

Module for Lab #15:

Introduction to Computer Aided Design

Revision: 2008 November 03 LAB

Overview: (Note that we are not going to do the simulations.)

Computer Aided Design (CAD) tools are an indispensable design resource used by engineers everywhere on a daily basis. Engineers with even basic computer resources can use CAD tools to create a picture-based or text-based definition of a circuit, simulate the circuit, and then implement the circuit in any one of a variety of different technologies. Many CAD tools have been in use for several generations, and are relatively stable and intuitive to use. Other CAD tools are emerging now, with still others just over the horizon. It is safe to say that practicing engineers will need to learn and apply many different CAD tools over their career.

This lab exercise introduces the Xilinx ISE/WebPack Computer Aided Design (CAD) tools. These tools can be used to design and test virtually any digital circuit, and to automatically implement such circuits in programmable chips. The Xilinx tools allow circuits to be defined using several different methods - this exercise introduces a graphics-based program to create circuits, and a logic simulator program to verify the circuit's performance. After design and verification, a chip configuration program can be used to download the circuit to the Digilab circuit board. The tools used in this lab, which include a starter version of the popular ModelSim simulator from Mentor Graphics, can best be learned by following the tutorials available at the class website.

Before beginning this module, you should...

- Be familiar with reading and constructing basic logic circuits;
- Understand logic equations, and how to implement a logic circuit from a logic equation;
- Know how to operate Windows computers and Windows programs.

After completing this module, you should...

- Understand how basic CAD tools are used in basic circuit design;
- Be able to implement any given combinational circuit using the Xilinx ISE schematic editor;
- Be able to simulate any logic circuit using the ModelSim simulator;
- Be able to examine the output of a logic simulator to verify whether a given circuit has been designed correctly.

This module requires:

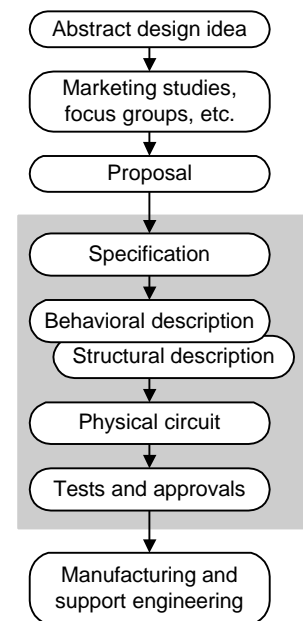
- A Windows PC
- The Xilinx ISE/WebPack software
- A Digilab circuit board

Background

An idea for a new circuit design rarely proceeds directly from concept to flawless implementation. Rather, during the design phase, several potential circuits are considered, constructed, and evaluated. These prototype circuits are intended to give the designer greater insight into a circuit's behaviors and characteristics before a final design is selected. In the early days of digital design, prototype circuits were sketched on paper and then constructed from discrete components or simple integrated circuits (like the 7400 devices used in earlier labs). But over the last 30 years, the use of CAD tools to specify and design digital circuits has made such methods obsolete. With the onset of the computer age, engineers learned they could be far more productive by designing a virtual circuit on a computer instead of actually building it. Now, after several generations of engineers have completed countless designs using CAD tools, they are accepted as a basic and irreplaceable design resource. Their prolific use across all engineering disciplines has allowed new concepts and new technologies to be developed and exploited at an incredible pace. Without their use, it is fair to say technological progress would be crippled. In recent years, CAD tools have become powerful enough to usher in a whole new class of design methods and engineering processes. At the same time, they have become so affordable that virtually any engineer can use them.

The product design process

A new product design process begins with an idea that might arise from any one of several sources, including customers, sales and marketing personnel, or engineers. A new idea that survives the scrutiny and challenges of various marketing studies results in a **proposal**. A proposal document typically describes high level product features, presents target budgets, defines schedules, outlines marketing plans, and generally discusses any useful information. Ideas that make it through the proposal stage enter the engineering design process (indicated by the grayed area of the flowchart). The engineering design process typically starts with a **specification**. A product specification is an engineering document that contains enough information to guide skilled engineers through the design process. Based on the specification, a **behavioral** description, a **structural** description, or some combination of both can be prepared. A behavioral description is essentially a highly detailed specification that states only how a new design is to behave, without providing any information as to how it might actually be built (this is the job of a structural description). For example, a specification for a status indicator on an automobile might be “a *fuel_low* warning light shall be illuminated whenever the fuel tank indicator reads less than 2 gallons for 10 continuous seconds”. A behavioral description might be “*fuel_warning_light* <= check_2s(*under_2_gallons*)”. This behavioral description is written in an easily readable format that clearly indicates a signal named “*fuel_warning_light*” gets assigned a logic value based on the output of a process that evaluates the input signal “*under_2_gallons*”. This behavioral description makes the basic design requirement perfectly clear, but it provides no information to indicate how a circuit might be constructed. In fact, before the circuit can be constructed, this behavioral description must be transformed into a **structural** description. A structural description, such as a circuit schematic showing all components and their interconnections, conveys not only a circuit's behavior, but the information needed to actually construct the circuit as well.



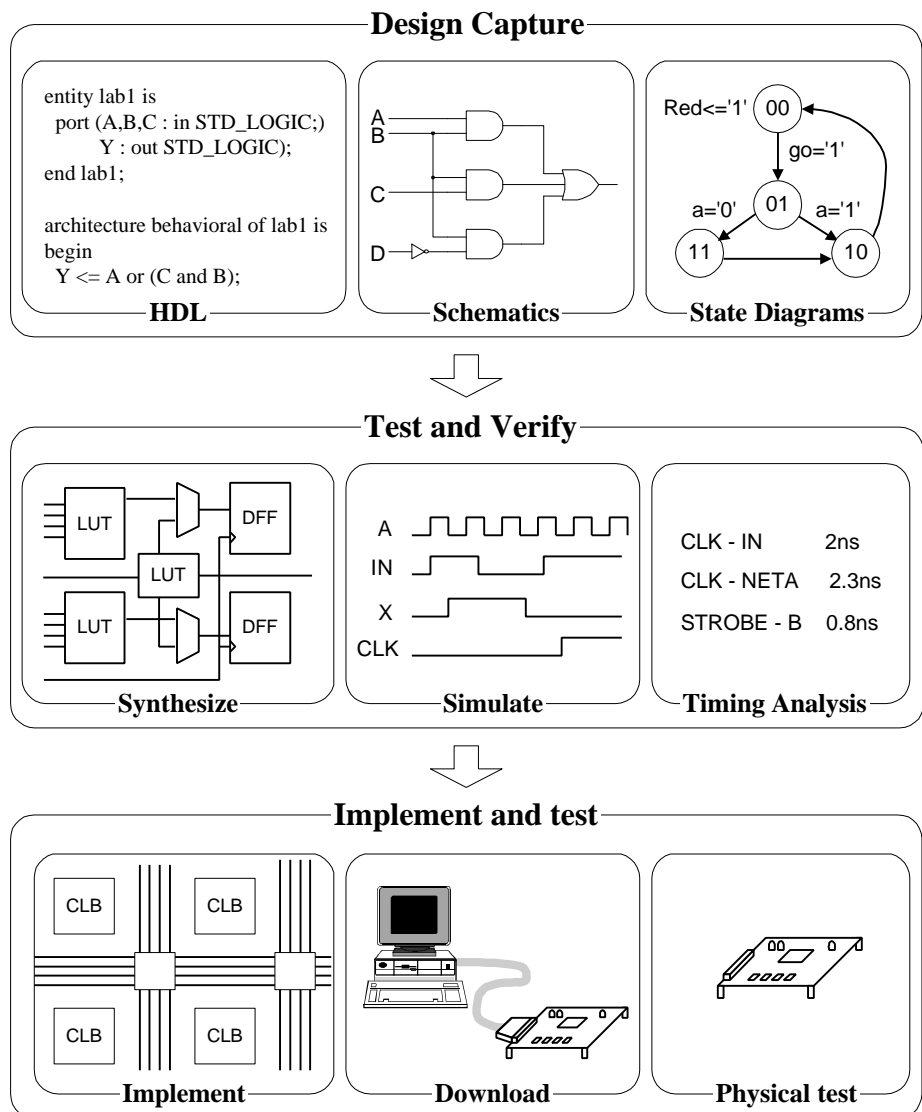
Product design process

This progression from a more abstract behavioral description to a more detailed structural description is a required part of any design process, and may in fact be *defined* as the design process. Even in this simple “warning light” example, the structural definition might take any one of several forms, including a circuit based on a microprocessor, a circuit based on discrete components, or a circuit based on a programmable device. Which form the structural design takes depends on many factors, including the designer’s skills, the cost of various components, the amount of power required by different approaches, etc.

CAD tools are useful throughout the engineering design process, and they benefit simple logic designs and complex system designs alike. In the early stages of a design, CAD tools allow designers to capture circuit definitions on a computer using any one of several different entry modes. Some text-based modes, such as those using a “Hardware Definition Language” or **HDL** editor, allow highly behavioral descriptions. Other picture-based modes, such as those using a **schematic** editor, require highly structural designs. Any given circuit can be entered in most any mode, but significant differences exist. For example, a schematic description that shows all components and interconnections can take significant effort to create, but it yields a description that can be accurately simulated and directly implemented. A

behavioral HDL definition can be quickly entered, but since it contains no information about the structure of a circuit, it must be transformed to a structural representation before a circuit can be implemented.

Much of the work in generating a structural description lies in *drawing* a circuit, and not in *defining* a circuit to meet a given need (i.e., its one thing to sketch a house to meet a family’s needs, but another thing to actually build it). Likewise, transforming a behavioral circuit description to a structural description can require significant work, and this work may not add significant value to the ultimate solution. A class of computer programs called **synthesizers** can perform this work, thereby freeing design engineers to focus on other design tasks. Although synthesizers use rules and



CAD tool framework

assumptions that allow for a wide range of behavioral definitions, several studies have shown that they are nevertheless able to produce structural descriptions that are better than most engineers can produce. HDL editors and synthesizers will be examined in a later lab exercise.

CAD tools allow designers to capture circuits in a convenient manner, using highly evolved tools that significantly reduce labor. They allow captured circuits to be simulated and thoroughly studied before they are actually constructed. They also allow a circuit definition to be implemented in a given technology, so that engineers can readily interact with their “virtual” designs in real hardware. Circuits captured in CAD tools are easily stored, transported, and modified. HDL definitions are largely CAD-tool and hardware platform independent, so that designers can change computing and software platforms. All of these reasons clearly show why CAD tools are used in virtually every new design. But of all of these obvious advantages, one overriding advantage exists: CAD-designed circuits can be simulated. Of all computer-based applications ever developed, it is safe to say that none are more important than circuit simulators.

Circuit Simulators

Constructing circuits from discrete components can be somewhat time consuming, and often of limited value in providing insight into circuit performance. Yet it is difficult to gain confidence in a circuit’s performance without actually testing and measuring its various characteristics. With the advent of modern computers, engineers realized that they could define a “virtual” copy of a circuit in the form of a computer program, and then use that virtual definition to simulate a circuit’s performance without actually building it. Simulators allow engineers to experiment with a circuit design, and challenge it with a wide array of inputs and operating assumptions before undertaking the job of actually building it. Further, complex circuits like modern microprocessors use far too many components to assemble into a prototype circuit – they simply could not have been built without the heavy use of simulators.

Simulators need two kinds of input – a description of the virtual circuit that includes all of the gates (or other components) and interconnections, and stimulus input describing how the circuit’s inputs are to be driven over time. The virtual circuit is entered in to the computer in the form of a “circuit definition language”. Several such languages are currently in use, and they may be divided into two major groups: the “netlist” languages (most popular is the **edif** format); and the “hardware definition languages”, or HDL’s (**VHDL** and **Verilog** are the most popular). For several decades, netlists have been the predominant form of circuit description, but lately, HDL’s are being used more and more. In this lab, we’ll look at netlists and the tools used to create, simulate, and download them to programmable devices.

A netlist is simply a textual description of the components and interconnections in a given circuit. A netlist for a simple circuit might appear as shown to the right. The first entry in each line of the netlist (before the colon) is a label that uniquely identifies a given logic gate or circuit. Next comes the name of the gate and a list of all the inputs and outputs in some predetermined order – in this netlist, the logic gate output is last in the list. Line 2, for example, describes a 2-input NAND gate labeled G2 with inputs *net1* and *a* and output *net2*.

```
G1: INV(sel,net1)
G2: NAND2(net1,a,net2)
G3: NAND2(sel,b,net3)
G4: NAND2(net2,net3,y)
```

Example netlist

Netlists use many different formats, with the "electronic data interchange format" (or edif) being the most popular. Although edif-formatted netlists look somewhat different than this example, they contain the same essential information. Whatever the appearance, the entries in a netlist provide a simulation program with all information needed to simulate the described circuit. In the example shown, you can think of each line as a subroutine call, where the logic function name refers to a particular subroutine and the input/output list provides the subroutine parameters. At each simulation time step, any subroutines whose inputs have changed are executed to compute a new output value. Each newly computed output value might be the input of some other subroutine, and that subroutine would then be executed in a later time step.

To simulate a circuit, a set of stimulus inputs is also required. Often, a sequential list of stimulus commands are collected into a text file, and then given to the simulator (along with the netlist) for a "batch" run. But it is also possible to enter the simulation commands one at a time, and watch the circuit respond in real-time. A set of stimulus inputs may look like those shown in the box to the right.

```
Force a,b,sel to '0'  
simulate 100ns  
Force a to '1'  
simulate 100ns  
Force sel to '1'  
simulate 100ns  
Force b to '1'  
Simulate 100ns
```

Example stimulus

Problem 0: ~~On the submission form, sketch the circuit described by the netlist above, and complete a timing diagram to show the circuit's response to the example stimulus.~~

Schematic Capture

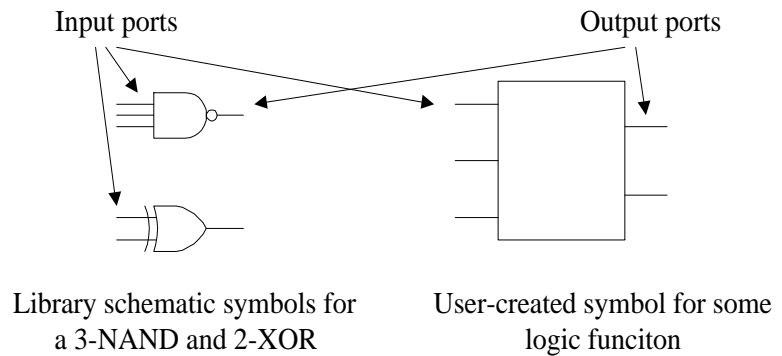
A netlist could be created by hand and typed directly into a computer. But this would be a tedious and laborious practice, even for a moderately complex circuit. First, an accurate and complete circuit sketch would need to be created, then all logic gates and interconnecting nets in a circuit would need to be assigned unique names, and finally the netlist itself, with all components together with a list of all interconnects could be prepared. Note that once a sketch of the circuit is prepared, the remaining tasks are straightforward, repetitious, and time consuming – characteristics that make them well suited to a computer.

A sketch (or a computer-based graphical drawing) of a circuit, with symbols representing logic functions and lines representing interconnecting wires, is commonly referred to as a schematic. A schematic is simply graphical rendition of a netlist, and it is much easier to draw a schematic on a computer than to create a netlist by hand. Computer programs known as "schematic capture tools" allow designers to draw circuits on a computer using a graphical interface. The schematic drawing tool allows symbols representing logic gates (or logic functions) and lines representing wires to be added to a computer-based drawing.

Basic symbols take the shape of recognizable logic gates and functions (NAND's, OR's, INV's, etc.), and more complex functions may appear as simple boxes. Users may also create their own custom symbols to represent logic circuits that they design themselves. Whether a symbol comes from a standard parts library, or whether it is designed by an user, it will have several protruding lines about its periphery representing inputs (generally on the left of the symbol) and outputs (generally on the right of the symbol). Referred to as pins or ports, these inputs and outputs provide connection points

for the lines that represent wires. Although symbols usually do not show ports for power and ground connections, their presence is always assumed.

A circuit is defined in the schematic capture tool by adding symbols and wires until all required components and interconnections are present. Once the schematic is complete, a program called a “netlister” processes the graphical information to produce (or “extract”) the netlist. A schematic must be transformed into netlist representation before it can be simulated. Although the netlist and schematic descriptions of a given circuit look very different from one another, they contain exactly the same information. A one-to-one relationship exists between the schematic and netlist, and it is always possible to convert from one to the other using a simple replacement algorithm. Since it is generally easier for humans to read a circuit schematic than a netlist, circuits are more often shown in schematic form. The process of defining and entering a circuit using a graphical computer tool, and extracting a netlist from the schematic is known as **schematic capture**.



Library schematic symbols for a 3-NAND and 2-XOR

User-created symbol for some logic function

Each circuit symbol has an outline shape and several pins that act as connection points. Many symbols represent common logic functions that can be readily identified due to shape association (and, or, xnor, etc.). Many symbols also appear as rectangular boxes that give no clue as to their function. These non-shape-specific symbols are “wrappers” around circuit blocks that have been designed from more basic logic gates. Circuits grouped into such symbols are commonly called **macros**, and they are frequently used by designers to hide the details of more basic circuits. In this sense, circuit macros are used in schematics in the same way that subprograms are used in computer programs. Circuit macros are most useful when used as building blocks for larger, more complex circuits. Macros are more complex than simple logic gates or circuits, but they are smaller, simpler and easier to understand than the overall circuit. A circuit built from macros is said to be a **hierarchical** circuit, and many levels of hierarchy can be used (i.e., macros can contain other macros as circuit elements). Once designed, macro components can be stored in a project library so that they can be recalled and reused as needed. “I/O markers” are used to identify signals in hierarchical circuits that are meant to be inputs or outputs (as opposed to signals that are limited to internal nodes).

Hierarchical schematic editors allow design complexities to be abstracted away, and hidden inside macros. Macros can be designed and verified independently, often before the overall design is started. Then, they can be used as trusted building blocks for a more complex design. Hierarchical editors allow a “divide and conquer” approach to complex design problems. A primary challenge, and one of the more important design tasks, is to **partition** a design appropriately – a good partition can make a complex task flow relatively smoothly, and a poor partition can create additional work or cause a design to fail.

Associated with each symbol in a schematic, hidden from view, are computer routines that tell a logic simulator program exactly how to model the circuit. A netlister translates the shapes and lines of a schematic into a netlist, and the netlist is essentially a list of calls to these computer routines. Thus, when a schematic is drawn on the screen, the source for a netlist (and therefore, the input to a simulator) is being created as well.

Schematic design flow

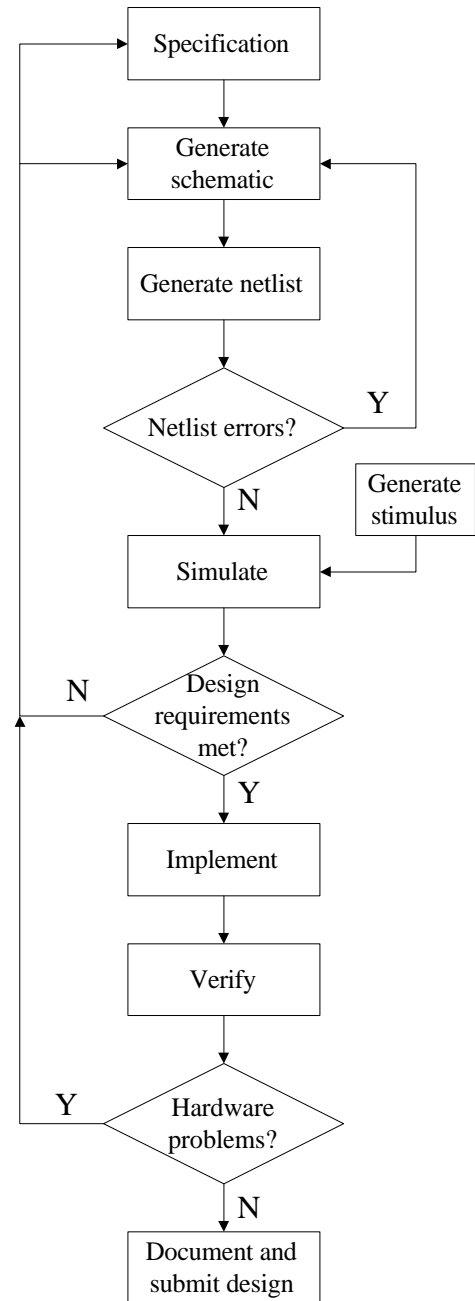
A detailed schematic design flow is shown to the right. The design flow starts with a clear specification, and the specification is used to generate a schematic (and therefore a netlist). The process of generating an error-free schematic and netlist can be somewhat challenging based on the complexity of the design and the features of the CAD tool. Once the netlist is complete, stimulus input can be generated to test the design. In a schematic flow, stimulus inputs can typically be generated using a simple graphical interface in a **waveform editor**. A waveform editor allows signals to be assigned different logic values over time. When all input values have been assigned, the simulation can be executed, and the simulator will produce output values based on the inputs. The output values are typically shown in the same graphic interface window so it is easy to match circuit inputs with the resulting outputs. In general, the simulator inputs should drive the circuit with all possible input conditions so that the designer can verify that the output is correct for every possible combination of inputs. Once the simulation has been executed, the designer must determine whether the simulation results demonstrate that the design requirements have been met. Verifying that the simulation outputs indicate a working design consistent with the specification is probably the most important and challenging process in the design flow.

Problem 3: Create a truth table that corresponds to the simulation shown in the waveform editor window in the submission form.

Once the simulation is correct and all design requirements have been confirmed, the design can be implemented and verified in hardware. This process involves the use of various meters, oscilloscopes, and other test and measurement equipment, all of which are introduced in later labs. The intent of the verification process is to ensure that the design still meets specification after it has actually been constructed in its target hardware. Several problems, such as slow operation, electronic noise, or excessive power consumption may be encountered for the first time after the circuit is actually constructed.

Most modern CAD tools have a top-level graphical interface called a **framework** from which all other required CAD tools can be launched. Such a framework is well illustrated in the Xilinx design tools used in this lab – all the steps that need to be taken from the beginning to the end of a design process are presented in outline form.

You are encouraged to perform the Xilinx tutorial, and/or watch the tutorial video before proceeding.



Schematic design flow

Lab Procedure Part 1: Basic Schematic Capture

Problem 4: Use the schematic capture tool and other required Xilinx CAD tools to enter ~~and simulate~~ the three circuits in the lab submission form. Be sure to add I/O markers to both inputs and outputs (and be sure to change the output signals to *output*). When the circuits have been completed ~~and simulated~~, print a copy of each schematic.

Problem 5: Create macros for all three circuits in problem 3 by selecting the schematic filename in the “Sources in Project” panel of the project navigator, and then double-clicking on “Create Schematic Symbol” in the “Processes for Current Source” panel. Then create a new (fourth) schematic page, and add all three macros as components. Tie all three of the A, B, and C inputs for each macro to single circuit node named A, B, and C in the fourth schematic, and connect the three outputs to three separate nodes named X, Y, and Z (the fourth schematic will therefore have three inputs and three outputs). ~~Simulate the fourth schematic and verify the results.~~ Print a copy of the schematic and ~~the simulation waveform~~, and have the lab assistant inspect your work and sign your lab submission form.

Part 2: A simple logic problem

Problem 6: Design and implement a circuit that meets the following design requirement. Use the Xilinx CAD tools to capture a schematic ~~and simulate the design~~, and then implement the circuit using the discrete chips and the Digilab board. Print the schematic for the design, and have the lab assistant inspect your work.

Amy, Baker, Cathy, and David are responsible for buying new beans for the "Overhead Coffee Company". Buy decisions are made according to the following criteria – a “buy” order is placed if:

- Amy, Cathy, and David vote NO and Baker votes YES,
- or Amy and David vote NO and the rest vote YES,
- or Baker and David vote YES and the rest vote NO,
- or Amy votes NO and the others vote YES,
- or Baker votes NO and the others vote YES,
- or Baker and Amy vote YES and the others vote NO,
- or Cathy votes NO and the others vote YES,
- or David votes NO and the others vote YES,
- or Amy and Cathy vote YES and the others vote NO,
- or they all vote YES.

Although they are good voters, they are unable to consistently tally all votes. Design and implement a simplified logic circuit that will indicate whether they should buy new beans. Use slide switches for vote entry (either "buy" or "not buy"), and an LED to indicate when beans should be purchased. (Hints: One way to solve this problem is to write down a function F in terms of A, B, C, and D from the list above, and then simplify the function using Boolean algebra. Consider using a truth table as well).

Boolean Identities

1 = TRUE

0 = FALSE

\cdot = AND

$+$ = OR

\oplus = XOR

Negation: $\bar{0} = 1$

$$\bar{1} = 0$$

Complementarity: $A \cdot \bar{A} = 0$

$$A + \bar{A} = 1$$

Involution: $\overline{\bar{A}} = A$

Idempotence: $AA = A$

$$A + A = A$$

Identity: $A \cdot 1 = A$

$$A + 0 = A$$

Dominance: $A \cdot 0 = 0$

$$A + 1 = 1$$

Commutativity: $AB = BA$

$$A + B = B + A$$

$$A \oplus B = B \oplus A$$

Associativity: $ABC = (AB)C = A(BC)$

$$A + B + C = (A + B) + C = A + (B + C)$$

$$(A \oplus B) \oplus C = A \oplus (B \oplus C)$$

Distributivity: $A(B + C) = AB + AC$

$$A + BC = (A + B)(A + C)$$

DeMorgan's Laws: $\overline{A + B} = \bar{A} \cdot \bar{B}$

$$\overline{AB} = \bar{A} + \bar{B}$$

Absorption: $A(A + B) = A$

$$A + AB = A$$

Redundancy: $A + \bar{A}B = A + B$

$$A(\bar{A} + B) = AB$$

Consensus: $AB + \bar{A}C + BC = AB + \bar{A}C$

$$(A + B)(\bar{A} + C)(B + C) = (A + B)(\bar{A} + C)$$

XOR (exclusive OR): $A \oplus B = \bar{A}B + A\bar{B}$

EQV (equivalence) aka XNOR: $A \oplus \bar{B} = (A + \bar{B})(\bar{A} + B)$

Examples using Boolean algebra to simplify logical expressions.

$$\begin{aligned}
 1) \quad & \overline{A} \cdot \overline{B} \cdot C + \overline{A+B+C} + \overline{A} \cdot \overline{B} \cdot \overline{C} \cdot D \\
 & = \overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot \overline{C} \cdot D \\
 & = \overline{A} \cdot \overline{B} (C + \overline{C} + \overline{C} \cdot D) = \overline{A} \cdot \overline{B} (1 + \overline{C} \cdot D) = \overline{A} \cdot \overline{B} = \overline{A+B}
 \end{aligned}$$

$$\begin{aligned}
 2) \quad & \overline{A} \cdot \overline{B} \cdot C + \overline{A+B+C} + \overline{A} \cdot \overline{B} \cdot \overline{C} \cdot D \\
 & = \overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot \overline{B} \cdot \overline{C} \cdot D \\
 & = \overline{A} \cdot \overline{B} (C + C + \overline{C} \cdot D) = \overline{A} \cdot \overline{B} (C + \overline{C} \cdot D) = \overline{A} \cdot \overline{B} (C + D)
 \end{aligned}$$

$$\begin{aligned}
 3) \quad & BD + B(D + E) + \overline{D}(D + E) \\
 & = BD + BD + BE + \overline{D}D + \overline{D}E \\
 & = BD + \overline{D}E + BE = BD + \overline{D}E
 \end{aligned}$$

$$\begin{aligned}
 4) \quad & ABCD + \overline{ABCD} + \overline{ABCD} \\
 & = AB(CD + \overline{CD}) + \overline{ABCD} \\
 & = AB + CD
 \end{aligned}$$

$$\begin{aligned}
 5) \quad & AB + (\overline{A} + \overline{B})(C + AB) \\
 & = AB + \overline{AB}(C + AB) = AB + (C + AB) \\
 & = AB + C
 \end{aligned}$$

$$\begin{aligned}
 6) \quad & \overline{A}\overline{B} + \overline{A}\overline{B} + AB\overline{C} \\
 & = (A + \overline{A})\overline{B} + AB\overline{C} = \overline{B} + AB\overline{C} = \overline{B} + A\overline{C}
 \end{aligned}$$

$$\begin{aligned}
 7) \quad & AB + \overline{ABC} + A \\
 & = AB + C + A = A(B+1) + C = A + C
 \end{aligned}$$

also

$$\begin{aligned}
 & = A(B+1) + (\overline{A} + \overline{B})C = A + \overline{A}C + \overline{B}C \\
 & = A + C + \overline{B}C = A + C(1 + \overline{B}) = A + C
 \end{aligned}$$

$$\begin{aligned}
 8) \quad & \overline{AB} + \overline{ABC} + \overline{ABCD} + \overline{ABCDE} \\
 & = \overline{AB} (1 + \overline{C} + CD + \overline{C}DE) = \overline{AB} \cdot 1 = \overline{AB}
 \end{aligned}$$