7.3 Combinational logic

(Original section[1])

In this section, we look at a couple of larger logic building blocks that we use heavily, and we discuss the design of structured logic that can be automatically implemented from a logic equation or truth table by a translation program. Last, we discuss the notion of an array of logic blocks.

Decoders

One logic block that we will use in building larger components is a *decoder*. The most common type of decoder has an *n*-bit input and 2^n outputs, where only one output is asserted for each input combination. This decoder translates the *n*-bit input into a signal that corresponds to the binary value of the *n*-bit input. The outputs are thus usually numbered, say, Out0, Out1, ... , Out2^n - 1. If the value of the input is *i*, then Out*i* will be true and all other outputs will be false. The figure below shows a 3-bit decoder and the truth table. This decoder is called a *3-to-8 decoder* since there are three inputs and eight (2^3) outputs. There is also a logic element called an ***encoder*** that performs the inverse function of a decoder, taking 2^n inputs and producing an *n*-bit output.

Decoder: A logic block that has an *n*-bit input and 2^n outputs, where only one output is asserted for each input combination.

Figure 7.3.1: A 3-bit decoder has 3 inputs, called 12, 11, and 10, and 2^3 = 8 outputs, called Out0 to Out7 (COD Figure A.3.1).

Only the output corresponding to the binary value of the input is true, as shown in the truth table. The label 3 on the input to the decoder says that the input signal is 3 bits wide.

Inputs	Outputs

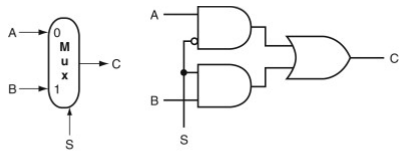
Multiplexors

One basic logic function that we use quite often in COD Chapter 4 (The Processor) is the multiplexor. A ***multiplexor*** might more properly be called a ***selector***, since its output is one of the inputs that is selected by a control. Consider the two-input multiplexor. The left side of the figure below shows this multiplexor has three inputs: two data values and a *selector* (or *control*) *value*. The selector value determines which of the inputs becomes the output. We can represent the logic function computed by a two-input multiplexor, shown in gate form on the right side of the figure below, as $C = (A \cdot \overline{S}) + (B \cdot S)$.

Selector value: Also called ***control value***. The control signal that is used to select on of the input values of a multiplexor as the output of the multiplexor.

Figure 7.3.2: A two-input multiplexor on the left and its implementation with gates on the right (COD Figure A.3.2).

The multiplexor has two data inputs (*A* and *B*), which are labeled *0* and *1*, and one selector input (*s*), as well as an output *C*. Implementing multiplexors in Verilog requires a little more work, especially when they are wider than two inputs. We show how to do this in COD Section A.4 (Using a hardware description language).



Multiplexors can be created with an arbitrary number of data inputs. When there are only two inputs, the selector is a single signal that selects one of the inputs if it is true (1) and the other if it is false (0). If there are *n* data inputs, there will need to be $\lceil log_2 n \rceil$ selector inputs. In this case, the multiplexor basically consists of three parts:

1. A decoder that generates *n* signals, each indicating a different input value
2. An array of *n* AND gates, each combining one of the inputs with a signal from the decoder
3. A single large OR gate that incorporates the outputs of the AND gates

To associate the inputs with selector values, we often label the data inputs numerically (i.e., 0, 1, 2, 3, ..., *n* - 1) and interpret the data selector inputs as a binary number. Sometimes, we make use of a multiplexor with undecoded selector signals.

Multiplexors are easily represented combinationally in Verilog by using *if* expressions. For larger multiplexors, *case* statements are more convenient, but care must be taken to synthesize combinational logic.

Two-level logic and PLAs

As pointed out in the previous section, any logic function can be implemented with only AND, OR, and NOT functions. In fact, a much stronger result is true. Any logic function can be written in a canonical form, where every input is either a true or complemented variable and there are only two levels of gates—one being AND and the other OR—with a possible inversion on the final output. Such a representation is called a two-level representation, and there are two forms, called *sum of products* and product of sums. A sum-of-products representation is a logical sum (OR) of products (terms using the AND operator); a product of sums is just the opposite. In our earlier example, we had two equations for the output *E*:

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot (\overline{A \cdot B \cdot C})$$

and

$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) \cdot (B \cdot C \cdot \overline{A})$$

This second equation is in a sum-of-products form: it has two levels of logic and the only inversions are on individual variables. The first equation has three levels of logic.

Sum of products: A form of logical representation that employs a logical sum (OR) of products (terms joined using the AND operator).

Elaboration

We can also write E as a product of sums:

$$E = \overline{(\overline{A} + \overline{B} + C) \cdot (\overline{A} + \overline{C} + B) \cdot (\overline{B} + C + A)}$$

To derive this form, you need to use DeMorgan's theorems, which are discussed in the exercises.

In this text, we use the sum-of-products form. It is easy to see that any logic function can be represented as a sum of products by constructing such a representation from the truth table for the function. Each truth table entry for which the function is true corresponds to a product term. The product term consists of a logical product of all the inputs or the complements of the inputs, depending on whether the entry in the truth table has a 0 or 1 corresponding to this variable. The logic function is the logical sum of the product terms where the function is true. This is more easily seen with an example.

Example 7.3.1: Sum of products.

Show the sum-of-products representation for the following truth table for *D*.

Inputs			Output
A	B	C	D
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

Answer

There are four product terms, since the function is true (1) for four different input combinations. These

$$\overline{A} \cdot \overline{B} \cdot C$$
$$\overline{A} \cdot B \cdot \overline{C}$$
$$A \cdot \overline{B} \cdot \overline{C}$$
$$A \cdot B \cdot C$$

Thus, we can write the function for *D* as the sum of these terms:

$$D = (\overline{A} \cdot \overline{B} \cdot C) + (\overline{A} \cdot B \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot B \cdot C$$

Note that only those truth entries for which the function is true generate terms in the equation.

We can use this relationship between a truth table and a two-level representation to generate a gate-level implementation of any set of logic functions. A set of logic functions corresponds to a truth table with multiple output columns. Each output column represents a different logic function, which may be directly constructed from the truth table.

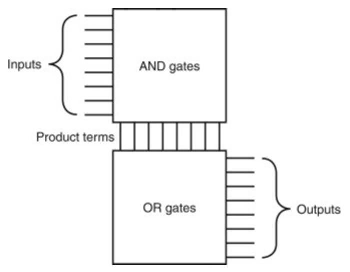
The sum-of-products representation corresponds to a common structured-logic implementation called a *programmable logic array (PLA)*. A PLA has a set of inputs and corresponding input complements (which can be implemented with a set of inverters), and two stages of logic. The first stage is an array of AND gates that form a set of *product terms* (sometimes called *minterms*); each product term can consist of any of the inputs or their complements. The second stage is an array of OR gates, each of which forms a logical sum of any number of the product terms. The figure below shows the basic form of a PLA.

Programmable logic array (***PLA***): A structured-logic element composed of a set of inputs and corresponding input complements and two stages of logic: the first generates product terms of the inputs and input complements, and the second generates sum terms of the product terms. Hence, PLAs implement logic functions as a sum of products.

Minterms: Also called ***product terms***. A set of logic inputs joined by conjunction (AND operations); the product terms form the first logic stage of the *programmable logic array* (PLA).

Figure 7.3.3: The basic form of a PLA consists of an array of AND gates followed by an array of OR gates (COD Figure A.3.3).

Each entry in the AND gate array is a product term consisting of any number of inputs or inverted inputs. Each entry in the OR gate array is a sum term consisting of any number of these product terms.



A PLA can directly implement the truth table of a set of logic functions with multiple inputs and outputs. Since each entry where the output is true requires a product term, there will be a corresponding row in the PLA. Each output corresponds to a potential row of OR gates in the second stage. The number of OR gates corresponds to the number of truth table entries for which the output is true. The total size of a PLA, such as that shown in the figure above, is equal to the sum of the size of the AND gate array (called the *AND plane*) and the size of the OR gate array (called the *OR plane*). Looking at the figure above, we can see that the size of the AND gate array is equal to the number of inputs times the number of diffe41rent product terms, and the size of the OR gate array is the number of outputs times the number of product terms.

A PLA has two characteristics that help make it an efficient way to implement a set of logic functions. First, only the truth table entries that produce a true value for at least one output have any logic gates associated with them. Second, each different product term will have only one entry in the PLA, even if the product term is used in multiple outputs. Let's look at an example.

Example 7.3.2: PLAs.

Consider the set of logic functions defined in COD Sections A.2 (Gates, truth tables, and logic equations). Show a PLA implementation of this example for *D*, *E*, and *F*.

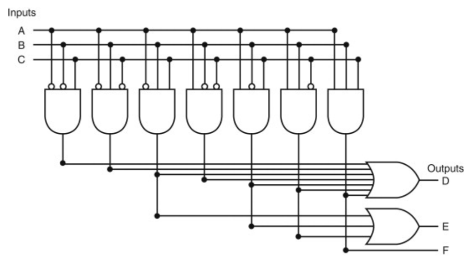
Answer

Here is the truth table we constructed earlier:

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Since there are seven unique product terms with at least one true value in the output section, there will be seven columns in the AND plane. The number of rows in the AND plane is three (since there are three inputs), and there are also three rows in the OR plane (since there are three outputs). The figure below shows the resulting PLA, with the product terms corresponding to the truth table entries from top to bottom.

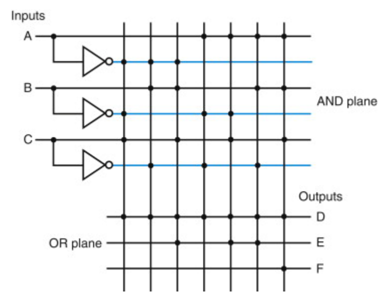
Figure 7.3.4: The PLA for implementing the logic function described in the example above (COD Figure A.3.4).



Rather than drawing all the gates, as we do in the figure above, designers often show just the position of AND gates and OR gates. Dots are used on the intersection of a product term signal line and an input line or an output line when a corresponding AND gate or OR gate is required. The figure below shows how the PLA of the figure above would look when drawn in this way. The contents of a PLA are fixed when the PLA is created, although there are also forms of PLA-like structures, called *PALs*, that can be programmed electronically when a designer is ready to use them.

Figure 7.3.5: The PLA drawn using dots to indicate the components of the product terms and sum terms in the array (COD Figure A.3.5).

Rather than use inverts on the gates, usually all the inputs run the width of the AND plane in both true and complement forms. A dot in the AND plane indicates that the input, or its inverse, occurs in the product term. A dot in the OR plane indicates that the corresponding product term appears in the corresponding output.



ROMs

Another form of structured logic that can be used to implement a set of logic functions is a *read-only memory* (*ROM*). A ROM is called a memory because it has a set of locations that can be read; however, the contents of these locations are fixed, usually at the time the ROM is manufactured. There are also *programmable ROMs* (*PROMs*) that can be programmed electronically, when a designer knows their contents. There are also erasable PROMs; these devices require a slow erasure process using ultraviolet light, and thus are used as read-only memories, except during the design and debugging process.

Read-only memory (***ROM***): A memory whose contents are designated at creation time, after which the contents can only be read. ROM is used as structured logic to implement a set of logic functions by using the terms in the logic functions as address inputs and the outputs as bits in each word of the memory.

Programmable ROM (***PROM***): A form of read-only memory that can be programmed when a designer knows its contents.

A ROM has a set of input address lines and a set of outputs. The number of addressable entries in the ROM determines the number of address lines: if the ROM contains 2^m addressable entries, called the *height*, then there are *m* input lines. The number of bits in each addressable entry is equal to the number of output bits and is sometimes called the *width* of the ROM. The total number of bits in the ROM is equal to the height times the width. The height and width are sometimes collectively referred to as the *shape* of the ROM.

A ROM can encode a collection of logic functions directly from the truth table. For example, if there are *n* functions with *m* inputs, we need a ROM with *m* address lines (and 2^m entries), with each entry being *n* bits wide. The entries in the input portion of the truth table represent the addresses of the entries in the ROM, while the contents of the output portion of the truth table constitute the contents of the ROM. If the truth table is organized so that the sequence of entries in the input portion constitutes a sequence of binary numbers (as have all the truth tables we have shown so far), then the output portion gives the ROM contents in order as well. In the example above (PLAs), there were three inputs and three outputs. This leads to a ROM with $2^3 = 8$ entries, each 3 bits wide. The contents of those entries in increasing order by address are directly given by the output portion of the truth table.

ROMs and PLAs are closely related. A ROM is fully decoded: it contains a full output word for every possible input combination. A PLA is only partially decoded. This means that a ROM will always contain more entries. For the earlier truth table the example above (PLAs), the ROM contains entries for all eight possible inputs, whereas the PLA contains only the seven active product terms. As the number of inputs grows, the number of entries in the ROM grows exponentially. In contrast, for most real logic functions, the number of product terms grows much more slowly (see the examples in COD Appendix C (Mapping control to hardware)). This difference makes PLAs generally more efficient for implementing combinational logic functions. ROMs have the advantage of being able to implement any logic function with the matching number of inputs and outputs. This advantage makes it easier to change the ROM contents if the logic function changes, since the size of the ROM need not change.

In addition to ROMs and PLAs, modern logic synthesis systems will also translate small blocks of combinational logic into a collection of gates that can be placed and wired automatically. Although some small collections of gates are usually not area-efficient, for small logic functions they have less overhead than the rigid structure of a ROM and PLA and so are preferred.

For designing logic outside of a custom or semicustom integrated circuit, a common choice is a field programming device we describe these devices in COD Sections A.12 (Field programmable devices).

Don't cares

Often in implementing some combinational logic, there are situations where we do not care what the value of some output is, either because another output is true or because a subset of the input combinations determines the values of the outputs. Such situations are referred to as *don't cares*. Don't cares are important because they make it easier to optimize the implementation of a logic function.

There are two types of don't cares: output don't cares and input don't cares, both of which can be represented in a truth table. *Output don't cares* arise when we don't care about the value of an output for some input combination. They appear as Xs in the output portion of a truth table. When an output is a don't care for some input combination, the designer or logic optimization program is free to make the output true or false for that input combination. *Input don't cares* arise when an output depends on only some of the inputs, and they are also shown as Xs, though in the input portion of the truth table.

Example 7.3.3: Don't cares.

Consider a logic function with inputs *A*, *B*, and *C* defined as follows:

- If *A* or *C* is true, then output *D* is true, whatever the value of *B*.
- If *A* or *B* is true, then output *E* is true, whatever the value of *C*.
- Output *F* is true if exactly one of the inputs is true, although we don't care about the value of *F*, whenever *D* and *E* are both true.

Show the full truth table for this function and the truth table using don't cares. How many product terms are required in a PLA for each of these?

Answer

Here's the full truth table, without don't cares:

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0

This requires seven product terms without optimization. The truth table written with output don't cares looks like this:

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	X
1	0	0	1	1	X
1	0	1	1	1	X
1	1	0	1	1	X
1	1	1	1	1	X

If we also use the input don't cares, this truth table can be further simplified to yield the following:

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
X	1	1	1	1	X
1	X	X	1	1	X

This simplified truth table requires a PLA with four minterms, or it can be implemented in discrete gates with one two-input AND gate and three OR gates (two with three inputs and one with two inputs). This compares to the original truth table that had seven minterms and would have required four AND gates.

Logic minimization is critical to achieving efficient implementations. One tool useful for hand minimization of random logic is *Karnaugh maps*. Karnaugh maps represent the truth table graphically, so that product terms that may be combined are easily seen. Nevertheless, hand optimization of significant logic functions using Karnaugh maps is impractical, both because of the size of the maps and their complexity. Fortunately, the process of logic minimization is highly mechanical and can be performed by design tools. In the process of minimization, the tools take advantage of the don't cares, so specifying them is important. The textbook references at the end of this appendix provide further discussion on logic minimization, Karnaugh maps, and the theory behind such minimization algorithms.

Arrays of logic elements

Many of the combinational operations to be performed on data have to be done to an entire word (64 bits) of data. Thus we often want to build an array of logic elements, which we can represent simply by showing that a given operation will happen to an entire collection of inputs. Inside a machine, much of the time we want to select between a pair of buses. A *bus* is a collection of data lines that is treated together as a single logical signal. (The term bus is also used to indicate a shared collection of lines with multiple sources and uses.)

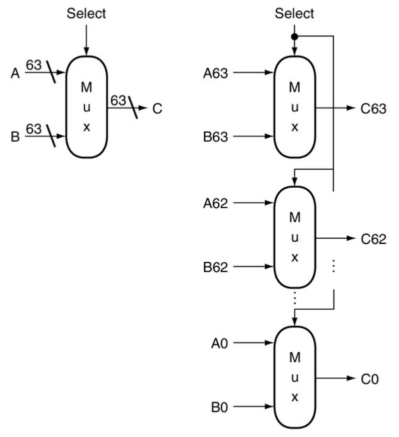
Bus: In logic design, a collection of data lines that is treated together as a single logical signal; also, a shared collection of lines with multiple sources and uses.

For example, in the LEGv8 instruction set, the result of an instruction that is written into a register can come from one of two sources. A multiplexor is used to choose which of the two buses (each 64 bits wide) will be written into the Result register. The 1-bit multiplexor, which we showed earlier, will need to be replicated 64 times.

We indicate that a signal is a bus rather than a single 1-bit line by showing it with a thicker line in a figure. Most buses are 64 bits wide; those that are not are explicitly labeled with their width. When we show a logic unit whose inputs and outputs are buses, this means that the unit must be replicated a sufficient number of times to accommodate the width of the input. The figure below shows how we draw a multiplexor that selects between a pair of 64-bit buses and how this expands in terms of 1-bit-wide multiplexors. Sometimes we need to construct an array of logic elements where the inputs for some elements in the array are outputs from earlier elements. For example, this is how a multibit-wide ALU is constructed. In such cases, we must explicitly show how to create wider arrays, since the individual elements of the array are no longer independent, as they are in the case of a 64-bit-wide multiplexor.

Figure 7.3.6: A multiplexor is arrayed 64 times to perform a selection between two 64-bit inputs (COD Figure A.3.6).

Note that there is still only one data selection signal used for all 64 1-bit multiplexors.



a. A 64-bit wide 2-to-1 multiplexor

b. The 64-bit wide multiplexor is actuall
an array of 64 1-bit multiplexors

PARTICIPATION ACTIVITY 7.3.1: Check yourself: Parity.

Parity is a function in which the output depends on the number of 1s in the input. For an even parity function, the output is 1 if the input has an even number of ones. Suppose a ROM is used to implement an even parity function with a 4-bit input.

Address	A	B	C	D
0	0	1	1	0
1	0	1	0	1
2	0	1	0	1
3	0	1	1	0
4	0	1	0	1
5	0	1	1	0
6	0	1	1	0
7	0	1	0	1
8	1	0	0	1
9	1	0	1	0
10	1	0	1	0
11	1	0	0	1
12	1	0	1	0
13	1	0	0	1
14	1	0	0	1
15	1	0	1	0

1) Which of A, B, C, or D represents the contents of the ROM?

- ○ ROM A
- ○ ROM B
- ○ ROM C
- ○ ROM D

(*1) This section is in original form.

Provide feedback on this section