

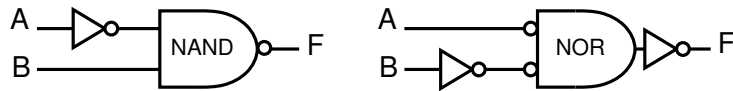
Mixed Logic

Introduction

Mixed logic is a gate-level design methodology used in industry. It allows a digital logic circuit designer to separate the *functional description* of the circuit from its *physical implementation*. For example, consider the function:

$$F = \overline{\overline{A} \cdot B}$$

This is a *functional description*. Two possible *physical implementations* are listed below, one using a NAND gate and inverters, the other using a NOR gate and inverters (note that we represent the NOR using the DeMorgan equivalent):

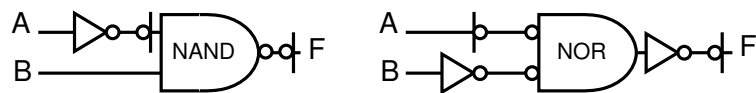


Both of the above circuits implement the same function, but are different physical implementations. Which is easier to read? Ideally, the implementation of the circuit should not affect your ability to figure out what the circuit does. The goal of mixed logic design is to:

1. Separate *what* the circuit does from *how* it does it ; and
2. support self-documenting circuits.

Analysis

Before getting into the details of how to design a mixed-logic circuit, let us modify the circuit of the preceding example to see how mixed-logic notation works. Using mixed-logic notation, the above circuits are now illustrated as:



Note the vertical bars in both circuits. They *do not represent physical circuit elements* (e.g. logical inverters) – they are simply a form of notation. To read a circuit using mixed logic notation:

1. Ignore all bubbles on logic gates and inverters. This means
 - (a) Read all gates in terms of the underlying gate body that is drawn. For example, in both cases above, read the NAND and NOR as AND gates ; and
 - (b) ignore all physical inverters.
2. Wherever you see a vertical bar with a bubble, take the complement.

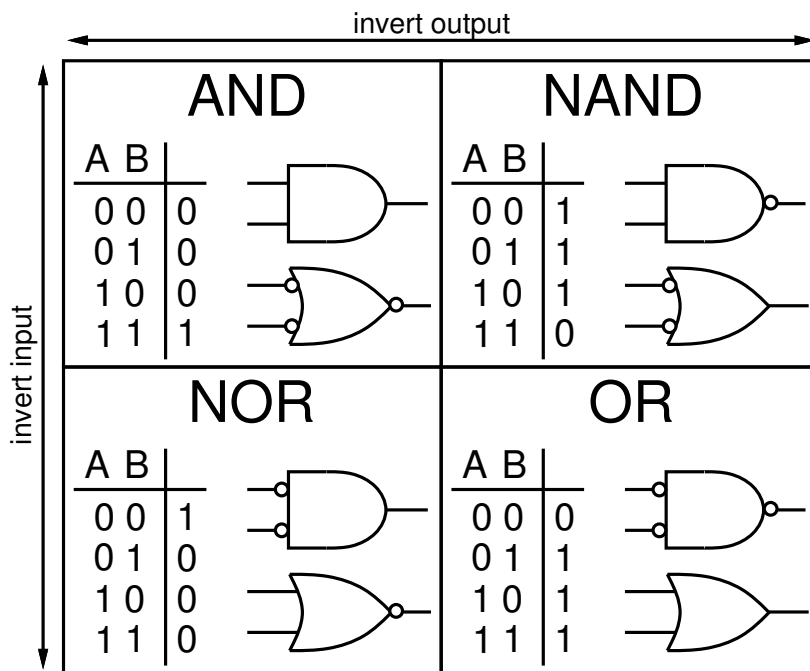
Thus when reading the circuit left to right, a bar should exist everywhere that a complement exists in the corresponding logic equation. By not worrying about every physical inversion, it is much easier to read the function implemented by the circuit.

Many students make the mistake of interpreting the vertical bars as physical inverters. The vertical bars are just a form of notation to indicate where complements occur in the underlying function.

In both examples above, A is complemented yielding \bar{A} . Ignoring all physical inverters and treating both logic gates as an AND, the output of the gate is $\bar{A}B$. Inverting the output by the vertical bar after the logic gates results in $\overline{\bar{A}B}$.

Design

Mixed logic design is based on the key observation of DeMorgan's theorem: logical operations have equivalencies when their inputs and outputs are inverted. DeMorgan's square, shown below, illustrates the equivalencies of the four basic gate types.



Inverting the output of the gate moves horizontally in the square. Moving vertically is accomplished by inverting all gate inputs (turning the truth table upside down).

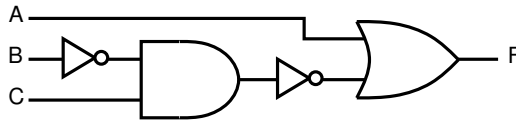
Each of the four fundamental gate types has both an AND-based and OR-based functional equivalency, based on DeMorgan's theorem. You have already seen this for NAND and NOR gates, but it applies to AND and OR gates as well. While it may seem counterintuitive to draw an AND gate as an OR body with inverted inputs and outputs, this variation makes mixed logic design possible.

Design Example 1

Design a logic circuit for the function

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

If implemented in a straightforward manner using traditional gate symbols, its implementation would be:



What if we wanted to implement it using just NAND gates? Or only NAND and NOR gates? Or any other constraint on what logic gates are used for the physical implementation? The basic design rules for implementing a mixed logic design are:

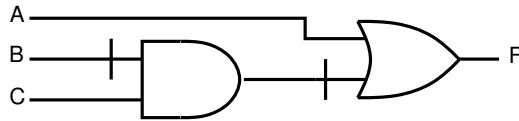
1. All logic operations in the function (sum and product) become gate bodies in the circuit. Draw the circuit using just AND and OR gates where sum and product terms occur.
2. Draw vertical bars in the circuit where all complements in the logical function occur.
3. Now depending on your design constraints, substitute for the AND and OR gates whatever gates you wish to use (NAND/NOR/AND/OR) using the appropriate equivalent body (AND/OR, as shown in DeMorgan's square). In other words,
 - (a) implement all OR operations in the logical function using the OR equivalency of the logic gate of your choice ; and
 - (b) implement all AND operations in the logical function using the AND equivalency of the logic gate of your choice.
4. Draw bubbles on all the vertical bars.
5. All bubbles in the circuit should be *paired* so that they cancel out. A bubble may be paired with:
 - (a) another bubble on a logic gate ; or
 - (b) a bubble on a vertical bar.

The vertical bars with bubbles *do not* represent physical devices (like physical inverters). They are just a form of notation to represent a logical complement in the underlying function. *Anywhere a pairing of a bubble is not possible, place a physical inverter.*

Now your circuit implementation is complete.

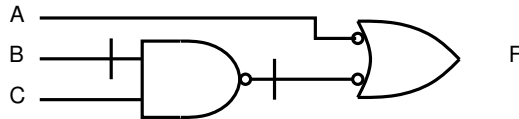
To illustrate mixed logic design, we will implement the function above four different ways: using NAND gates and inverters, NOR gates and inverters, AND gates and inverters, and OR gates and inverters.

First, draw the circuit graphically (this is not a physical implementation), using AND/OR gates for the operations and the vertical bars for the complement. As a reminder, $F = A + (\overline{\overline{B} \cdot C})$.

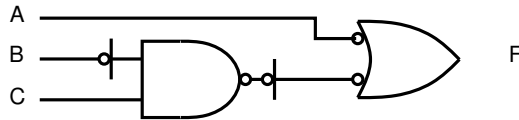


Now using the above as a starting point, design it using just NAND gates and inverters.

1. Implement the AND and OR operations of the circuit using the corresponding equivalencies of the NAND gate.

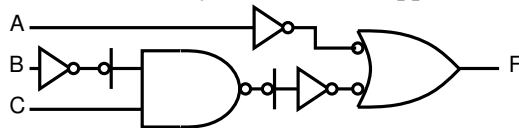


2. Draw a bubble on each inversion bar.



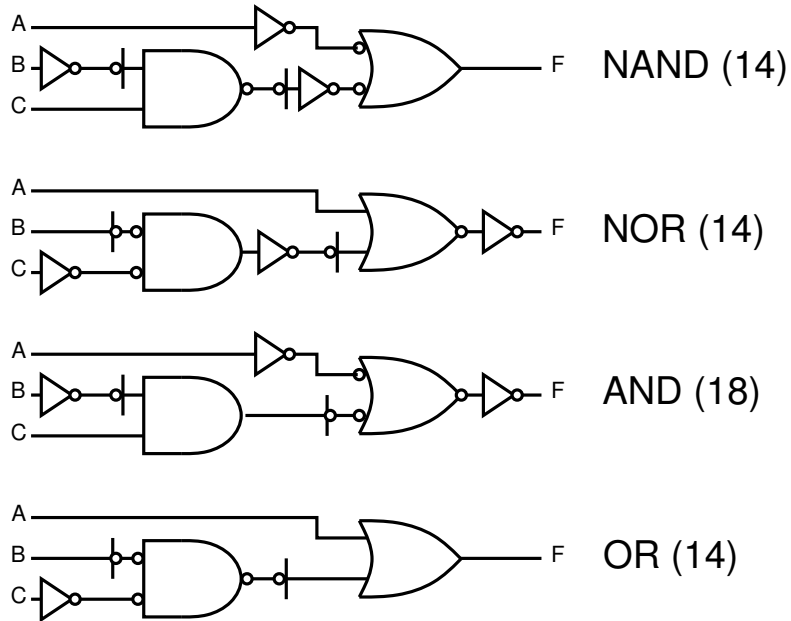
3. All bubbles in the circuit should be *paired* so that they cancel out. A bubble may be paired with:
 - (a) another bubble on a logic gate ; or
 - (b) a bubble on a vertical bar.

Insert physical inverters where necessary to make this happen.



You're done! You've now implemented the circuit using 2 NAND gates and 3 inverters, which requires 14 transistors (4 each for the NAND gates, 2 each for the inverters). The circuit is also self-documenting, in that by ignoring the inverters and bubbles on the gates and just paying attention to the vertical bars and the underlying gate body, you can easily read off the function being implemented.

The figure below shows 4 different circuit implementations of this function, each using a different type of gate (NAND, NOR, AND, OR). All circuits implement the same function. The first is the example that you just saw. Also shown with each circuit is the transistor count for that particular implementation.



Note that three of the four circuits have similar transistor counts, while the AND-based implementation requires more. Why would you choose a particular implementation? There are three general reasons.

1. To reduce transistor count.
2. Component reuse. Prior to the development of programmable logic, digital circuit boards were implemented entirely with chips (integrated circuits, or ICs) that would have a fixed number of gates on them. For example, an IC called a **7400** is a “quad 2-input NAND gate.” It has 4 NAND gates on it that can be wired up to other ICs. If the designer had one spare NAND gate unallocated on an IC, she may choose to fit it into a circuit design rather than add another chip to the circuit board. While this is not used as much in practice today, many “real-world” design projects today often require re-designing or reverse engineering older circuit implementations that have used such techniques.
3. More component reuse. Today, technologies such as programmable devices, field-programmable gates arrays (FPGAs), and very-large scale integrated circuits (VLSI), allow very high density logic programming on individual chips. At this level of density, logic design is typically done using software development tools, and designers typically have libraries of parts pre-designed for particular sub-circuits or functional implementations.

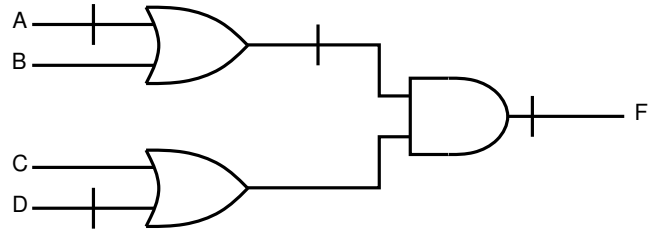
Design Example 2

Consider implementing the function

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

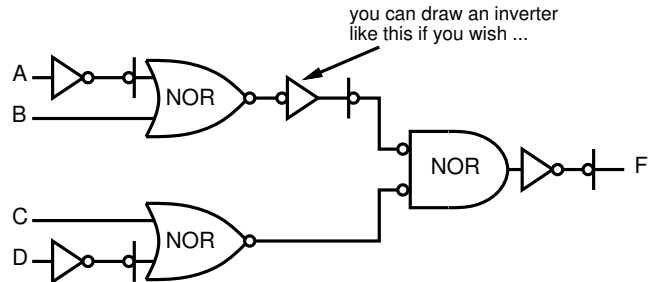
using just NOR gates.

Again, the form of the functional expression is independent of implementation. First represent the function graphically:

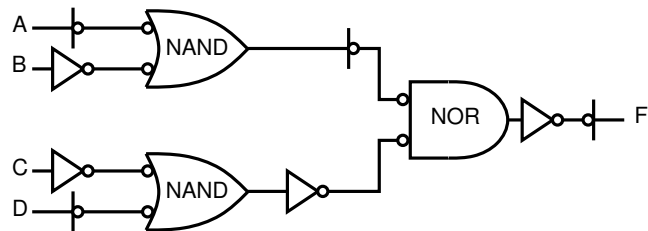


While not an implementation, this circuit shows the logical combination of the inputs that yields the appropriate output. The bars represent where an inversion of the signal is required, and correspond to the complementing bars in the actual function. This basic structure remains unchanged regardless of the gate types used to implement it.

To realize an implementation using NOR gates, again follow the mixed-logic design procedure. First, replace all the gates with the AND/OR equivalency for the NOR gate. Second, place vertical bars with bubbles where each complement should occur. Third, add inverters where necessary so that all bubbles are paired up. Your final implementation should look like:

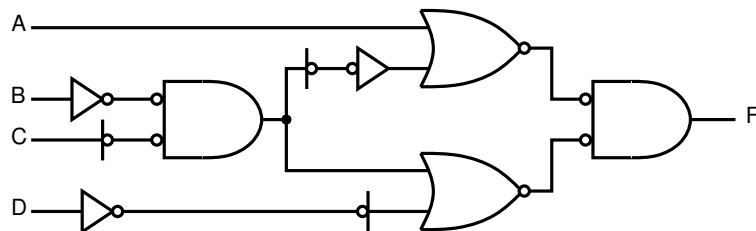


There is no reason why we are constrained to using one particular type of gate. The circuit below implements the same function using NAND gates for the OR terms and NOR gates for the AND terms.



Design Example 3

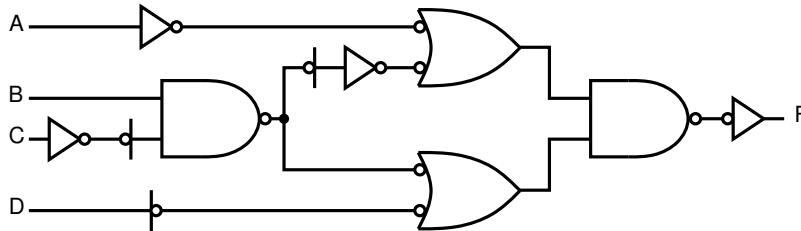
Here's another example, showing a circuit implemented with just NOR gates using mixed-logic notation.



Thanks to mixed-logic notation, it is relatively easy to extract the implemented function. Reading the circuit from left to right, ignoring all bubbles, and complementing where vertical bars occur:

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

The fact that this implementation includes only NOR gates neither changes nor distorts the behavior of the circuit. Not the common subexpression $((B \cdot \overline{C}))$ used in two places in the circuit. When bubbles are added to an output wire that is used in multiple places (*fanout* > 1), multiple bubbles on the “consumer-end” must be added to match the bubble on the “producer-end.” This can be seen when this circuit is re-implemented using just NAND gates.



Summary

In summary, mixed logic design:

1. Decouples the functional behavior of a circuit from its physical implementation ;
2. supports self-documenting circuits ;
3. provides a simple process to re-implement a circuit using different gate types.