Software Engineering 2DA4

Slides 2: Introduction to Logic Circuits

Dr. Ryan Leduc

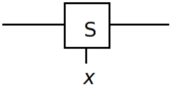
Department of Computing and Software
McMaster University

Material based on S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with Verilog Design, 3rd Ed*.

Variables and Functions

- Basic unit of a circuit is a switch.
- Can be closed (conducts electricity) or open (doesn't conduct).
- Given switch is controlled by input variable x.

(a) Two states of a switch

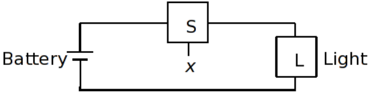


(b) Symbol for a switch

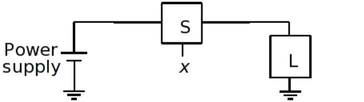
Light controlled by Switch

- We design circuits to implement logic functions.
- We combine basic circuits to create more complicated circuits to implement useful logic functions.
- We can represent the light as logic function $L(x) = x$, where light is on when $L(x) = 1$.

- Represeting the light's state as a function of input x allows us to determine if the light is on based on the current value of x.



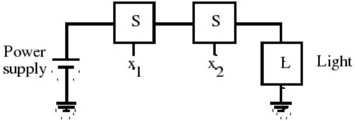
(a) Simple connection to a battery



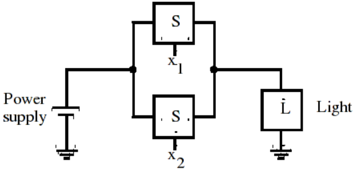
(b) Using a ground connection as the return path

Logical AND and OR Functions

- Here we see two basic building blocks of larger circuits.
- We write the logical AND function as $L(x_1, x_2) = x_1 \cdot x_2$ or $L(x_1, x_2) = x_1 x_2$ if meaning clear.
- We write the logical OR function as $L(x_1, x_2) = x_1 + x_2$



(a) The logical AND function (series connection)

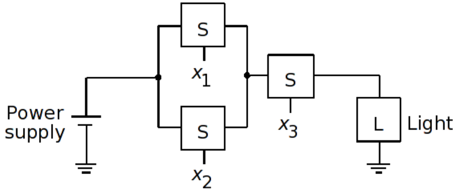


(b) The logical OR function (parallel connection)

Combined Circuit

- Here we combine an AND and OR structure to create a more complicated function.
- Curcuit implements logical function

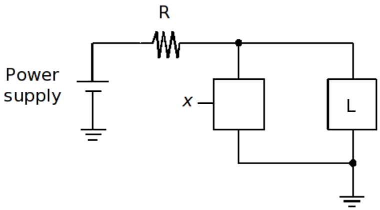
$$L(x_1, x_2, x_3) = (x_1 + x_2)x_3$$



Inverting Circuit

- Here we see the last basic logic function, NOT.
- For NOT, the output function is the logical *negation* or the *complement* of the input variable.
- Circuit implements logical function

$$L(x) = \overline{x} = !x$$



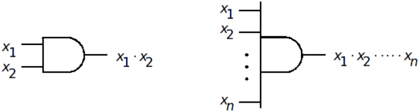
Truth Tables

- Truth Tables are common ways to represent logic functions.
- Any logic function can be completely specified by listing all possible input combinations (valuations) to the left of $||$ divider, and the desired value of the function for that input combination on the right.
- Not efficient representation as n variables will have 2^n possible valuations. i.e. $2^{10} = 1024$ rows!

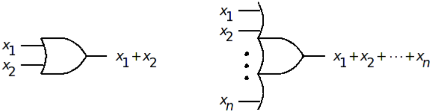
		F_1	F_2
x_1	x_2	$x_1 \cdot x_2$	$x_1 + x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1
		AND	OR

The Basic Gates

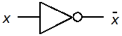
- The AND, OR, and NOT logic functions can be implemented electronically using transistors.
- We refer to these circuit elements as logic gates, and use the symbols below to represent them.



(a) AND gates



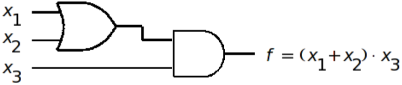
(b) OR gates



(c) NOT gate

Circuits and Networks

- A logic circuit or network is a collection of gates connected to implement a logic function. i.e. the actual items
- A schematic is a drawing of a logic circuit.



Precedence of Operations

- Operations in a logic expression must be performed in the order: NOT, AND, OR.
- eg. $x_1 \cdot x_2 + \overline{x_1} \cdot x_2 = (x_1 \cdot x_2) + ((\overline{x_1}) \cdot x_2)$
- If you wish a different order, you must use parentheses.
- eg. $x_1 \cdot (x_2 + (\overline{x_1})) \cdot x_2$
- The two logical expressions above do not implement the same logic function (consider input valuation $x_1 = 0$ and $x_2 = 1$).

Information for Lab 1

- For information about the DE1-SoC alterra boards used in the lab, refer to *DE1-SoC User manual* that accompanies the boards and is also available as a PDF from the URL: `http://www.cas.mcmaster.ca/~leduc/slides2d04/DE1-SoC_User_manual_ref.pdf`

- **Pg. 7**: Refer to board layout to find switches and LEDs etc.

- **Pg. 23-26**: Board contains 10 toggle switches (SW[0] to SW[9]).

- When the switch is set to its DOWN position (closest to the board edge), you get a logic low (0). The UP position gives logic high (1).

- Table 3-6 shows how each switch maps to a pin on the Cyclone V FPGA chip.

Information for Lab 1 - II

- **Pg. 23-26**: Board contains 10 red LED lights. Each is connected to a pin on the FPGA (see Table 3-8).

- Driving the associated pin to a high logic level turns the LED on, and a low logic level turns it off.

- For step 4 of part 2 of lab, assign inputs of circuit to the pins matching the switches. ie. for x_1 assigned to SW[9], you would map signal x_1 to pin PIN_AE12 of the FPGA.

- Assign outputs to the red LEDs. ie. for f assigned to LEDR[9], you would map f to pin PIN_Y21.

- **NOTE:** Will not always be able to cover information for lab beforehand. You will sometimes have to read ahead on own. Only so many lab periods. Lectures can't always keep up.

CAD Introduction

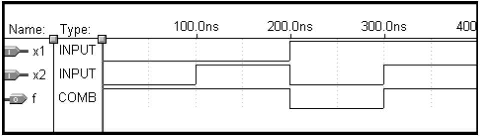
- Computer Aided Design (CAD) systems have tools for the following tasks:
  - Design entry
  - Synthesis and optimization
  - Simulation
  - Physical design

Design Entry Methods

- Truth table as text file or waveforms.
- Schematic Capture.
- Hardware Description Language (HDL). Two IEEE standards:
  - Very high speed integrated Circuit HDL (VHDL)
  - Verilog HDL

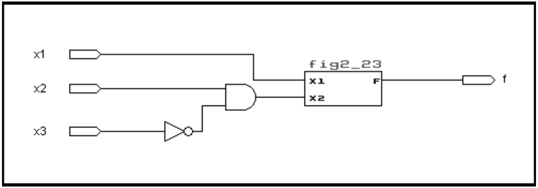
Truth Table as Waveform

- Example uses waveform editor to specify truth table.
- Input signals are x_1 and x_2, and output is f.
- To specify circuit, designer must specify all possible input combinations and desired value of output for each.



Schematic Editor

- In a schematic editor, gates are shown as symbols, and lines show connections.
- Input and output signals to circuit are shown as arrows.
- Schematic editors used to be most common way to design circuits, but now largely replaced by HDL.



HDL

- Replaces schematic capture as standard entry method.
- More portable, and easier to script and scale.
- Similar to a sequential programming language, but describes layout and logic functions of hardware. Unless specified otherwise, everything occurs in parallel.
- Signals in circuits are represented as variables.
- Logic functions expressed by assigning values to variables.
- Will focus on Verilog.

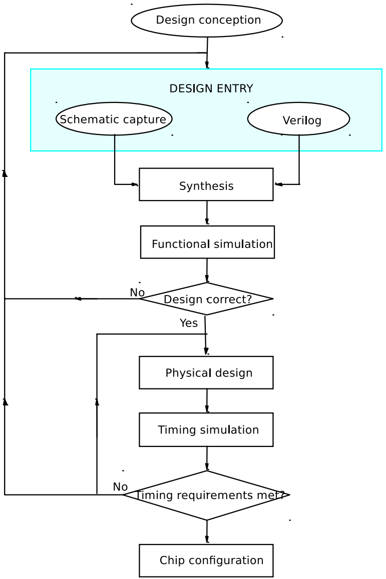
Synthesis and Optimization

- Synthesis is action of generating set of logic equations to represent circuit from truth tables or HDL code.
- Equations are automatically optimized to produce a better yet still equivalent circuit.
- Maps equations to actual technology used (in our case, the Altera chips) for implementation.
- Process called technology mapping and physical design.

Functional Simulation

- Purpose to verify circuit implements specification correctly.
- Accepts specified sequence of input values created using the waveform editor.
- Evaluates outputs of circuits using logic equations from synthesis and input sequences and displays result as a waveform.
- Does not use timing information. Using timing information requires selecting an implementation technology. Called a *timing simulation.*

First Stages of CAD System



Verilog Introduction

- For reference, see Section 2.10 and Appendix A (A.1-A.10,A.15).
- Verilog can be used to describe a circuit, and then CAD tools can synthesize the code into a hardware implementation.
- Important to not write Verilog code that resembles a computer program (i.e. containing many variables and loops).
- You want to write Verilog code so that it is obvious what circuit the code represents.
- Verilog syntax is similar to that of the C programming language.
- Single line comments begin with "//" and multi-line comments begin with "/*" and end with "*/".

Identifiers

- *Identifiers* are names of variables and other items.
- Identifiers can contain letters, digits, and the "_" and "$" characters.
- Identifiers can not begin with a digit and can not be Verilog Keywords.
- Verilog is case sensitive so "BYTE" and "Byte" are not the same name.

Signals

- *Signals* in a circuit are represented in Verilog as either a net or variable.
- A net represents a node (a point where two or more elements interconnect) in the circuit and lets one describe a circuit's interconnection, but not its behavior.
- A variable allows us to describe a circuit's behavior, and can be of type reg or integer.
- A net can be of type wire or tri.
- Type wire is a normal wire connection, and type tri is a special tri-state connection that we will discuss in Appendix B.

Signals - II

- Signal x and y below are a scalar net definition.

  wire x,y;

- S and P are vector definitions, where range $[R_a : R_b]$ defines the value of the most-significant (leftmost) bit (R_a) of the vector, and R_b defines the least-significant (rightmost) bit.

  wire [3:0] S;
  wire [1:2] P;

- For example, if S was assigned the binary constant "0011", then $S[3] = 0$, $S[2] = 0$, $S[1] = 1$, and $S[0] = 1$.

Signal Values and Constants

- A signal can take on four possible values:

$$
\begin{aligned}
0 &= \text{logical value } 0\\
1 &= \text{logical value } 1\\
z &= \text{tri-state (high impedance)}\\
x &= \text{unknown value}
\end{aligned}
$$

- A constant (ie. 'b10, 10, 4'b110) is defined in the form below, where square brackets represent optional parameters.

  [size]['radix]constant

- Here, *size* is number of bits in constant and zeros are usually added (unless x or z is leftmost bit) to left if needed.
- *Radix* is the number base such as ($d =$ decimal - the default), ($b =$ binary), ($h =$ hexadecimal), and ($o =$ octal).

Verilog Circuit Representation

- Verilog allows one to define a circuit using either a structural representation or a behavioral representation.
- A structural representation is when one describes a circuit using constructs that describe individual logic gates and transistors and how they are connected.
- A behavioral representation uses logic expressions and programming constructs to describe how the circuit should operate, but not necessarily its structure in terms of gates and how they are connected.

Structural Representation

- Verilog contains a set of gate level primitives for common logic gates (see Table A.2 in Appendix A.9 for details).

- A gate is defined by giving its functional name, output, and its inputs.

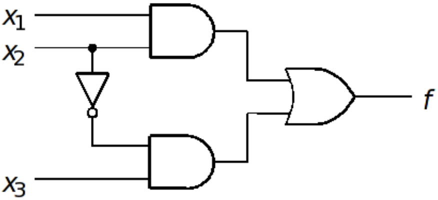
- For example, a two-input AND gate with inputs x_1 and x_2, and output f would be:

$$\textbf{and}(f, x_1, x_2)$$

- A circuit is specified in Verilog as a module that provides the statements that define the circuit.

- A module is given a name, and it may have input and outputs called ports.

Structural Representation e.g. 1

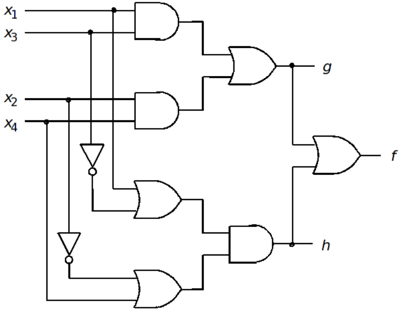



```
module example1 (x1, x2, x3, f);
        input x1, x2, x3;
        output f;

        and (g, x1, x2);
        not (k, x2);
        and (h, k, x3);
        or (f, g, h);

endmodule
```

Structural Representation e.g. 2




```
module example2 (x1, x2, x3, x4, f, g, h);
        input x1, x2, x3, x4;
        output f, g, h;

        and (z1, x1, x3);
        and (z2, x2, x4);
        or (g, z1, z2);
        or (z3, x1, ~x3);
        or (z4, ~x2, x4);
        and (h, z3, z4);
        or (f, g, h);

endmodule
```

Behavioral Representation

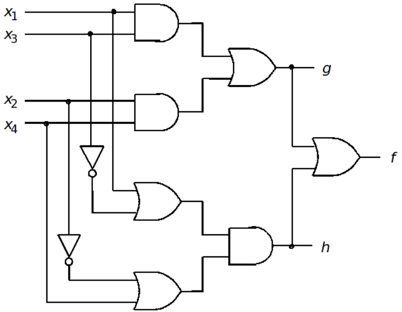
- Using gate primitives would be tedious for large circuits.
- Instead we can use abstract expressions and programming constructs to describe how the circuit should behave.
- For example we can use logical expressions to define the circuit (see Verilog operators in Table A.1 in Appendix A.7).
- The AND operator is "&" and the OR operator is "|".
- The **assign** keyword gives a *continuous assignment* for f.

```
module example3 (x1, x2, x3, f);
        input x1, x2, x3;
        output f ;

        assign f = (x1 & x2) | (~x2 & x3);

endmodule
```


Behavioral Representation e.g. 2




```
module example4 (x1, x2, x3, x4, f, g, h);
        input x1, x2, x3, x4;
        output f, g, h;

        assign g = (x1 & x3) | (x2 & x4);
        assign h = (x1 | ~x3) & (~x2 | x4);
        assign f = g | h;

endmodule
```

Analysis and Synthesis

Analysis: Take a logic network and determine its output function(s).

Synthesis: Design a network to implement a desired logic function.

Analysis

- Using intermediate variables and Truth Tables is one way to analyze a circuit.
- Another way is to draw *timing diagrams*: plots of values of logic variables & functions vs. time.
- Timing diagrams occur in 2 places:
  - using Computer Aided Design (CAD) software to "simulate" circuit.
  - using a logic analyzer in a lab
- This type of analysis allows us to verify that two logic circuits are *functionally equivalent.*
- This means both realize the same logical function. ie. for all input combinations, they will produce the same output value.
- As we will see there are many ways to *implement* a circuit.

Intro to Boolean Algebra

- From example on board, we can see that the two functions $f(x_1, x_2) = \overline{x_1} + x_1 \cdot x_2$ and $g(x_1, x_2) = \overline{x_1} + x_2$ are functionally equivalent.

- Want to be able to start with function f and be able to simplify to g as it is smaller and thus less costly to implement.

- We want to be able to show equivalence without using truth tables.

- One solution is to use Boolean algebra. Provides basis of modern design techniques.

Boolean Algebra Axioms

Axioms 1-4 define truth tables for operators.

	(a)	(b)
1.	$0 \cdot 0 = 0$	$1 + 1 = 1$
2.	$1 \cdot 1 = 1$	$0 + 0 = 0$
3.	$0 \cdot 1 = 1 \cdot 0 = 0$	$1 + 0 = 0 + 1 = 1$
4.	If $x = 0$, then $\overline{x} = 1$	If $x = 1$, then $\overline{x} = 0$

Properties:

	(a)	(b)
5.	$x \cdot 0 = 0$	$x + 1 = 1$
6.	$x \cdot 1 = x$	$x + 0 = x$
7.	$x \cdot x = x$	$x + x = x$
8.	$x \cdot \overline{x} = 0$	$x + \overline{x} = 1$
9.	$\overline{\overline{x}} = x$	

Boolean Algebra Theorems

Properties:

10.(a)	$x \cdot y = y \cdot x$	(Commutative)
(b)	$x + y = y + x$	
11.(a)	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	(Associative)
(b)	$x + (y + z) = (x + y) + z$	
12.(a)	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	(Distributive)
(b)	$x + (y \cdot z) = (x + y) \cdot (x + z)$	
13.(a)	$x + (x \cdot y) = x$	(Absorption)
(b)	$x \cdot (x + y) = x$	
14.(a)	$x \cdot y + x \cdot \overline{y} = x$	(Combining)
(b)	$(x + y) \cdot (x + \overline{y}) = x$	
15.(a)	$\overline{x \cdot y} = \overline{x} + \overline{y}$	(DeMorgan)
(b)	$\overline{x + y} = \overline{x} \cdot \overline{y}$	
16.(a)	$x + \overline{x} \cdot y = x + y$	
(b)	$x \cdot (\overline{x} + y) = x \cdot y$	

NOTE: the textbook added propositions 17.a and 17.b in later editions. We will not be using them in assignments, labs, or tests.

Logic Principles

Principle of Duality: The dual of any true statement (axiom or theorem) in Boolean algebra is also true. It is obtained by:

- Swap all $+$ operators by $\cdot$ operators and vice-versa.
- Swap all 0s by 1s and vice-versa.

For example the dual of $x \cdot 1 = x$ is $x + 0 = x$

DeMorgan's Theorem: $\overline{x \cdot y} = \overline{x} + \overline{y}$

Proving Properties/Theorems

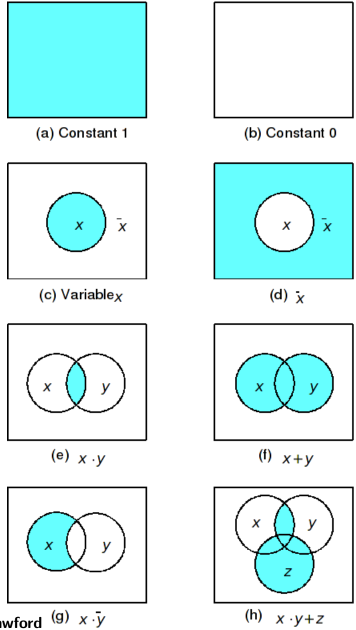
- May be asked to prove properties or theorems.
- One approach is to use truth tables. This is called proof by *perfect induction.*
- For example, prove that **Property 15.a** ($\overline{x \cdot y} = \overline{x} + \overline{y}$) is true.

x	y	$x \cdot y$	$\overline{x \cdot y}$	$\overline{x}$	$\overline{y}$	$\overline{x} + \overline{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0
		LHS		RHS		

Venn Diagrams

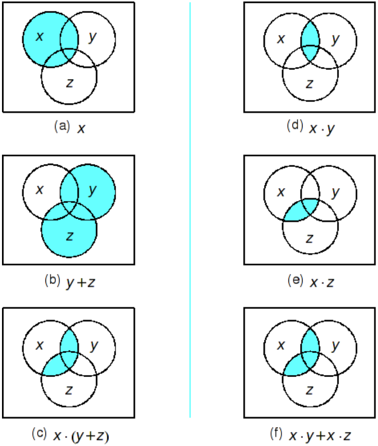
- Another approach is to use Venn diagrams.
- Venn diagrams are a visual aid used to illustrate operations and relations in set algebra.
- In a Venn diagram, the elements of a set are represented by the area enclosed by a contour (i.e. a square or circle).
- Coloured area is portion of region where function is true.

Venn Diagrams - II



Venn Diagram Example

- Verify distributive property, $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$.



Algebraic Manipulation

- A more effective way to prove that a theorem is true, or to simplify an expression, is to use algebraic manipulation.
- This entails using the axioms, theorems, and properties to transform an expression, step by step, into another, equivalent expression.
- At each step you **MUST** cite which axioms etc. you use or you will lose a lot of marks on assignments/tests.
- **Example:** prove property 13a: $x + xy = x$

$$
\begin{aligned}
\textbf{LHS} &= x + xy \\
&= x(1+y) \quad \textbf{12a} \\
&= x \cdot 1 \quad \textbf{5b} \\
&= x \quad \textbf{6a} \\
&= \textbf{RHS}
\end{aligned}
$$

Algebraic Manipulation Example

- Use algebraic manipulation to minimize the function below:

$$
\begin{aligned}
f &= \overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}\,z + \overline{x}\,y\,\overline{z} + \overline{x}\,yz + xy\overline{z} + xyz \\
&= \overline{x}\,\overline{y}(\overline{z} + z) + \overline{x}\,y(\overline{z} + z) + xy(\overline{z} + z) \quad \textbf{12a} \\
&= \overline{x}\,\overline{y} + \overline{x}\,y + xy \quad \textbf{8b, 6a} \\
&= \overline{x}(\overline{y} + y) + xy \quad \textbf{12a} \\
&= \overline{x} + xy \quad \textbf{8b, 6a} \\
&= \overline{x} + y \quad \textbf{16a}
\end{aligned}
$$

- This approach not practical for complex expressions.

- Method is the basis for automating synthesis of logic functions in CAD tools.

Algebraic Manipulation Example 2

- Use algebraic manipulation to minimize the function below:

$$
\begin{aligned}
f &= (x_1 + x_2 + x_3) \cdot (x_1 + \overline{x_2} + x_3) \cdot (\overline{x_1} + \overline{x_2} + x_3) \cdot \\
&\quad (x_1 + x_2 + \overline{x_3}) \\
&= (x_1 + x_2 + x_3) \cdot (x_1 + x_2 + \overline{x_3}) \cdot (x_1 + \overline{x_2} + x_3) \cdot \\
&\quad (\overline{x_1} + \overline{x_2} + x_3) \quad \textbf{10a} \\
&= ((x_1 + x_2) + x_3) \cdot ((x_1 + x_2) + \overline{x_3}) \cdot (x_1 + (\overline{x_2} + x_3)) \cdot \\
&\quad (\overline{x_1} + (\overline{x_2} + x_3)) \quad \textbf{11b} \\
&= ((x_1 + x_2) + x_3) \cdot ((x_1 + x_2) + \overline{x_3}) \cdot ((\overline{x_2} + x_3) + x_1) \cdot \\
&\quad ((\overline{x_2} + x_3) + \overline{x_1}) \quad \textbf{10b}
\end{aligned}
$$

Algebraic Manipulation Example 2 - II

$$= \quad ((x_1 + x_2) + x_3) \cdot ((x_1 + x_2) + \overline{x_3}) \cdot ((\overline{x_2} + x_3) + x_1) \cdot ((\overline{x_2} + x_3) + \overline{x_1}) \quad \textbf{10b}$$

- Want to use property 12b: $x + y \cdot z = (x + y)(x + z)$
- Important to realize that terms in a theorem can mean a variable, or an expression.
- Can apply to: $((x_1 + x_2) + x_3) \cdot ((x_1 + x_2) + \overline{x_3})$ by taking $x = (x_1 + x_2)$, $y = x_3$, and $z = \overline{x_3}$

$$\begin{aligned} &= \quad ((x_1 + x_2) + x_3\overline{x_3}) \cdot ((\overline{x_2} + x_3) + x_1\overline{x_1}) \quad \textbf{12b} \\ &= \quad (x_1 + x_2) \cdot (\overline{x_2} + x_3) \quad \textbf{8a, 6b} \end{aligned}$$

Synthesis

- We can generate or *synthesize* a circuit from truth table:

x_1	x_2	x_3	f
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	1
1	1	1	1

- For the above truth table, an equivalent logic function is: $f = x_1 x_2 + x_3$.
- How was it derived? That's our next topic..

Synthesis Intro

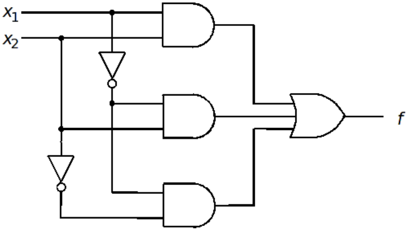
- Have function that monitors inputs x_1 and x_2 such that $f = 1$ when $(x_1, x_2) = (0,0), (0,1), or (1,1)$, otherwise $f = 0$.
- This gives us:

x_1	x_2	f
0	0	1
0	1	1
1	0	0
1	1	1

- How can we represent this as a function? We know f is true for all input combos but 1.
- For each input valuation that $f = 1$, we can find a term that is true only for that input combo. ie. $(x_1, x_2) = (0, 1)$ can be represented as $\overline{x_1}x_2$.
- We can then OR these three terms together.
- $f = \overline{x_1}\,\overline{x_2} + \overline{x_1}x_2 + x_1x_2$

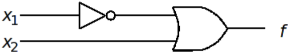
Synthesis Intro - II

- Below is $f = \overline{x_1}\,\overline{x_2} + \overline{x_1}x_2 + x_1x_2$ implemented as a circuit.



- Can show that $g = \overline{x_1} + x2$, implemented below, is functionally equivalent to f.

- Which is better? Not always obvious so we use a cost metric.



Circuit Cost

- **Defn:** The *cost* of the circuit is the sum of the logic gates added to the sum of their inputs.
- Cost $=$ # gates $+$ # inputs
- Unless told otherwise, ignore the cost of inverters. Why? see Appendix B.

f: cost $= 4 + 9 = 13$

g: cost $= 1 + 2 = 3$

Synthesis: Sum-of-Products

Literal: A variable in its uncomplemented or complemented form (ie. A, $\overline{A}$, B).

Product term: One literal, or 2 or more literals ANDed together (ie. A, $x_1\overline{x_2}x_3$).

Minterm: For a function of **n** variables, a *minterm* is a product term containing each of the **n** variables only once.

Sum-of-products expression: Expression formed by combining product terms with the "+" operator (ie. $A + x_1\overline{x_2}x_3$)

Canonical sum-of-products

Canonical sum-of-products: A sum-of-products expression for a function consisting only of minterms. Unique to a truth table.

Procedure:

- Identify rows of truth table where $f = 1$
- Form minterms for these rows. If variable is zero in valuation, then use complement in minterm, else variable. ie, if $x = 0$ in valuation, then use $\overline{x}$, else x.
- Create sum-of-products of these minterms

Sum-of-Products Example

- Derive canonical sum-of-products from truth table.

#	x_1	x_2	x_3	h	minterm	label
0	0	0	0	0		
1	0	0	1	1	$\overline{x}_1\overline{x}_2x_3$	m_1
2	0	1	0	0		
3	0	1	1	1	$\overline{x}_1x_2x_3$	m_3
4	1	0	0	0		
5	1	0	1	1	$x_1\overline{x}_2x_3$	m_5
6	1	1	0	1	$x_1x_2\overline{x}_3$	m_6
7	1	1	1	1	$x_1x_2x_3$	m_7

$$\begin{aligned} h &= \overline{x}_1\overline{x}_2x_3 + \overline{x}_1x_2x_3 + x_1\overline{x}_2x_3 + x_1x_2\overline{x}_3 + x_1x_2x_3 \\ &= m_1 + m_3 + m_5 + m_6 + m_7 \\ &= \Sigma(m_1, m_3, m_5, m_6, m_7) \\ &= \Sigma m(1, 3, 5, 6, 7) \end{aligned}$$

- cost $= 6 + 20 = 26$

Product-of-sums Intro

- Earlier, we found the canonical sum-of-products for the truth table below:

$$f = \overline{x_1}\,\overline{x_2} + \overline{x_1}x_2 + x_1x_2$$

x_1	x_2	f
0	0	1
0	1	1
1	0	0
1	1	1

- If we took $h = \overline{f}$, it would be true when $f = 0$. The sum of products for h is:
- $h = x_1\overline{x_2}$
- We can derive an expression for f as follows:
- $f = \overline{h} = \overline{(x_1\overline{x_2})} = \overline{x_1} + x_2$
- Called product-of-sums form. Want to be able to derive from truth table

Synthesis: Product-of-sums

Sum term: One literal, or 2 or more literals ORed together (ie. A, $(x_1 + \overline{x_2} + x_3)$).

Maxterm: For a function of **n** variables, a *maxterm* is a sum term containing each of the **n** variables only once.

Product-of-sums expression: Expression formed by combining sum terms with the "$\cdot$" operator (ie. $A \cdot (x_1 + \overline{x_2} + x_3)$)

Canonical product-of-sums

Canonical product-of-sums: A product-of-sums expression for a function consisting only of maxterms. Unique to a truth table.

Procedure:

- Identify rows of truth table where $f = 0$
- Form maxterms for these rows. If variable is one in valuation, then use complement in maxterm, else variable. ie, if $x = 1$ in valuation, then use $\overline{x}$, else x.
- Create product-of-sums of these maxterms.

Product-of-Sums e.g.

- Earlier, we evaluated the sum-of-products for truth table. Now we find the canonical product-of-sums.

#	x_1	x_2	x_3	f	maxterm
0	0	0	0	0	$x_1 + x_2 + x_3$
1	0	0	1	1	
2	0	1	0	0	$x_1 + \overline{x}_2 + x_3$
3	0	1	1	1	
4	1	0	0	0	$\overline{x}_1 + x_2 + x_3$
5	1	0	1	1	
6	1	1	0	1	
7	1	1	1	1	

$$\begin{aligned} f &= (x_1 + x_2 + x_3)(x_1 + \overline{x}_2 + x_3)(\overline{x}_1 + x_2 + x_3) \\ &= M_0 \cdot M_2 \cdot M_4 = \Pi(M_0, M_2, M_4) = \Pi M(0, 2, 4) \end{aligned}$$

cost $= 4 + 12 = 16$

Design Steps

1. Specify desired behavior of circuit.
2. Synthesize circuit and optimize.
3. Implement circuit.
4. Verify circuit - if incorrect, go back to step 2.

Design Example: Multiplexor

- **Word Description:** You have two data sources, x_1 and x_2, and one output, f.
- We want to use a third input, s, to select which input is transmitted to the output.
- If $s = 0$, then f has the same value as x_1.
- If $s = 1$, then f has the same value as x_2.
- This type of circuit is called a multiplexor.

Design Example: Multiplexor - II

- Step 1: create truth table from word problem.

#	s	x_1	x_2	f	minterm
0	0	0	0	0	
1	0	0	1	0	
2	0	1	0	1	$\overline{s}x_1\overline{x}_2$
3	0	1	1	1	$\overline{s}x_1x_2$
4	1	0	0	0	
5	1	0	1	1	$s\overline{x}_1x_2$
6	1	1	0	0	
7	1	1	1	1	sx_1x_2

- Step 2: Synthesize. Determine minterms and form s-of-p. We then optimize to reduce cost of circuit.

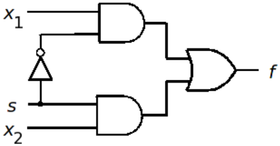
$$\begin{aligned} f &= \overline{s}x_1\overline{x}_2 + \overline{s}x_1x_2 + s\overline{x}_1x_2 + sx_1x_2 \\ &= \overline{s}x_1(\overline{x}_2 + x_2) + sx_2(\overline{x}_1 + x_1) \quad \text{12a} \\ &= \overline{s}x_1 + sx_2 \quad \text{6a,8b} \end{aligned}$$

Design Example: Multiplexor - III

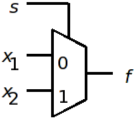
- Step 3: implement the circuit.

$s\ x_1\ x_2$	$f(s, x_1, x_2)$
000	0
001	0
010	1
011	1
100	0
101	1
110	0
111	1

(a)Truth table



(b) Circuit



(c) Graphical symbol

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

(d) More compact truth-table representation

Design Example: Multiplexor - IV

- Step 4: verify circuit by analyzing implementation. Take point A to be $\overline{s}$, point B to be $\overline{s}x_1$ and point C to be sx_2 and fill in the truth table below:

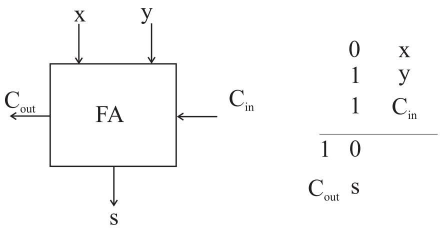
#	s	x_1	x_2	A	B	C	f'
0	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
2	0	1	0	1	1	0	1
3	0	1	1	1	1	0	1
4	1	0	0	0	0	0	0
5	1	0	1	0	0	1	1
6	1	1	0	0	0	0	0
7	1	1	1	0	0	1	1

- Must have column for each unique signal in circuit.

- f' column identical to original f column, thus our circuit is functionally equivalent.

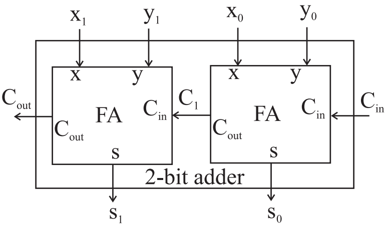
Verilog Subcircuit Example: 2-bit adder

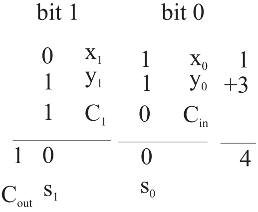
- Start with a circuit called a "full adder."
- Adds two 1 bit numbers with a carry in from previous position (will discuss later in Section 3.2 of text).



The 2-bit adder

- Will link two full adders together to create a 2-bit adder.





Full Adder Definition

- First, we define a module for our subcircuit.
- See Appendix A, Section 12 for more information on using subcircuits.
- To use a subcircuit, the subcircuit's module definition must be in the same file as the main circuit's module definition.

```
module fulladd (Cin, x, y, s, Cout);
  input Cin, x, y;
  output s, Cout;

  assign s = x ^ y ^ Cin;
  assign Cout = (x & y) | (Cin & x) | (Cin & y);

endmodule
```

Four Bit Adder Definition

- To create a module instantiation, we need to specify the module name, give a unique identifier, and give the port connections.

- Port connections can be give in either *ordered* form (listed in same order as in subcircuit) or in *named* form (explicit mapping).

```
module adder4 (carryin, X, Y, S, carryout);
  input carryin;
  input [3:0] X, Y;
  output [3:0] S;
  output carryout;
  wire [3:1] C;

  fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
  fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
  fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
  fulladd stage3 (.Cout(carryout), .s(S[3]), .y(Y[3]), .x(X[3]), .Cin(C[3]));

endmodule
```