Here's a final logic implementation strategy using read-only memories.

This strategy is useful when you need to generate many different outputs from the same set of inputs, a situation we'll see a lot when we get to finite state machines later on in the course.

Where MUXes are good for implementing truth tables with one output column, read-only memories are good for implementing truth tables with many output columns.

One of the key components in a read-only memory is the decoder which has K select inputs and 2^K data outputs.

Only one of the data outputs will be 1 (or HIGH) at any given time, which one is determined by the value on the select inputs.

The Jth output will be 1 when the select lines are set to the binary representation of J.

Here's a read-only memory implementation for the 2-output truth table shown on the left.

This particular 2-output device is a full adder, which is used as a building block in addition circuits.

The three inputs to the function (A, B, and CI) are connected to the select lines of a 3-to-8 decoder.

The 8 outputs of the decoder run horizontally in the schematic diagram and each is labeled with the input values for which that output will be HIGH.

So when the inputs are 000, the top decoder output will be HIGH and all the other decoder outputs LOW.

When the inputs are 001 - i.e., when A and B are 0 and CI is 1 - the second decoder output will be HIGH.

And so on.

The decoder outputs control a matrix of NFET pulldown switches.

The matrix has one vertical column for each output of the truth table.

Each switch connects a particular vertical column to ground, forcing it to a LOW value when the switch is on.

The column circuitry is designed so that if no pulldown switches force its value to 0, its value will be a 1.

The value on each of the vertical columns is inverted to produce the final output values.

So how do we use all this circuitry to implement the function described by the truth table?

For any particular combination of input values, exactly one of the decoder outputs will be HIGH, all the others will be low.

Think of the decoder outputs as indicating which row of the truth table has been selected by the input values.

All of the pulldown switches controlled by the HIGH decoder output will be turned ON, forcing the vertical column to which they connect LOW.

For example, if the inputs are 001, the decoder output labeled 001 will be HIGH.

This will turn on the circled pulldown switch, forcing the S vertical column LOW.

The COUT vertical column is not pulled down, so it will be HIGH.

After the output inverters, S will be 1 and COUT will be 0, the desired output values.

By changing the locations of the pulldown switches, this read-only memory can be programmed to implement any 3-input, 2-output function.

For read-only memories with many inputs, the decoders have many outputs and the vertical columns in the switch matrix can become quite long and slow.

We can reconfigure the circuit slightly so that some of the inputs control the decoder and the other inputs are used to select among multiple shorter and faster vertical columns.

This combination of smaller decoders and output MUXes is quite common in these sorts of memory circuits.

Read-only memories, ROMs for short, are an implementation strategy that ignores the structure of the particular boolean expression to be implemented.

The ROM's size and overall layout are determined only by the number of inputs and outputs.

Typically the switch matrix is fully populated, with all possible switch locations filled with an NFET pulldown.

A separate physical or electrical programming operation determines which switches are actually controlled by the decoder lines.

The other switches are configured to be in the permanently off state.

If the ROM has N inputs and M outputs, then the switch matrix will have 2^N rows and M output columns, corresponding exactly to the size of the truth table.

As the inputs to the ROM change, various decoder outputs will turn off and on, but at slightly different times.

As the decoder lines cycle, the output values may change several times until the final configuration of the pulldown switches is stable.

So ROMs are not lenient and the outputs may show the glitchy behavior discussed earlier.

Whew!

This has been a whirlwind tour of various circuits we can use to implement logic functions.

The sum-of-products approach lends itself nicely to implementation with inverting logic.

Each circuit is custom-designed to implement a particular function and as such can be made both fast and small.

The design and manufacturing expense of creating such circuits is worthwhile when you need high-end performance or are producing millions of devices.

MUX and ROM circuit implementations are mostly independent of the specific function to be implemented.

That's determined by a separate programming step, which may be completed after the manufacture of the devices.

They are particularly suited for prototyping, low-volume production, or devices where the functionality may need to be updated after the device is out in the field.