UC Berkeley – College of Engineering, EECS Department
CS61C: Representations of Combinational Logic Circuits

Original document by J. Wawrzynek (2003-11-15)
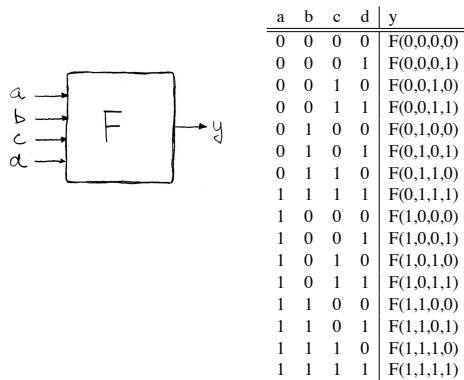Revised by Chris Sears and Dan Garcia (2004-04-26)

1   Introduction

In the previous lecture we looked at the internal details of registers. We found that every register, regardless of its use, has the same internal structure. Combinational logic (CL) blocks, on the other hand, are all different from one another. The internal circuit structure of each is tailored to the functional requirements of that particular circuit.

In this lecture we will look at three different ways to represent the function and structure of a combination logic block.

2   Truth-Tables

Combinational logic circuit behavior can be specified by enumerating the functional relationship between input values and output values. For each input pattern of 1's and 0's applied to the CL block, there exists a single output pattern. This input/output relationship is commonly enumerated in a tabular form, called a *truth-table*. In general, a truth-table takes the form shown below. In this case, for a generic block of four inputs:

a	b	c	d	y
0	0	0	0	F(0,0,0,0)
0	0	0	1	F(0,0,0,1)
0	0	1	0	F(0,0,1,0)
0	0	1	1	F(0,0,1,1)
0	1	0	0	F(0,1,0,0)
0	1	0	1	F(0,1,0,1)
0	1	1	0	F(0,1,1,0)
1	1	1	1	F(0,1,1,1)
1	0	0	0	F(1,0,0,0)
1	0	0	1	F(1,0,0,1)
1	0	1	0	F(1,0,1,0)
1	0	1	1	F(1,0,1,1)
1	1	0	0	F(1,1,0,0)
1	1	0	1	F(1,1,0,1)
1	1	1	0	F(1,1,1,0)
1	1	1	1	F(1,1,1,1)

For each row of the table, the output column shows the output value of the block for the input pattern shown in the input columns.
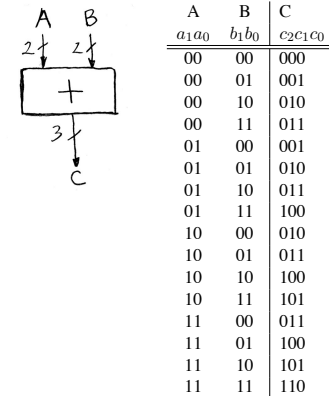
Many CL blocks have more than one output, or a single output that is more than one bit wide. In these cases, each single bit output gets its own truth-table. Often they are combined into a single table with multiple output columns, one for each single bit output.

Below are some example truth-tables:

1. Consider a CL block with two inputs, a & b, and a single output y. The output y has value 1 if one, but not both, of the inputs is a 1.

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

2. A 2-bit wide unsigned adder circuit with a 3-bit wide output:

A	B	C
$a_1 a_0$	$b_1 b_0$	$c_2 c_1 c_0$
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110

3. From the lecture notes "State Elements: Circuits That Remember", the finite state machine next state and output logic.

PS	INPUT	NS	OUTPUT
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1

4. 32-bit unsigned adder with 33-bit output:

A	B	C
000 ... 0	000 ... 0	000 ... 00
000 ... 0	000 ... 1	000 ... 01
.	.	.
.	.	.
.	.	.
111 ... 1	111 ... 1	111 ... 10

This table has 2^{64} rows! (Convince yourself by looking at the 2-bit adder on the previous page.)

In principle, the function of any combinational logic circuit can be completely specified with truth-tables; in practice some are too big.

5. Three input majority circuit. The output y takes on the value that matches *the majority* of the input values:

a	b	c	y
0	0	0	0
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	1

3 Logic Gates

What do we do if we need to determine the circuit details for a combinational logic block? In most cases we will use a collection of smaller combinational logic circuits called *logic gates*. Logic gates are simple circuits (each with only a handful of transistors) that can be wired together to implement any CL function. In CS61c we consider logic gates as primitive elements; they are the basic building blocks for our circuits.

Here are some common logic gates. For each we show its name, its graphical representation, and a truth-table that defines its function:

AND

ab	c
00	0
01	0
10	0
11	1

OR

ab	c
00	0
01	1
10	1
11	1

NOT

a	b
0	1
1	0

XOR

ab	c
00	0
01	1
10	1
11	0

NAND

ab	c
00	1
01	1
10	1
11	0

NOR

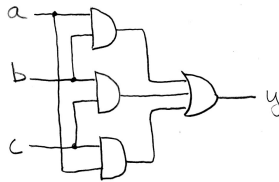| ab	c
00	1
01	0
10	0
11	0

The NOT gate is commonly called an *inverter*.

Except for NOT, we have shown 2-input versions of these gates. Versions of these gates with more than two inputs also exist. However, for performance reasons, the number of inputs to logic gates is usually restricted to around a maximum of four. The function of these gates with more than two inputs is obvious from the function of the two input version, except in the case of the the exclusive-or gate,

XOR. For more than two inputs, the XOR gate generates a 1 at its output if the number of 1's at its input is odd. Below is shown the truth-table for a three input XOR gate.

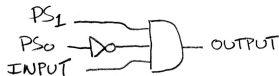| a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

These simple logic gates can be wired together to build useful circuits. In fact, any CL block can be implemented with nothing but logic gates. For instance, below is the circuit for the majority function:

If we want to understand the operation of the circuit for any particular set of input values, we can manually apply the values to the circuit, propagating the correct value to the output of each logic gate, and finally to the output. In this case, applying an input pattern of 001 would result in an output of 0, whereas an input pattern of 101 would result in a 1 at the output. We could fully characterize the function of this circuit by trying all possible input patterns to generate a truth-table.

Here is another example. It is the OUTPUT function from the finite state machine problem presented last time. The truth-table and corresponding circuit is shown below:

PS	INPUT	OUTPUT
00	0	0
00	1	0
01	0	0
01	1	1
10	0	0
10	1	1

In this case, the circuit should output a 1 if and only if the input pattern is 101. The circuit for that purpose essentially just matches the 101 pattern, when a match occurs, it outputs a 1—for all other input patterns it outputs a 0. Try it!

In general, the complete set of logic gates shown above is not needed to implement any CL function. Select subsets are sufficient. Any CL function can be implemented with nothing other than the set of AND and NOT, the set of OR and NOT, NAND gates only, and NOR gates only. However, for simplicity, a larger subset is usually employed.

One particularly nice subset is the set of AND, OR, and NOT. This set has a directly relationship to *Boolean algebra*, a mathematical system that we can use to manipulate circuits.

Above we explained how to generate a truth-table from a given circuit (we simply need to evaluate it for input combinations). The big question is: How do we go the other way? How do we derive a circuit of logic gates from a truth-table?
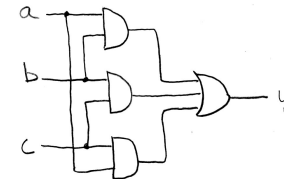
4 Boolean Algebra

In the 19th Century, the mathematician George Boole developed a mathematical system (algebra) involving logic; this system later became known as Boolean Algebra. His variables took on the values of TRUE and FALSE. Later, Claude Shannon, the father of information theory, showed for his Master's thesis how to map Boolean algebra to digital circuits.

The primitive functions of Boolean algebra are AND, OR, and NOT, just as we defined them above. The power of Boolean algebra comes from the fact that there is a one-to-one correspondence between circuits made up of AND, OR, and NOT gates and equations in Boolean algebra. For instance, the equation for the majority function

$$y = a \cdot b + a \cdot c + b \cdot c$$

has the corresponding gate circuit shown below. The equation says that the output y is the OR (we use "+" to mean OR in Boolean expressions) of three terms, where each term is the AND of two input variables (we use "·" to mean AND in Boolean expressions). As with conventional algebra, "·" has precedence over "+". Often we drop the "·" and write the equation as:
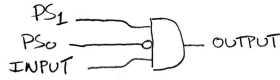
$$y = ab + ac + bc$$

The equation that corresponds to the output circuit for our FSM example is
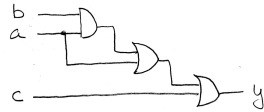
$$y = PS_1 \cdot \overline{PS_0} \cdot INPUT$$

A bar over a variable (e.g., $\overline{x}$) in Boolean algebra indicates that the variable is inverted (passed through an inverter). Also, in our circuit diagram, we often transform standalone inverters into small 'inverter bubbles' at the immediate input or output of a logic gate. We saw this with the NAND and NOR gates two pages ago. So, our original circuit for this FSM could be redrawn as:

PS₁
PS₀
INPUT → OUTPUT

The value of Boolean algebra for circuit design comes from the fact that Boolean expressions can be manipulated mathematically. For example, an expression can be simplified, which will lead directly to a circuit simplification. In general, it is much easier to manipulate equations than it is to directly manipulate circuits.
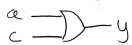
This process is illustrated below:

b
a
c → y

original circuit

↓

$y = ((ab) + a) + c$ equation derived from original circuit

↓

$= ab + a + c$
$= a(1) + c$ algebraic simplification $= a(b + 1) + c$
$= a + c$

↓

a
c → y

simplified circuit

Another use for Boolean algebra is in circuit *verification*. Given a circuit and a Boolean equation, we can ask the question "do the two different representations represent the same function." The first step would be to derive a new Boolean expression based on the circuit. Then through algebraic manipulation we could manipulate the expressions until they match, thereby "proving" the equivalence of the two.

4.1 Laws of Boolean Algebra

Along with Boolean algebra comes a collection of laws that apply to Boolean expressions. These are simple algebraic equalities that are known to be true (most of them are easy to prove). We can manipulate other Boolean expressions through successive application of these laws. Below we list the most important of the common Boolean laws:

$x \cdot \overline{x} = 0$	$x + \overline{x} = 1$	complementarity
$x \cdot 0 = 0$	$x + 1 = 1$	laws of 0's and 1's
$x \cdot 1 = x$	$x + 0 = x$	identities
$x \cdot x = x$	$x + x = x$	idempotent law
$x \cdot y = y \cdot x$	$x + y = y + x$	commutativity
$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$	associativity
$x(y + z) = xy + xz$	$x + yz = (x + y)(x + z)$	distribution
$xy + x = x$	$(x + y)x = x$	uniting theorem
$\overline{x \cdot y} = \overline{x} + \overline{y}$	$\overline{(x + y)} = \overline{x} \cdot \overline{y}$	DeMorgan's Law

Because of the symmetry of Boolean algebra all these laws come in two versions, one being called the *dual* version of the other. Practically speaking, this means you only need to remember one version of the law and the other one can be easily derived.

Given the names of these laws, we can go back and label each step of the algebraic simplification from our earlier example:

$$\begin{aligned} y \;&= ab + a + c \\ &= a(b + 1) + c \quad &\textit{distribution} \\ &= a(1) + c \quad &\textit{law of 0's and 1's} \\ &= a + c \quad &\textit{identities} \end{aligned}$$

4.2 Canonical Forms

Let's go back to the truth-table as our definition of a desired function. Starting from a truth-table, by inspection, it is possible to write a Boolean expression that matches its behavior. The type of Boolean expression that we will write is called the "sum-of-products" *canonical* (i.e., simplest) form.

This canonical form gets its name from the fact that it is formed by taking the OR of a set of terms, each term being a product (AND) of input variables or their complements. Every place (row) where a 1 appears in the output, we will use a product (AND) term in the Boolean expression. Each AND term includes all the input variables in either complemented or uncomplemented form. A variable appears in complemented form if it is a 0 in the row of the truth-table, and as a 1 if it appears as a 1 in the row.

Here is an example of a 3-input function, with the canonical sum-of-products form written to the right. The equation has four product terms, one for each of the 1's in the function y.
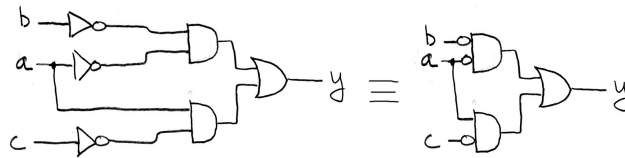
	abc	y
$\overline{a} \cdot \overline{b} \cdot \overline{c}$	000	1
$\overline{a} \cdot \overline{b} \cdot c$	001	1
	010	0
	011	0
$a \cdot \overline{b} \cdot \overline{c}$	100	1
	101	0
$a \cdot b \cdot \overline{c}$	110	1
	111	0

$$y = \overline{a}\overline{b}\overline{c} + \overline{a}\overline{b}c + a\overline{b}\overline{c} + ab\overline{c}$$

Any CL circuit that can be practically specified as a truth-table can be represented with a Boolean expression by writing its canonical form. Following that, through algebraic manipulation it can be simplified, and then translated to a circuit of logic gates.

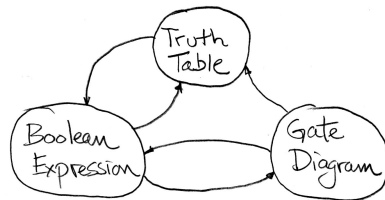Starting with the example sum-of-products canonical form above, we can simplify the expression in a few steps:

$$
\begin{aligned}
y &= \overline{a}\overline{b}\overline{c} + \overline{a}\overline{b}c + a\overline{b}\overline{c} + ab\overline{c} \\
&= \overline{a}\overline{b}(\overline{c}+c) + a\overline{c}(\overline{b}+b) \quad &\text{distribution} \\
&= \overline{a}\overline{b}(1) + a\overline{c}(1) \quad &\text{complementarity} \\
&= \overline{a}\overline{b} + a\overline{c} \quad &\text{identity}
\end{aligned}
$$

Next we can draw the circuit diagram, shown here drawn both ways:

5 Summary

We can summarize the ideas of this lecture with the following diagram:

It shows the three possible representation for a CL block. The truth-table form is unique—there is only one truth-table for every possible CL block. However, there are multiple Boolean expressions possible and thus multiple gate level representations for any CL block (the sum-of-products canonical form is unique). The uniqueness of the truth-table makes it a useful way to clearly define the function of a CL block. Boolean expressions are useful for algebraic manipulation, particularly simplification. The gate diagram gives us a prescription for converting to an actual transistor circuit; each gate in the diagram can be replaced by a small transistor circuit that achieves its respective gate level function.

We discussed how to convert among these three representations, as represented by the arcs in the diagram:

Truth-table to Boolean Expression. Write the canonical form and follow with algebraic simplification if desired.

Boolean Expression to Truth-table. Evaluate the expression for all input combinations and record output values.

Boolean Expression to Gates. Use AND gates for the AND operators, OR gates for the OR operators, and inverters for the NOT operator. Wire up the gates to match the structure of the expression.

Gates to Boolean Expression. Reverse the above process.

Gates to Truth-table. Pass through all input combinations and evaluate output.

Truth-table to Gates. Map to Boolean expression then to gates.