.

2. Check for redundant terms, and eliminate them. That is, if there are two states that perform the exact same function, find a way to remove them. Two states S_i and S_j are distinguishable if there exists an input sequence such that when the machine receives the sequence at state S_i, it produces a different output to a start at state S_j.

   If they are not distinguishable, they are equivalent. The goal of this step is to find and remove equivalent states.

   (a) Check for a sequence of length 0.

   (b) Check for increasing sequence lengths with increments of 1. That is, group all the states that produce the same outputs for the same inputs. If they produce different outputs, they are to be grouped separately.

   (c) Stop when you reach n, that is the number of states or when adding another input does not create any new groups. We stop at n because in the worst case scenario, each step will distinguish a single state.

3. Select number of state variables, and assign combinations of these to the states.

   At least $\lceil \log_2(n) \rceil$ state variables are necessary.

   The steps are

   (a) Replace each item with the binary representation of it. So, for a four state, you would assign 00, 01, 10 and 11 to each of the states.

   (b) Minimize using Karnaugh maps or the Quine-McKluskey method.

   (c) Try other arrangements of assignments of states to find which results in the least number of literals.

The number of state assignments for an n state machine with k bits, is $\frac{2^k!}{(2^k - n)!}$.

4. Derive transition and output tables.

5. Select the type of flip-flop that matches the requirements, and write an excitation table specifying the next term in terms of the current term.

6. Derive logic equations based on excitation table.

7. Perform logic minimization.