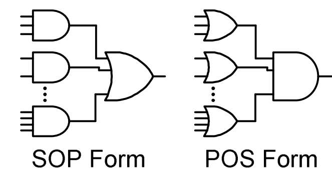
Module 3: Logic Circuit Structure
With an Introduction to Computer Aided Design (CAD) tools

Revision: August 31, 2009

DIGILENT®
www.digilentinc.com
215 E Main Suite D | Pullman, WA 99163
(509) 334 6306 Voice and Fax

Overview

This module presents the basic structure of combinational logic circuits, and introduces the use of computer aided design (CAD) tools in modern circuit design. Combinational logic circuits use networks of logic gates to produce outputs that change in strict relation to input changes; that is, an output can only change state immediately after an input changes state. In a combinational circuit, some input signal changes propagate through the logic gates and interconnections and produce output signal changes, while some input changes may have no effect on outputs; further, the same input patterns will always produce the same outputs. In contrast, outputs from a "sequential circuit", or a circuit that contains memory devices, can change irrespective of input signal changes, and the same input patterns applied at different times can produce different outputs (memory containing circuits are covered in a later module). All combinational circuits can be expressed in two forms: an OR'ing of AND'ed terms, or an AND'ing of OR'ed terms. Since these general forms, called "Sum of Products" (or SOP) and "Product of Sums" (or POS), can be used to express any combinational logic requirement, we will examine them in some detail.



CAD tools are an indispensable design resource used by electrical engineers on a daily basis. They allow engineers to easily create picture-based or text-based circuit definitions, perform circuit simulations, and implement circuits in a variety of technologies. Because CAD tools allow engineers to work with "virtual" circuits before constructing them, more time can be spent studying different solution methods and circuit architectures, and less time on building and rebuilding prototypes. Although CAD tools have been used for generations, they are still being modified and improved on a regular basis. As technologies and methodologies advance, new tools are being developed to take advantage of them. It is safe to say that practicing engineers will need to learn and apply many different CAD tools over their career. This module provides some general discussion about CAD tools, and then introduces the use of Xilinx ISE/WebPack tools. The Xilinx tools can be used to design, test, and implement (in a programmable chip) virtually any digital circuit. They allow circuits to be defined using several different graphical and text-based methods. The graphical method known as "schematic capture" will be covered first, together with a simulation tool that can be used to verify a circuit's performance. A later module will introduce text-based methods.

Before beginning this module, you should…

- Be familiar with reading and constructing basic logic circuits
- Be familiar with logic equations and their relation to logic circuits
- Know how to operate Windows computers and Windows programs

After completing this module, you should…

- Construct a logic circuit from a logic equation
- Understand CAD tool use 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
- Be able to use a logic simulator to verify a given circuit's behavior

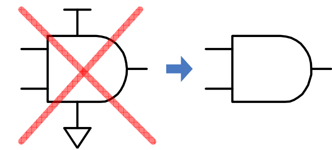
Contains material © Digilent, Inc.

This module requires:

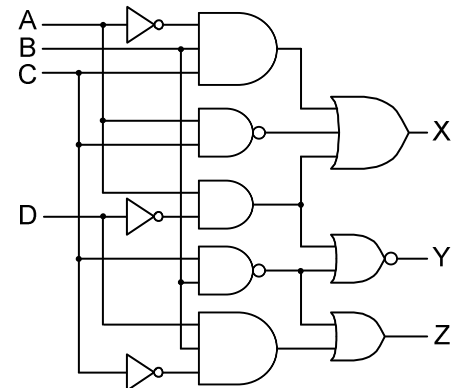
- A Windows PC running the Xilinx ISE/WebPack software
- A Digilent circuit board

Schematics and Prototypes

A schematic is a pictorial representation of a circuit that directly defines circuit structure, and indirectly defines circuit behavior (that is, behavior must be deduced from circuit structure). A schematic is composed of shapes that represent electronic components, lines that represent interconnecting wires, and connector symbols that show external connections. Labels are used to uniquely identify every component, wire, and connector. Symbols used in digital logic circuits are typically restricted to the AND, OR, NAND, NOR, XOR, XNOR, and INV logic gates. Most digital schematics use symbols without Vdd and GND connections, because it is understood that all logic gates require electric power to operate, and showing the power supply connections add no value.



A schematic shows how input signals are combined in logic gates to drive one or more outputs. In the example circuit shown below, the output Z is driven to a logic ‘1’ if B is a ‘0’ or C is a ‘0’, or if B and D are a ‘1’ at the same time that C is a ‘0’. No information is provided to indicate where the inputs originated, or what function the outputs will perform; this is typical in logic circuit schematics, where such details are left for a higher level system description. In the example shown, inputs might arise from switches, sensors, or other logic circuits, and outputs might drive indicators or other circuits.



In a schematic editor, circuits can readily be constructed by assembling graphical shapes that describe logic gates, interconnections, and I/O ports. The completed schematic defines a virtual circuit model, and such models serve two primary purposes: they can be *simulated* so that a circuit’s behavior can be analyzed before it is built; and they can be *synthesized*, or automatically implemented in a real, physical circuit device. The widespread use of simulation and synthesis CAD tools has defined a new and powerful design approach used by virtually all digital design engineers. But it is important to remember that CAD tools work with virtual circuit models, and not with real, physical circuits. Even the most powerful circuit simulators cannot fully model all circuit behaviors, and much about circuit function can only be learned through building and interacting with a physical circuit.

The use of CAD tools greatly simplifies the job of creating a circuit definition that meets the needs of any given design problem. Design problems are typically expressed as a “behavioral” requirement – for example, a design requirement might be to illuminate a warning light if a measured temperature exceeds 90C for 10 seconds, or if coolant level is too low. This worded description describes how a circuit should behave, but it provides no information about circuit structure. A circuit model can be developed to meet the needs of such a behavioral problem statement, and that circuit model can be simulated so that its performance can be compared to the problem requirements. But note that the assumptions used to create the circuit model are verified against the assumptions in the problem statement, and therefore the overall solution is only as good as the assumptions. In any environment or discipline, assumptions are used in place of rigorous knowledge, and they are usually lacking. When circuits are implemented in real, physical devices, their behavior and performance can be thoroughly checked and validated, leaving no room for faulty assumptions – the circuit either works

properly in its intended environment or it does not. It is fair to say that a solution to a given problem is only “proven” after a real circuit has been built and verified. In fact, most design work is performed, and most knowledge is gained, after a virtual circuit model seems complete, and the process of implementing and validating a circuit has begun. Restated, there is no substitute for implementing and working with a real circuit.
Once a design has been proven, it can be implemented in some final target technology. A design destined for an electronic device that might sell millions of units would probably be implemented in a fully custom chip; a design that may sell in the tens of thousands might be implemented in a programmable device; and a design for a low-end, inexpensive toy or novelty might be implemented using discrete components and paper-based circuit boards. In any case, if a circuit prototype is defined in a CAD tool, it is easy to reuse any or all of the source files in the final design.

Beginning with this module and continuing in all subsequent modules, you will learn to use the features of the Xilinx CAD tools to define, simulate, and synthesize circuits. Many of the circuits will be implemented in a Digilent board for validation and verification. In later modules, after you have achieved a level of proficiency with the tools, more information will be presented about CAD tool methods and their use in modern designs.

Combinational Circuit Structure

Combinational logic circuits produce outputs that are some logical function (i.e., AND, OR, NOT, etc.) of their inputs. Any given pattern of inputs to a combinational circuit will always produce the same outputs, regardless of when the inputs are applied. The behavior of combinational logic circuits is most typically identified and specified by a logic equation or by a truth table. Either of these methods provides a clear, concise, and unambiguous definition of how input signals are combined to drive outputs signals.

Logic equations arise naturally when a given worded problem is stated in a more rigorous engineering formalism. For example, the worded problem statement “the latch should be released when the EAST and WEST buttons are pressed simultaneously, or when the NORTH button is pressed provided the WEST button is not pressed at the same time, or whenever the SOUTH button is pressed all by itself” could be cast in a logic equation as:

L <= (E and W) or (N and not W) or (S and not E and not W and not N)

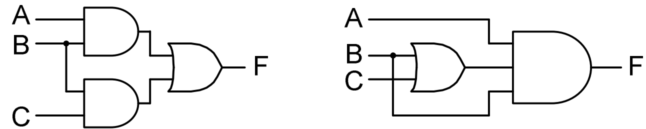
A logic equation states behavioral requirements in a concise, unambiguous fashion. Often, for simple equations (as with the example), a structural circuit can be constructed directly from the equation.

Truth tables are perhaps the most rigorous expression of a combinational logic system, because they define output behavior under all possible combinations of inputs. A truth table for N variables contains 2^N rows, with each row showing a unique pattern of inputs. The rows are typically arranged so that each successive N-bit row is the next binary number in sequence from the preceding row. The truth table on the right shows the input-output behavior of the logic system example above. A circuit schematic can readily be defined from either a logic equation or from a truth table.

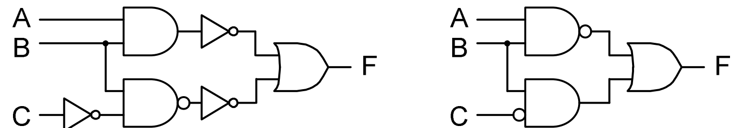
E	W	N	S	L
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

A schematic for a logic equation can be created by substituting logic gate symbols for logical operators, and by showing inputs as signal wires arriving at the logic gates. Perhaps the only step requiring some thought is in deciding which logic operation (and therefore, which logic gate) drives the

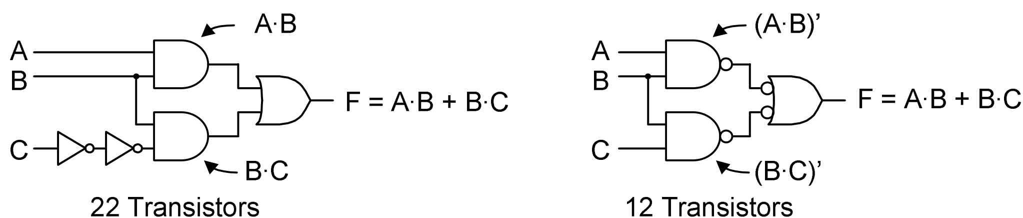
output signal, and which logic operations drive internal circuit nodes. Any confusion can be avoided if parenthesis are used in logic equations to show operator precedence, of if rules of precedence are followed. For example, a schematic for the logic equation "F <= A·B + C·B" might use an OR gate to drive the output signal F, and two AND gates to drive the OR gate inputs, or it might use a three-input AND gate to drive F, with AND inputs coming from the A and B signals directly and a "B + C" OR gate. If no parentheses are used in a logic equation, then INV has the highest precedence, followed by NAND/AND, followed by XOR, and then NOR/OR. In general, it is easiest to sketch circuits from logic equations if the output gate is drawn first. In the figure, the left-most schematic is correct for F <= A·B + C·B.



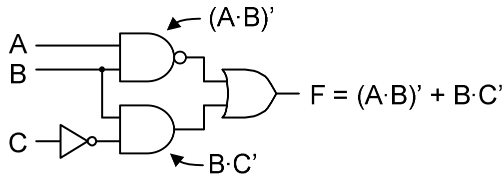
Inversions in logic equations show when an input signal must be inverted prior to driving a logic gate, and also when an output from a logic gate must be inverted. These inversions can map directly to inverters in schematics. For example, in the schematic for the equation F <= (A·B)' + B·C' below left, an inverter is placed on the C input prior to a 2-input NAND gate and on the output of the A·B gate as required by the equation. It is common practice to "absorb" an inverter that follows a logic gate into the gate itself, by placing an inverting "bubble" on the gate output (if one did not exist), or removing an output bubble if one was already present. In fact, using an output bubble instead of an inverter often results in a more minimal CMOS circuit. For example, an inverter following an AND gate represents 8 transistors, while a NAND gate performing the same logic uses only 4 transistors. It is also common to absorb an input inverter into a subsequent logic gate, particularly if the inverted signal only drives one single logic input. The figure on the right above shows an example of absorbing inverters into gate symbols. The meaning of the one-bubble AND gate symbol for B·C' is clear: drive the gate output to a '1' if B is a '1' and C is a '0'.

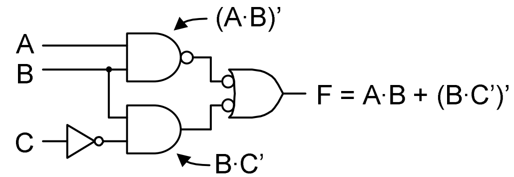


Two "back-to-back" signal inversions cancel each other. That is, if a signal is inverted, and immediately inverted again before it is used anywhere else, then the circuit would perform identically if both inversions were simply removed. This observation can be used to simplify circuits, or to make them more efficient. As an example, consider the circuits below, both of which perform identical logic functions. The circuit on the right has been simplified by removing the two inverters on signal C, and made more efficient by adding inversions on internal nodes so that NAND gates (at four transistors each) could be used instead of AND/OR gates (at six transistors each).



Reading logic equations from schematics is also straightforward. The logic gate that drives the output signal defines the "major" logic operation, and it can be used to determine how other terms must be grouped in the equation. An inverter, or an output bubble on a logic gate, requires that the inverted signal or function output be shown in the output of the "downstream" gate (see example below). A bubble on the input of a logic gate can be thought of as an inverter on the signal leading to the gate.





<u>SOP and POS circuits</u>

The terms “product” and “sum” have long been borrowed from mathematics to describe AND and OR logic operations. A product term is defined as an AND relationship between any number of variables, and a sum term is defined as an OR relationship between any number of logic variables. Any logic system can be represented in two logically equivalent ways: as the OR’ing of AND’ed terms, known as the Sum Of Products (SOP) form; or as the AND’ing of OR’ed terms, known as the Product of Sums (POS) form. The two forms are interchangeable, and one form can be transformed to the other following a few basic rules. As an example, consider the XOR relationship Y_{SOP} <= (not A and B) or (A and not B). This SOP relationship can be expressed in POS form as Y_{POS} <= (A or B) and (not A or not B). In this example, the POS and SOP forms are equally simple, but this is not always the case. For circuits with more than two inputs, it may turn out that one form is simpler that the other. If a circuit is to be constructed, it makes sense to evaluate both forms so that the simplest one can be constructed.

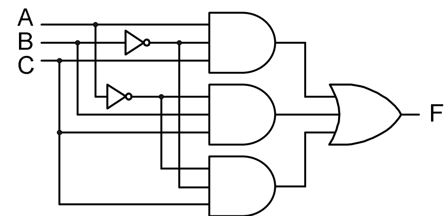
A logic equation (and therefore a logic circuit) can easily be constructed from any truth table by applying the rules presented below.

For SOP circuits:

- A circuit for a truth table with N input columns can use AND gates with N inputs, and each row in the truth table with a ‘1’ in the output column requires one N-input AND gate;
- Inputs to the AND gate are inverted if the input shows a ‘0’ on the row, and not inverted if the input shows a ‘1’ on the row;
- All AND terms are connected to an M-input OR gate, where M is the number of ‘1’ output rows;
- The output of the OR gate is the function output.

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

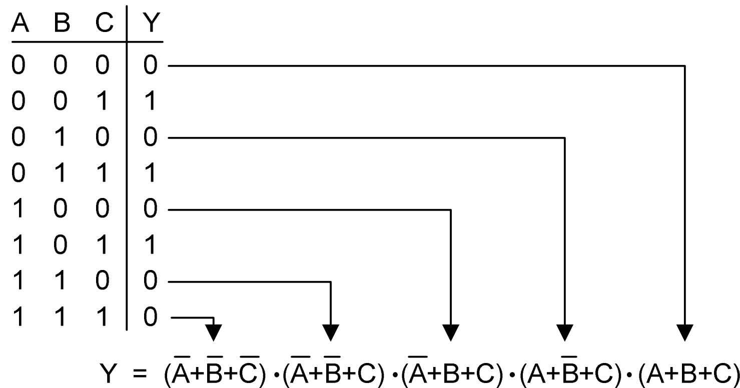
$$Y = A \cdot \overline{B} \cdot C + \overline{A} \cdot B \cdot C + \overline{A} \cdot \overline{B} \cdot C$$

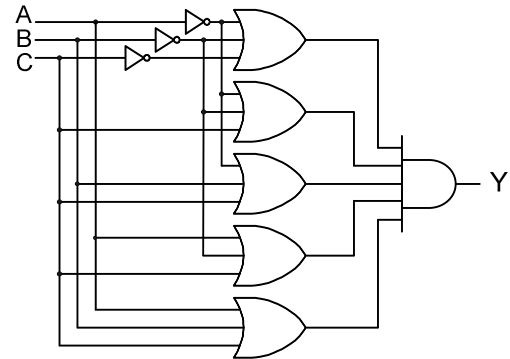


And for POS circuits:

- A circuit for a truth table with N input columns can use OR gates with N inputs, and each row in the truth table with a ‘0’ in the output column requires one N-input OR gate;
- Inputs to the OR gate are inverted if the input shows a ‘1’ on the row, and not inverted if the input shows a ‘0’ on the row;
- All OR terms are connected to an M-input AND gate, where M is the number of ‘1’ output rows;

- The output of the AND gate is the function output.





In the SOP circuit shown above, every product term contains all three input variables. Likewise, in the POS circuit, every sum term contains all three input variables. Product terms that contain all input variables are known as minterms, and sum terms that contain all input variables are known as maxterms. A minterm or maxterm number can be assigned to each row in a truth table if the input 1's and 0's on a given row are interpreted as a binary number. Thus, the SOP equation above (and in the truth table to the right) contains minterms 1, 3, and 5, and the POS equation contains maxterms 0, 2, 4, 6, and 7. In an SOP equation, an input value of '1' creates a non-inverted variable in the minterm (and '0' creates an inverted variable). This defines a minterm code that associates each minterm with a corresponding truth table row. In a POS equation, an input value of '1' creates an inverted variable (and '0' creates a non-inverted variable). This defines a maxterm code that associates every maxterm with a particular truth table row.

A	B	C	#	Minterm	Maxterm	F
0	0	0	0	A'·B'·C'	A+B+C	0
0	0	1	1	A'·B'·C	A+B+C'	1
0	1	0	2	A'·B·C'	A+B'+C	0
0	1	1	3	A'·B·C	A+B'+C'	1
1	0	0	4	A·B'·C'	A'+B+C	0
1	0	1	5	A·B'·C	A'+B+C'	1
1	1	0	6	A·B·C'	A'+B'+C	0
1	1	1	7	A·B·C	A'+B'+C'	0

Using minterm and maxterm codes, it is possible to write a new, compact form of SOP and POS equations that follow directly from a truth table. The SOP equation uses the summation symbol Σ to suggest the summing of terms, and the POS equation uses the symbol Π to suggest taking the product of terms. Both equations simply list the minterms or maxterms present in a given truth table after the initial symbol. Every truth table output row that contains a '1' defines a minterm, and every row that contains a '0' defines a maxterm. Minterm and maxterm equations are shown for the truth table above.

$$F = \Sigma m(1, 3, 5) \qquad F = \Pi M(0, 2, 4, 6, 7)$$

XOR Functions

The Exclusive OR (or XOR) relationship F <= A xor B is defined by the truth tables shown and the equivalent two-variable logic expressions

F_{SOP} <= A·B' + A'·B and F_{POS} <= (A + B)·(A' + B').

The ⊕ symbol is also used frequently for XOR functions, for example, F <= A ⊕ B or F <= A ⊕ B ⊕ C. The XOR function is frequently used in digital circuits to manipulate signals that represent binary

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

2-input XOR

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

3-input XOR

numbers – these circuits will be presented in a later module. For now, note the XOR output is asserted whenever an odd number of inputs are asserted. This "odd detector" nature of the XOR relationship holds for any number of inputs.

Compound XOR functions like $F <= A \oplus (B \cdot C)$ can always be written in an equivalent SOP or POS forms $F_{SOP} <= A' \cdot (B \cdot C) + A \cdot (B \cdot C)'$ and $F_{POS} <= (A + (B \cdot C))(A' + (B \cdot C)')$.

A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

2-input XNOR

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

3-input XNOR

The XNOR function is the inverse of the XOR function. Since the output of a 2-input XNOR is asserted when both inputs are the same, it is sometimes referred to as the Equivalence function (EQV), but this name is misleading, because it does not hold for three or more variables (i.e., the output of a 3-input XNOR is not asserted whenever all three inputs are the same) Truth tables for 2 and 3 input XNOR functions are shown, and it can be seen that for each combination of inputs, the output is the inverse of the XOR truth tables above. The Exclusive NOR (or XNOR) relationship F <= A xnor B shown in the truth tables has the equivalent two-variable logic expressions

$$F_{SOP} <= A' \cdot B' + A \cdot B \text{ and } F_{POS} <= (A' + B) \cdot (A + B').$$

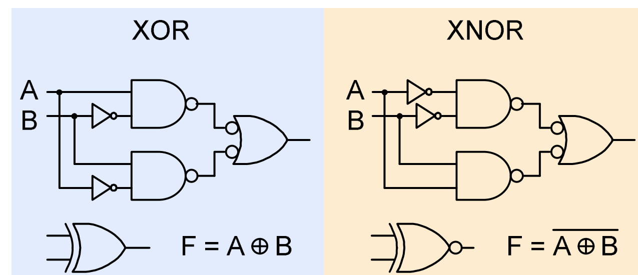
The $\oplus$ symbol is also used frequently for XNOR functions, with the entire expression inverted: $F <= (A \oplus B)'$ or $F <=$ not $(A \oplus B)$; or $F <= (A \oplus B \oplus C)'$ or $F <=$ not $(A \oplus B \oplus C)$.

If either the A or B inputs are in the XNOR truth table are inverted, then XOR outputs are produced; that is, $F <= (A \oplus B)'$ produces the same logic output as $F <= A' \oplus B$ or $F <= A \oplus B'$. If both the A and B inputs are inverted, XNOR outputs are still produced: $F <= (A \oplus B)'$ produces the same output as $F <= A' \oplus B'$. This same property holds for the XOR function – inverting any single input variable will result in XNOR function, and inverting two inputs will again produce the XOR function. In fact, this property can be generalized to XOR/XNOR functions of any number of inputs: any single input inversion changes the function output between the XOR and XNOR functions; any two input signal inversions does not change function outputs; any three input signal inversions changes the function output between the XOR and XNOR functions, etc. More succinctly, inverting an odd number of inputs changes an XOR to an XNOR and vice-versa, inverting an even number of inputs changes nothing, and inverting the entire function has the same effect as inverting a single input. Some representative cases are shown.

$$F = A \text{ xnor } B \text{ xnor } C \Leftrightarrow F <= (A \oplus B \oplus C)' \Leftrightarrow F <= A' \oplus B \oplus C \Leftrightarrow F <= (A' \oplus B' \oplus C)' \text{ etc.}$$
$$F = A \text{ xor } B \text{ xor } C \Leftrightarrow F <= A \oplus B \oplus C \Leftrightarrow F <= A' \oplus B' \oplus C \Leftrightarrow F <= (A \oplus B' \oplus C)' \text{ etc.}$$

An even more succinct description of the XOR and XNOR function outputs can be drawn from the properties discussed. The XOR output is asserted whenever an odd number of inputs are asserted, and the XNOR is asserted whenever an even number of inputs are asserted: the XOR is an *odd* detector, and the XNOR, an *even* detector. This very useful property will be exploited in data error detection circuits discussed in a later module.

XOR and XNOR gate symbols are shown. CMOS circuits for either function can be can built from just 6 transistors, but those circuit have some undesirable features. More typically, XOR and XNOR logic gates are built from three NAND gates and two inverters, and so take 16 transistors.



An useful application of the XOR function is the “controlled inverter” circuit illustrated below. The truth table, derived directly from the XOR truth table, uses an XOR gate with one input tied to a signal named “control”. When control is a ‘1’, the input A is inverted, but when control is a ‘0’, A is simply passed through the logic gate without modification. This controlled inversion function will be useful in later work.

.

A	B	F	Control	A	F	
0	0	0	0	0	0	} ← Pass
0	1	1	0	1	1	
1	0	1	1	0	1	} ← Invert
1	1	0	1	1	0	
XOR truth			**Controlled INV**			

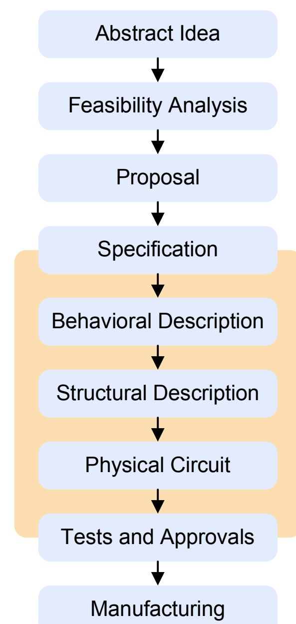
A
Control
F

Introduction to Computer Aided Design Tools

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, and some of them are implemented and evaluated. These prototype circuits help the designer build a greater understanding of the design requirements and possible solutions 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. More recently, CAD tools are used to specify and design digital circuits, rendering pencil-and-paper techniques all but 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

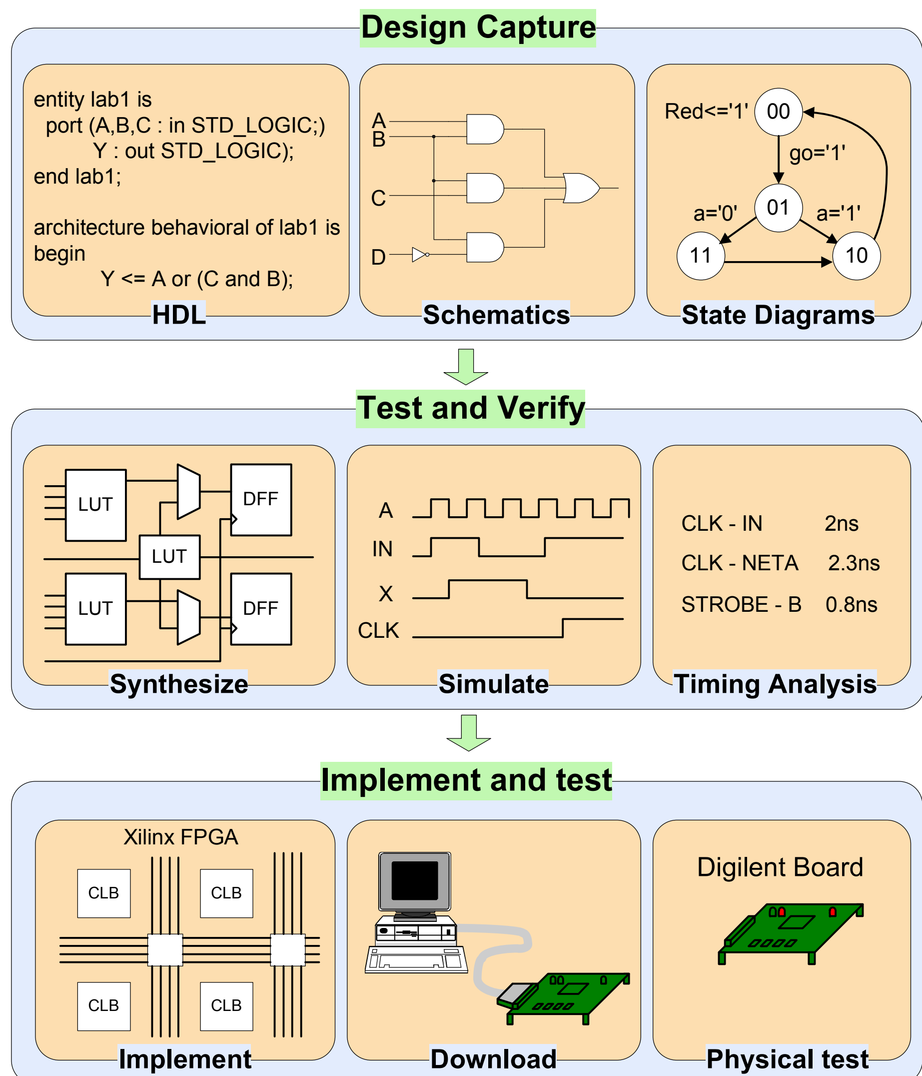
A new product or circuit design process begins with an idea that might arise from any one of several sources, including customers, sales and marketing personnel, or engineers. A new idea that survives the scrutiny and challenges of various feasibility studies typically results in a proposal that describes 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 shaded 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.

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 descriptions. Any given circuit can be described by a behavioral or structural source file, but significant differences exist. For example, a schematic description that shows all components and interconnections can take significant effort to create,



CAD tool framework

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 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 inputs – a description of the virtual circuit that includes all of the gates (or other components) and interconnections, and stimulus input file 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 module, we'll look at netlists and the tools used to create, simulate, and download them to programmable devices. HDLs will be examined in a later module.

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

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

Example netlist

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.*

Netlists use many different formats, with the "electronic data interchange format" (or edif) being the most popular. Although edif-fomatted 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
```

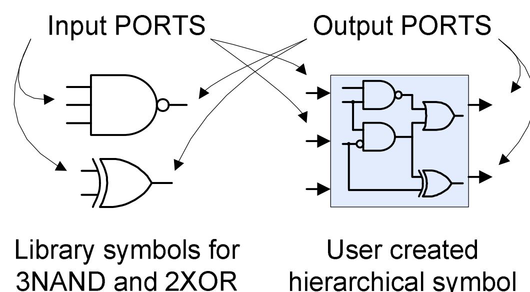
Example stimulus

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



Library symbols for 3NAND and 2XOR

User created hierarchical symbol

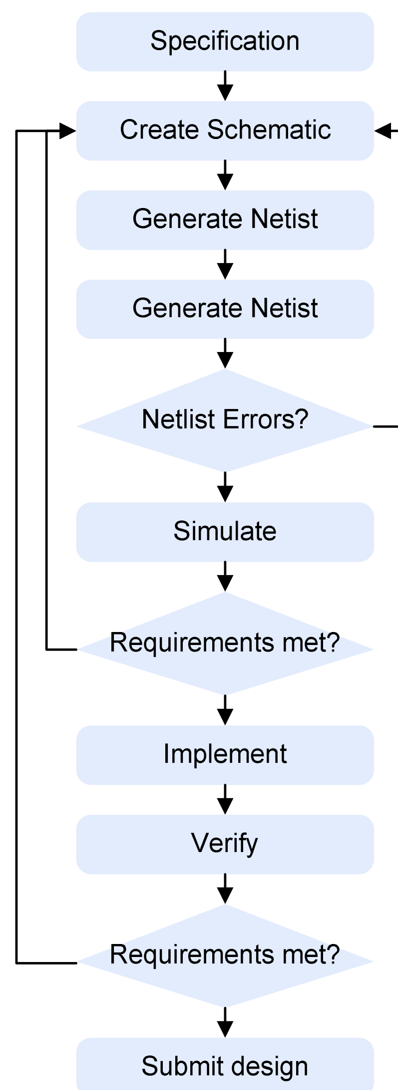
usually do not show ports for power and ground connections, their presence is always assumed.

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.

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



Schematic design flow

netlist (and therefore, the input to a simulator) is being created as well.

Schematic design flow

A detailed schematic design flow is shown in the flowchart above. 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 called 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 one of the more important and challenging process in the design flow.

Once the simulation is correct and all design requirements have been confirmed, the design can be implemented and verified in hardware. This is the most important and telling step in the design process; it is not until the design is made “real” in hardware that all of design requirements can truly be validated. And as importantly, all of the assumptions about the design’s interaction with its intended physical environment can be challenged. For example, if a design is intended to process signals acquired from a sensor input, and perhaps drive an actuator based on the acquired data, the circuit can be placed in this context and real, live data can be processed. In this ultimate proving ground of the design, many behaviors, both anticipated and unexpected, can be observed, validated, and corrected if needed.

Hardware validation often involves the use of various meters, oscilloscopes, and other test and measurement equipment to observe and measure various electronic signals in the design. For example, it is common to check that signal timings meet requirements, that power consumption falls within acceptable limits, or that electronic noise is well contained.

The exercises that accompany this module will reinforce many of the topics presented and provide an opportunity to define digital circuits to meet the requirements of some basic design problems. The associated lab project provides a basic tutorial targeted at first-time users of the Xilinx CAD software.