CHAPTER 6

Optimizations and Tradeoffs

6.1 INTRODUCTION

The previous chapters described how to design digital circuits using straightforward techniques. This chapter will describe how to design *better* circuits. For our purposes, *better* means circuits that are smaller, faster, or consume less power. Real-world design may involve additional criteria.

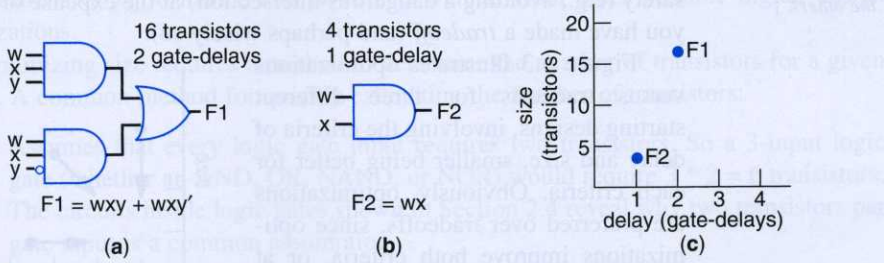


Figure 6.1 A circuit transformation that improves both size and delay, called an *optimization*: (a) original circuit, (b) optimized circuit, (c) plot of size and delay of each circuit.

Consider the circuit for the equation involving `F1` shown in Figure 6.1(a). The circuit's size, assuming two transistors per gate input and ignoring inverters for simplicity, is 8 * 2 = 16 transistors. The circuit's delay, which is the longest path from any input to the output, is 2 gate-delays. We could algebraically transform the equation into that for `F2` shown in Figure 6.1(b). `F2` represents the same function as `F1`, but requires only 4 transistors instead of 16, and has a delay of only 1 gate-delay instead of 2. The transformation improved both size and delay, as shown in Figure 6.1(c). A transformation that improves all criteria of interest is called an ***optimization***.

Now consider the circuit for a different function in Figure 6.2(a), implementing the equation for `G1`. The circuit's size (assuming 2 transistors per gate input) is 14 transistors, and the circuit's delay is 2 gate-delays. We could algebraically transform the equation into that shown for `G2` in Figure 6.2(b), which results in a circuit having only 12 transistors. However, the reduction in transistors comes at the expense of a longer delay of 3 gate-delays, as shown in Figure 6.2(c). Which circuit is better, the circuit for `G1` or for

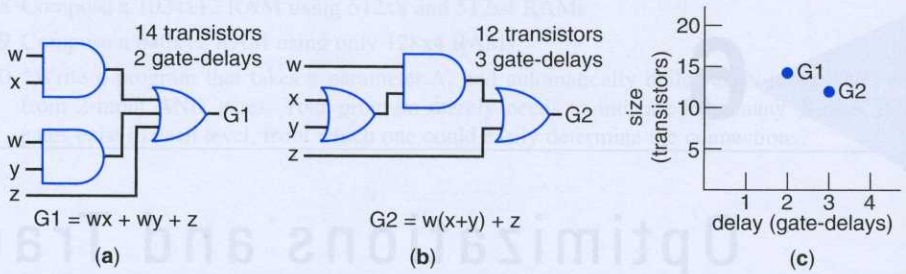


Figure 6.2 A circuit transformation that improves size but worsens delay, called a *tradeoff:* (a) original circuit, (b) transformed circuit, (c) plot of size and delay of each circuit.

*A **tradeoff** improves some criteria at the expense of other criteria of interest. An **optimization** improves all criteria of interest, or improves some of those criteria without worsening the others.*

G2? The answer depends on whether the size or delay criteria is more important. A transformation that improves one criteria at the expense of another criteria is called a ***tradeoff***.

You likely perform optimizations and tradeoffs every day. Perhaps you regularly commute by car from one city to another via a particular route. You might be interested in two criteria: commute time and safety. Other criteria, such as scenery along the route, may not be of interest. If you choose a new route that improves both commute time and safety, you have *optimized* your commute. If you instead choose a route that improves safety (e.g., avoiding a dangerous intersection) at the expense of increased commute time, you have made a *tradeoff* (and perhaps wisely so).

Figure 6.3 illustrates optimizations versus tradeoffs for three different starting designs, involving the criteria of delay and size, smaller being better for each criteria. Obviously, optimizations are preferred over tradeoffs, since optimizations improve both criteria, or at least improve one criteria without worsening another criteria, as shown by the horizontal and vertical arrows in Figure 6.3(a). But sometimes one criteria can't be improved without worsening another criteria. For example, if a car designer wants to improve a car's fuel efficiency, the designer may have to make the car smaller—a tradeoff between the criteria of fuel efficiency and comfort.

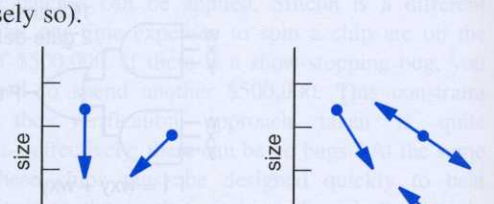


Figure 6.3 Optimizations (a), versus tradeoffs (b).

Some criteria commonly of interest to digital system designers include:

- ***Performance***: a measure of execution time for a computation on the system.
- ***Size***: a measure of the number of transistors, or silicon area, of a digital system.
- ***Power***: a measure of the energy consumed by the system per second, directly relating to both the heat generated by the system and to the battery energy consumed by computations.

Dozens of other criteria exist.

Optimizations and tradeoffs can be made throughout nearly all stages of digital design. This chapter describes some common optimizations and tradeoffs for typical criteria, at various levels of design.

▶ 6.2 COMBINATIONAL LOGIC OPTIMIZATIONS AND TRADEOFFS

Chapter 2 described how to design combinational logic, namely how to convert desired combinational behavior into a circuit of gates. Optimization and tradeoff methods can be applied to make those circuits better.

Two-Level Size Optimization Using Algebraic Methods

Implementing a Boolean function using only two levels of gates—a level of AND gates followed by one OR gate—usually results in a circuit having minimum delay. Recall from Chapter 2 that any Boolean equation can be written in sum-of-products form, simply by "multiplying out" the equation—for example, `xy(w+z) = xyw + xyz`. Thus, any Boolean function can be implemented using two levels of gates, simply by converting its equation to sum-of-products form and then using AND gates for the products followed by an OR gate for the sum.

A popular optimization is to *minimize the number of transistors* of a two-level logic circuit implementation of a Boolean function. Such optimization is traditionally called *two-level logic optimization*, or sometimes *two-level logic minimization*. We'll refer to it as ***two-level logic size optimization***, to distinguish such optimization from the increasingly popular optimizations of *performance* and *power*, as well as from other possible digital design optimizations.

In the 1970s/1980s, when transistors were costly (e.g., cents each), logic optimization was synonymous with size minimization, which dominated digital design. Today's cheaper transistors (e.g., 0.0001 cents each) make optimizations of other criteria equally or more critical.

Optimizing size requires a method to determine the number of transistors for a given circuit. A common method for quickly estimating the number of transistors:

- Assumes that every logic gate input requires two transistors. So a 3-input logic gate (whether an AND, OR, NAND, or NOR) would require 3 * 2 = 6 transistors. The circuits inside logic gates shown in Section 2.4 reveal why two transistors per gate input is a common assumption.
- Ignores inverters when determining the number of transistors, for simplicity.

The problem of two-level logic size optimization can be viewed algebraically as the problem of *minimizing the number of literals and terms of a Boolean equation that is in sum-of-products form*. The reason the problem can be viewed algebraically is because, as shown in Chapter 2, a sum-of-products Boolean equation can be translated directly to a circuit using a level of AND gates followed by an OR gate. For example, the equation `F = wxy + wxy'` from Figure 6.1(a) has six literals, `w`, `x`, `y`, `w`, `x`, and `y'`, and two terms, `wxy` and `wxy'`, for a total of 6 + 2 = 8 literals and terms. Each literal and each term translates approximately to a gate input in a circuit, as shown in Figure 6.1(a)—the literals translate to AND gate inputs, and the terms translate to OR gate inputs. The circuit thus has 3 + 3 + 2 = 8 gate inputs. With two transistors per gate input, the circuit has 8 * 2 = 16 transistors. The number of literals and terms can be minimized algebraically: `F = wxy + wxy' = wx(y+y') = wx`. That equation has two literals, `w` and `x`, resulting in 2 gate inputs, or 2 * 2 = 4 transistors, as shown in Figure 6.1(b). Note that a one-term equation does not require an OR gate, so the term is not counted for the transistor estimate. Likewise, a one-literal term would not require an AND gate and so that literal would not be counted.

Example 6.1 Two-level logic size optimization using algebraic methods

Minimize the number of literals and terms in a two-level implementation of the equation

```
F = xyz + xyz' + x'y'z' + x'y'z
```

Minimization can be done using algebraic transformations:

```
F = xy(z + z') + x'y'(z + z')
F = xy*1 + x'y'*1
F = xy + x'y'
```

There doesn't seem to be any further minimization possible. Thus, the equation has been transformed from having 12 literals and 4 terms (meaning 12 + 4 = 16 gate inputs, or 32 transistors), down to having only 4 literals and 2 terms (meaning 4 + 2 = 6 gate inputs, or 12 transistors).

The previous example showed the most common algebraic transformation used to simplify a Boolean equation in sum-of-products form, a transformation that can be written as:

```
ab + ab' = a(b+b') = a*1 = a
```

The transformation is sometimes called ***combining terms to eliminate a variable***, and is known formally as the ***uniting theorem***. The previous example applied the transformation twice, once with `xy` being `a` and with `z` being `b`, and a second time with `x'y'` being `a` and with `z` being `b`.

Sometimes a term must be duplicated to increase opportunities for combining terms to eliminate a variable, as illustrated in the next example.

Example 6.2 Reusing a term during two-level logic size optimization

Minimize the number of literals and terms in a two-level implementation of the equation

```
F = x'y'z' + x'y'z + x'yz
```

You might notice two opportunities to combine terms to eliminate a variable:

```
1: x'y'z' + x'y'z = x'y'
2: x'y'z + x'yz = x'z
```

Notice that the term `x'y'z` appears in both opportunities, but that term only appears once in the original equation. We'll therefore first replicate the term in the original equation (such replication doesn't change the function, because `a = a + a`) so that we can use the term twice when combining terms to eliminate a variable, as follows:

```
F = x'y'z' + x'y'z + x'yz
F = x'y'z' + x'y'z + x'y'z + x'yz
F = x'y'(z+z') + x'z(y'+y)
F = x'y' + x'z
```

After combining terms to eliminate a variable, the resulting term might also be combinable with other terms to eliminate a variable, as shown in the following example.

Example 6.3 Repeatedly combining terms to eliminate a variable

Minimize the number of literals and terms in a two-level implementation of the equation

```
G = xy'z' + xy'z + xyz + xyz'
```

We can combine the first two terms to eliminate a variable, and the last two terms also:

```
G = xy'(z'+z) + xy(z+z')
G = xy' + xy
```

We can combine the two remaining terms to eliminate a variable:

```
G = xy' + xy
G = x(y'+y)
G = x
```

In the previous examples, how did we "see" the opportunities to combine terms to eliminate a variable? The examples' original equations happened to be written in a way that made seeing the opportunities easy—terms that could be combined were side-by-side. Suppose instead the equation in Example 6.1 had been written as

```
F = x'y'z + xyz + xyz' + x'y'z'
```

That's the same function, but the terms appear in a different order. We might see that the middle two terms can be combined:

```
F = x'y'z + xyz + xyz' + x'y'z'
F = x'y'z + xy(z+z') + x'y'z'
F = x'y'z + xy + x'y'z'
```

But then we might not see that the left and right terms can be combined. We therefore might stop minimizing, thinking that we had obtained a fully minimized equation.

There is a visual method to help us *see* opportunities to combine terms to eliminate a variable, a method that the next section describes.

A Visual Method for Two-Level Size Optimization—K-Maps

Karnaugh Maps, or ***K-maps*** for short, are a visual method intended to assist humans to algebraically minimize Boolean equations having a few (two to four) variables. They actually are not commonly used any longer in design practice, but nevertheless, they are a very effective means for *understanding* the basic optimization methods underlying today's automated tools. A K-map is essentially a graphical representation of a truth table, meaning a K-map is yet another way to represent a function (other ways including an equation, truth table, and circuit). The idea underlying a K-map is to graphically place minterms adjacent to one another if those minterms differ in one variable only, so that we can actually "see" the opportunity for combining terms to eliminate a variable.

Three-Variable K-Maps

Figure 6.4 shows a K-map for the equation

```
F = x'y'z + xyz + xyz' + x'y'z'
```

which is the equation from Example 6.1 but with terms appearing in a different order. The map has eight cells, one for each possible combination of variable values. Let's examine the cells in the top row. The upper-left cell corresponds to `xyz=000`, meaning `x'y'z'`. The next cell to the right corresponds to `xyz=001`, meaning `x'y'z`. The next cell to the right corresponds to `xyz=011`, meaning `x'yz`. And the rightmost top cell corresponds to `xyz=010`, meaning `x'yz'`. Notice that the ordering of those top cells is *not* in increasing binary order. Instead, the order is `000`, `001`, *`011`*, `010`, rather than `000`, `001`, *`010`*, `011`. The ordering is such that *adjacent cells differ in exactly one variable*. For example, the cells for `x'y'z` (`001`) and `x'yz` (`011`) are adjacent, and differ in exactly one variable, namely `y`. Likewise, the cells for `x'y'z'` and `xy'z'` are adjacent, and differ only in variable `x`. The map is also assumed to have its *left and right edges adjacent*, so the rightmost top cell (`010`) is adjacent to the leftmost top cell (`000`)—note that those cells too differ in exactly one variable. Adjacent means abutted either horizontally or vertically, but *not diagonally*, because diagonal cells differ in more than one variable. Adjacent bottom-row cells also differ in exactly one variable. And cells in a column also differ in exactly one variable.

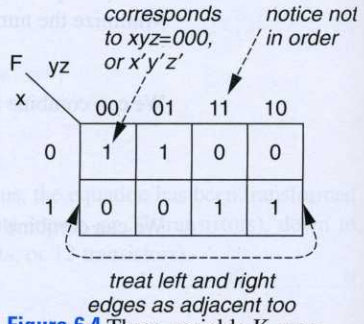


Figure 6.4 Three-variable K-map.

In a K-map, adjacent cells differ in exactly one variable.

A Boolean function can be represented as a K-map by placing `1`s in the cells corresponding to the function's minterms. So for the equation `F` above, `1`s are placed in cells corresponding to minterms `x'y'z`, `xyz`, `xyz'`, and `x'y'z'`, as shown in Figure 6.4. `0`s are placed in the remaining cells. Notice that a K-map is just another representation of a truth table. Rather than showing the output for every possible combination of inputs using a table, a K-map uses a graphical map. Therefore, a K-map is yet another representation of a Boolean function, and in fact is another standard representation.

*K-maps enable us to **see** opportunities to combine terms to eliminate a variable.*

The usefulness of a K-map for size minimization is that, because the map is designed such that adjacent cells differ in exactly one variable, then we know that *two adjacent `1`s in a K-map indicate that we can combine the two minterms to eliminate a variable*. In other words, a K-map lets us easily see when two terms can be combined to eliminate a variable. Such combining is indicated by drawing a circle around two adjacent `1`s, and then writing the term that results after the differing variable is removed. The following example illustrates.

Example 6.4 Two-level logic size optimization using a K-map

Minimize the number of literals and terms in a two-level implementation of the equation

```
F = xyz + xyz' + x'y'z' + x'y'z
```

Note that this is the same equation as in Example 6.1. The K-map representing the function is shown in Figure 6.5. Adjacent `1`s exist at the upper left of the map, so we circle those `1`s to yield the term `x'y'`—in other words, *the circle is a shorthand notation* for `x'y'z' + x'y'z = x'y'`. Likewise, adjacent `1`s exist at the bottom right cell of the map, so we draw a circle representing `xyz + xyz' = xy`. Thus, `F = x'y' + xy`.

The term "circle" is used even though the shape may be an oval or other shape.

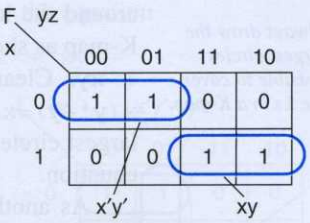


Figure 6.5 Minimizing a three-variable function using a K-map.

Recall from Example 6.3 that sometimes terms can be repeatedly combined to eliminate a variable, resulting in even fewer terms and literals. That example can be redone using a different order of simplifications as follows:

```
G = xy'z' + xy'z + xyz + xyz'
G = x(y'z'+ y'z + yz + yz')
G = x(y'(z'+z) + y(z+z'))
G = x(y'+y)
G = x
```

Notice that the second line ANDs `x` with the OR of all possible combinations of variables `y` and `z`. Obviously, one of those combinations of `y` and `z` will be true for any values of `y` and `z`, and thus the subexpression in parentheses will always evaluate to `1`, as was algebraically affirmed in the remaining lines above.

In addition to helping us see when two minterms can be combined to eliminate a variable, K-maps provide a graphical way to see when four minterms can be combined to eliminate two variables, as done above for `G`. We need to look for four `1`s in adjacent cells, where the cells form either a rectangle or a square (but not a shape like an "L"). Those four cells will have one variable the same and all possible combinations of the other two variables. Figure 6.6 shows the above function `G` as a three-variable K-map. The K-map has four adjacent `1`s in the bottom row. The four minterms corresponding to those `1`s are `xy'z'`, `xy'z`, `xyz`, and `xyz'`— note that `x` is the same in all four minterms, while all four combinations of `y` and `z` appear in those minterms. A circle drawn around the bottom four `1`s represents the simplification of `G` shown in the equations above. The result is `G = x`. In other words, the circle is a shorthand notation for the algebraic simplification of `G` shown in the five equations above.

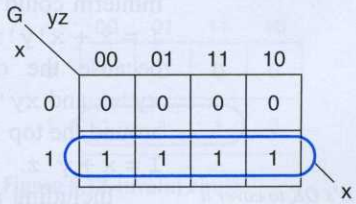


Figure 6.6 Four adjacent `1`s.

Always draw the largest circles possible to cover the 1s in a K-map.

Note that circles could have been drawn around the left two 1s and the right two 1s of the K-map as shown in Figure 6.7, resulting in G = xy' + xy. Clearly, G can be further simplified to x(y'+y)=x. Thus, we should always draw the largest circle possible in order to best minimize the equation.

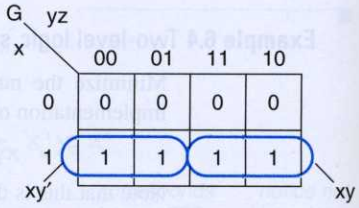


Figure 6.7 Non-optimal circles.

As another example of four adjacent 1s, consider the equation

```
H = x'y'z + x'yz + xy'z + xyz
```

Figure 6.8 shows the K-map for that equation's function. Circling the four adjacent 1s yields the minimized equation H = z.

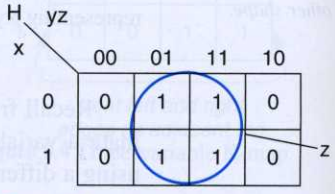


Figure 6.8 Four adjacent 1s.

Sometimes circles need to be drawn that include the same 1 twice. Consider the following equation:

```
I = x'y'z + xy'z' + xy'z
  + xyz + xyz'
```

Figure 6.9 shows the K-map for that equation's function. A circle can be drawn around the bottom four 1s to reduce those four minterms to just x. But that leaves the single 1 in the top row, corresponding to minterm x'y'z. That minterm must be somehow included in the minimized equation, since leaving that minterm out would change the function. The minterm could be ORed with the equation, yielding I = x + x'y'z, but that equation is not minimized because the original equation included minterm xy'z, and xy'z + x'y'z = (x+x')y'z = y'z. On the K-map, a circle can be drawn around the top 1 that also includes the 1 in the cell below. The minimized function is thus I = x + y'z.

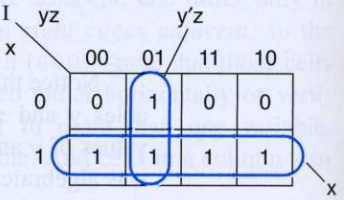


Figure 6.9 Circling a 1 twice.

It's OK to cover a 1 more than once to minimize multiple terms.

Including a 1 twice in different circles doesn't change the function, because doing so is the same as duplicating a minterm. Duplicating a minterm doesn't change the function, because a = a + a. The algebraic equivalent of the twice-circled 1 of Figure 6.9 is:

```
I = x'y'z + xy'z' + xy'z + xyz + xyz'
I = x'y'z + xy'z + xy'z' + xy'z + xyz + xyz'
I = (x'y'z + xy'z) + (xy'z' + xy'z + xyz + xyz')
I = (y'z) + (x)
```

The duplicated minterm resulted in better optimization.

On the other hand, there's no reason to circle 1s more than once if the 1s are already included in a minimized term. For example, the K-map for the equation

```
J = x'y'z' + x'y'z + xy'z + xyz
```

appears in Figure 6.10. There's no reason to draw the circle resulting in the term y'z. The other two circles cover all the 1s, meaning those two circles' terms cause the equation to output 1 for all the required input combinations. The third circle just results in an extra term without changing the function. Thus, not only should the largest possible circles be drawn to cover all the 1s, but the *fewest* circles should be drawn too.

Draw the fewest circles possible, to minimize the number of terms.

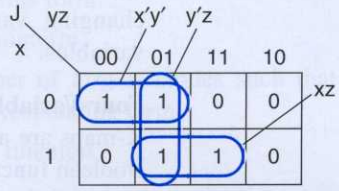


Figure 6.10 An unnecessary term.

As mentioned earlier, the left and right sides of a K-map are considered to be adjacent. Thus, circles can be drawn that wrap around the sides of a K-map. For example, the K-map for the equation

```
K = xy'z' + xyz' + x'y'z
```

appears in Figure 6.11. The two cells in the corners with 1s are adjacent since the left and right sides of the map are adjacent, and therefore one circle can be drawn that covers both, resulting in the term xz'.

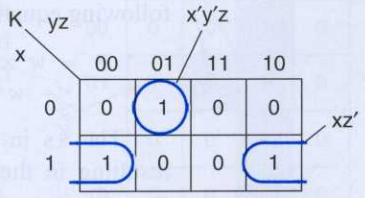


Figure 6.11 Sides are adjacent.

Sometimes a 1 does not have any adjacent 1s. In that case, a circle is drawn around that single 1, resulting in a term that is also a minterm. The term x'y'z in Figure 6.11 is an example of such a term.

A circle in a three-variable K-map must involve one cell, two adjacent cells, four adjacent cells, or eight adjacent cells. A circle can *not* involve only three, five, six, or seven cells. The circle must represent algebraic transformations that eliminate variables appearing in all possible combinations, since those variables can be factored out and then combined to a 1. Three adjacent cells don't have all combinations of two variables—one combination is missing. Thus, the circle in Figure 6.12 would not be valid, since it corresponds to xy'z' + xy'z + xyz, which doesn't simplify down to one term. To cover that function, two circles are needed, one around the left pair of 1s, the other around the right pair.

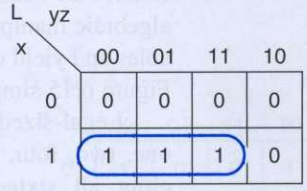


Figure 6.12 Invalid circle.

If all the cells in a K-map have 1s, like for the function E in Figure 6.13, then there are eight adjacent 1s. A circle can be drawn around those eight cells. Since that circle represents the ORing of all possible combinations of the function's three variables, and since obviously one of those combinations will be true for any combination of input values, the equation minimizes to just E = 1.

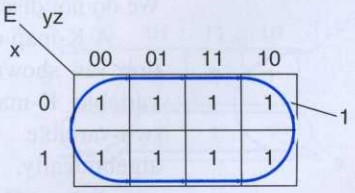


Figure 6.13 Four adjacent 1s.

Whenever in doubt as to whether a circle is valid, just remember that the circle represents a shorthand for algebraic transformations that combine terms to eliminate a variable. A circle must represent a set of terms for which all possible combinations of some variables appear while other variables are identical in all terms. The changing variables can be eliminated, resulting in a single term without those variables.

Four-Variable K-Maps

K-maps are also useful for minimizing four-variable Boolean functions. Figure 6.14 shows a four-variable K-map. Again, notice that every adjacent pair of cells in the K-map differs by exactly one variable. Also, the left and right sides of the map are considered adjacent, and the top and bottom edges of the map are also adjacent—note that the left and right cells differ by only one variable, as do the top and bottom cells.

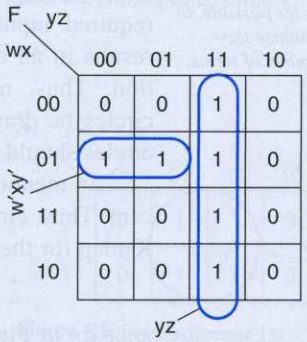


Figure 6.14 Four-variable K-map.

The K-map in the figure has been filled in for the following equation:

```
F = w'xy'z' + w'xy'z + w'x'yz
  + w'xyz + wxyz + wx'yz
```

The `1`s in the map can be covered with the two circles shown in Figure 6.14, resulting in the terms `w'xy'` and `yz`. The resulting minimized equation is therefore `F = w'xy' + yz`.

A circle covering eight adjacent cells would represent all combinations of three variables, so algebraic manipulation would eliminate all three variables and yield one term. For example, the function in Figure 6.15 simplifies to the single term `z` as shown.

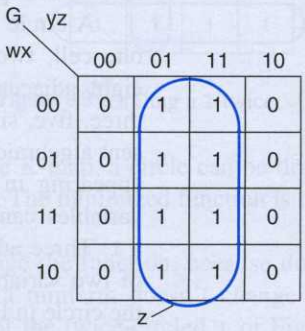


Figure 6.15 Eight adjacent cells.

Legal-sized circles in a four-variable K-map are one, two, four, eight, or sixteen adjacent cells. Circling all sixteen cells results in a function that equals `1`.

Larger K-Maps

K-maps for five and six variables have been proposed, but are rather cumbersome to use effectively. We do not discuss them further.

A K-map can be drawn for a two-variable function, as shown in Figure 6.16. However, a two-variable K-map isn't particularly useful, because two-variable functions are easy to minimize algebraically.

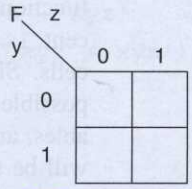


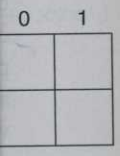
Figure 6.16 Two-variable K-map.

er that the circle
rms to eliminate
ble combinations
all terms. The
n without those

-variable K-map.

n in Figure 6.14,
ation is therefore

ght adjacent cells.



wo-variable K-map.

Using a K-Map

Given any Boolean function of three or four variables, the following method summarizes how to use a K-map to minimize the function:

1. *Convert* the function's equation into sum-of-minterms form.
2. *Place* a `1` in the appropriate K-map cell for each minterm.
3. *Cover* all the `1`s by drawing the *minimum* number of *largest* circles such that every `1` is included at least once, and write the corresponding term.
4. *OR* all the resulting terms to create the minimized function.

The first step, converting to sum-of-minterms form, can be done algebraically as in Chapter 2. Alternatively, many people find it easier to combine steps 1 and 2, by converting the function's equation to sum-of-products form (where each term is not necessarily a minterm), and then filling in the `1`s on the K-map corresponding to each term. For example, consider the four-variable function

```
F = w'xz + yz + w'xy'z'
```

The term `w'xz` corresponds to the two lightly shaded cells in Figure 6.17, so `1`s are placed in those cells. The term `yz` corresponds to the entire dark-shaded column in the figure. The term `w'xy'z'` corresponds to the single unshaded cell shown on the left with a `1`.

Minimization would proceed by covering the `1`s with circles and ORing all the terms. The function in Figure 6.17 is identical to the function in Figure 6.14, for which the obtained minimized equation was `F = w'xy' + yz`.

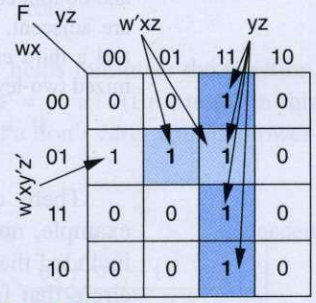


Figure 6.17 `w'xz` and `yz` terms.

Example 6.5 Two-level logic size optimization using a three-variable K-map

Minimize the following equation:

```
G = a + a'b'c' + b(c' + bc')
```

Let's begin by converting the equation to sum-of-products:

```
G = a + a'b'c' + bc' + bc'
```

We place `1`s in a three-variable K-map corresponding to each term as in Figure 6.18. The bottom row corresponds to the term `a`, the top left cell to term `a'b'c'`, and the right column to the term `bc'` (which happens to appear twice in the equation).

We then cover the `1`s using the two circles shown in Figure 6.19. ORing the resulting terms yields the minimized equation `G = a + c'`.

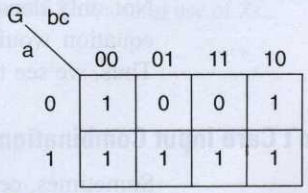


Figure 6.18 Terms on the K-map.

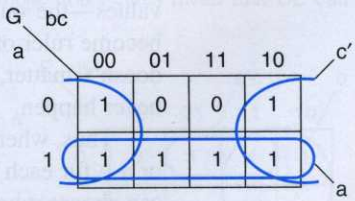


Figure 6.19 A cover.

Example 6.6 Two-level logic size optimization using a four-variable K-map

Minimize the following equation:

```
H = a'b'(cd' + c'd') + ab'c'd' + ab'cd' + a'bd + a'bcd'
```

Converting to sum-of-products form yields

```
H = a'b'cd' + a'b'c'd' + ab'c'd' + ab'cd' + a'bd + a'bcd'
```

We fill in the 1s corresponding to each term, resulting in the K-map shown in Figure 6.20. The term a'bd corresponds to the two cells whose 1s are in italics. All the other terms are minterms and thus correspond to one cell.

We cover the 1s using circles as shown. One "circle" covers the four corners, resulting in the term b'd'. That circle may look strange, but remember that the top and bottom cells are adjacent, and the left and right cells are adjacent. Another circle results in the term a'bd, and a third circle results in the term a'bc. The minimized two-level equation is therefore:

```
H = b'd' + a'bc + a'bd
```

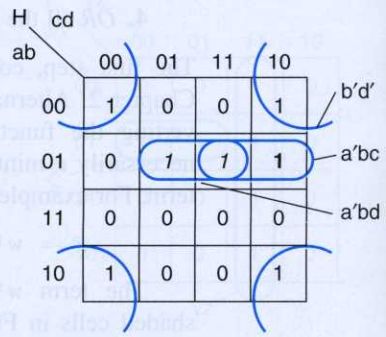


Figure 6.20 K-map example.

There can be many different minimized equations for the same function. For example, note the bolded 1 in Figure 6.20. We covered that 1 by drawing a circle that included the 1 to the left, yielding the term a'bc. Alternatively, we could have drawn a circle that included the 1 above, yielding the term a'cd', resulting in the minimized equation

```
H = b'd' + a'cd' + a'bd
```

Not only does that equation represent the same function as the previous equation, that equation would also require the same number of transistors as the previous equation. Thus, we see that there may be multiple minimized equations that are equally good.

Don't Care Input Combinations

Sometimes, certain input combinations, known as ***don't care*** combinations, of a Boolean function are guaranteed to never appear. For those combinations, we don't care whether the function outputs a 1 or a 0, because the function will never actually see those input values—the output for those inputs just doesn't matter. As an intuitive example, if you become ruler of the world, will you live in a palace or a castle? The output (your answer) doesn't matter, because the input (you becoming ruler of the world) is guaranteed to never happen.

Thus, when given a don't care input combination, we can choose whether to output a 1 or a 0 for each input combination, such that we obtain the best minimization possible. We can choose whatever output yields the best minimization, because the output for those don't care input combinations doesn't matter, as those combinations simply won't happen.

Algebraically, don't care terms can be introduced into an equation during algebraic minimization to create the opportunity to combine terms to eliminate a variable. As a simple example, consider a function `F = xy'z'`, for which it is also guaranteed that the terms `x'y'z'` and `xy'z` can each never evaluate to `1`. Notice that introducing the first don't care term to the equation would result in `F = xy'z' + x'y'z' = (x+x')y'z' = y'z'`. Thus, introducing that don't care term `x'y'z'` into the equation yields a minimization benefit. However, continuing by introducing the second don't care term does not yield such a benefit, so there is no need to introduce that term too.

In a K-map, don't care input combinations can be easily handled by placing an `X` in a K-map for each don't care minterm. The `X`s don't *have* to be covered with circles, but some `X`s *can* be covered if doing so enables drawing bigger circles while covering the `1`s, meaning fewer literals will appear in the term corresponding to the circle. The above function `F` can be converted to the K-map shown in Figure 6.21, having a `1` corresponding to `xy'z'` when the function *must* output `1`, and having two `X`s corresponding to `x'y'z'` and `xy'z` when the function *may* output `1` if that helps minimize the function. Drawing a single circle results in the minimized equation `F = y'z'`. (Be careful in this discussion not to confuse the uppercase `X`, corresponding to a don't care, with the lowercase `x`, corresponding to a variable.)

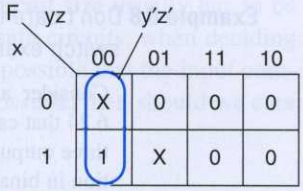


Figure 6.21 K-map with don't cares.

Remember, don't cares don't *have* to be covered. The cover in Figure 6.22 gives an example of a wasteful use of don't cares. The circle covering the bottom `X`, yielding term `xy'`, is not needed. That term is not wrong, because we don't care whether the output is `1` or `0` when `xy'z` evaluates to `1`. But that term yields a larger circuit because the resulting equation is `F = y'z' + xy'`. Since we don't care, it is better to make the output `0` when `xy'z` is `1` to yield a smaller circuit.

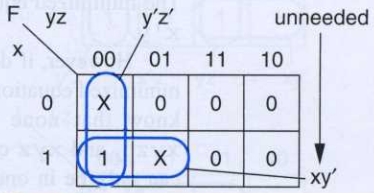


Figure 6.22 Wasteful use of `X`s.

Example 6.7 Two-level logic size optimization with don't cares on a K-map

Minimize the following equation

```
F = a'bc' + abc' + a'b'c
```

given that terms `a'bc` and `abc` are don't cares. Intuitively, those don't cares mean that `bc` can never be `11`.

We begin by creating the 3-variable K-map in Figure 6.23. We place `1`s in the three cells for the function's minterms. We then place `X`s in the two cells for the don't cares. We can cover the upper-left `1` using a circle that includes an `X`. Likewise, including the two `X`s in a circle covers the two `1`s on the right with a bigger circle. The resulting minimized equation is `F = a'c + b`.

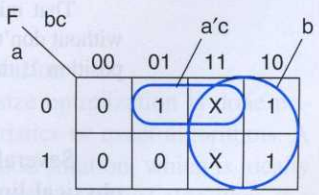


Figure 6.23 Using don't cares.

Without don't cares, the equation would have minimized to `F = a'b'c + bc'`. Assuming two transistors per gate input and ignoring inverters, the equation minimized without don't cares would require (3+2+2) * 2 = 14 transistors (3 gate inputs for the first AND gate, 2 for the second AND gate, and 2 for the OR gate, times 2 transistors per gate input). In contrast, the equation minimized with don't cares requires only (2 + 0 + 2)*2 = 8 transistors.

Example 6.8 Don't care input combinations in a sliding switch example

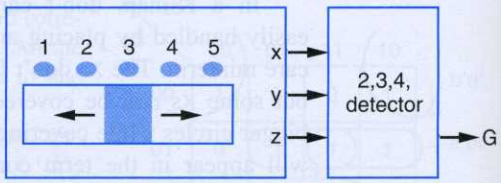


Figure 6.24 Sliding switch example.

Consider a sliding switch shown in Figure 6.24 that can be in one of five positions, with three outputs `x`, `y`, and `z` indicating the position in binary. So `xyz` can take on the values of `001`, `010`, `011`, `100`, and `101`. The other values for `xyz` are not possible, namely the values `000`, `110`, and `111`, meaning or `x'y'z'`, `xyz'`, and `xyz`. We wish to design combinational logic, with `x`, `y`, and `z` inputs, that outputs `1` if the switch is in position 2, 3, or 4, corresponding to `xyz` values of `010`, `011`, or `100`.

A Boolean equation describing the desired logic is: `G = x'yz' + x'yz + xy'z'`. We can minimize the equation using a K-map as shown in Figure 6.25. The minimized equation that results is: `G = xy'z' + x'y`.

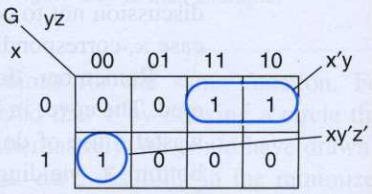


Figure 6.25 Without don't cares.

However, if don't cares are considered, a simpler minimized equation can be obtained. In particular, we know that none of the three minterms `x'y'z'`, `xyz'`, and `xyz` can ever be true, because the switch can only be in one of the above-stated five positions. So it doesn't matter whether a circuit outputs a `1` or a `0` for those three other minterms.

We can include these don't care input combinations as `X`s on the K-map as shown in Figure 6.26. When covering the `1`s in the top right, a larger circle can now be drawn, resulting in the term `y`. When covering the `1` at the bottom left, a larger circle can also be drawn, resulting in the term `z'`. Although all `X`s were covered in this example, recall that not all `X`s need be covered. The minimized equation that results is: `G = y + z'`.

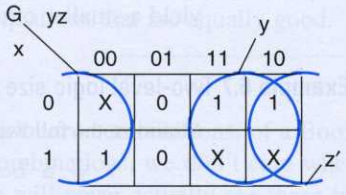


Figure 6.26 With don't cares.

That minimized equation using don't cares looks a lot different than the minimized equation without don't cares. But keep in mind the circuit still works the same. For example, if the switch is in position `1`, then `xyz` will be `001`, so `G = y + z'` evaluates to `0` as desired.

Several common situations lead to don't cares. Sometimes don't cares come from physical limits on the inputs—a switch can't be in two positions at once, for example. If you've read Chapter 5, then you may realize that another common situation where don't

cares arise is in the controller of a datapath. If the controller isn't reading or writing to a particular memory or register file in a given state, then it doesn't matter what address appears at the memory or register file during that state. Likewise, if a mux feeds into a register and the controller isn't loading the register in a given state, then the mux select value is a don't care in that state because it doesn't matter which mux data input passes through the mux during that state. If a controller isn't going to load the output of an ALU into a register in a given state, then it doesn't matter what function the ALU computes during that state.

Don't cares must be used with caution. The criteria of circuit size usually has to be balanced with other criteria, like reliable, error-tolerant, and safe circuits, when deciding whether to use don't cares. We must ask ourselves—is it *ever* possible that the input combination might occur, even in an error situation? And if it is possible, then should we care what the circuit outputs in that situation?

Automating Two-Level Logic Size Optimization

Visual Use of K-Maps Is Limited

Although the visual K-map method is helpful in two-level optimization of three- and four-variable functions, the visual method is unmanageable for functions with many more variables. One problem is that we can't effectively visualize maps beyond 5 or 6 variables. Another problem is that humans make mistakes, and might accidentally not draw the biggest circle possible on a K-map. Furthermore, the order in which a designer begins covering `1`s may result in a function that has more terms than would have been obtained using a different order. For example, consider the function shown in the K-map of Figure 6.27(a). Starting from the left, a designer might first draw the circle yielding the term `y'z'`, then the circle yielding `x'y'`, then the circle yielding `yz`, and finally the circle yielding `xy`, for a total of four terms. The K-map in Figure 6.27(b) shows an alternative cover. After drawing the circle yielding the term `y'z'`, the designer draws the circle yielding `x'z`, and then the circle yielding `xy`. The alternative cover uses only three terms instead of four.

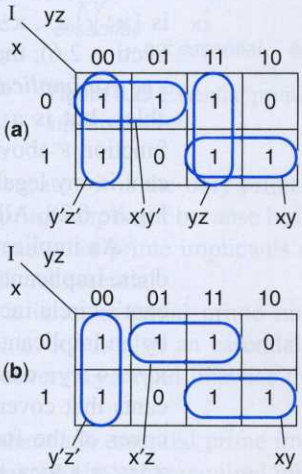


Figure 6.27 A cover is not necessarily optimal: (a) a four-term cover, and (b) a three-term cover of the same function.

Concepts Underlying Automated Two-Level Size Optimization

Because of the above-mentioned problems, two-level logic size optimization is done primarily using automated computer-based tools executing heuristics or exact algorithms. A ***heuristic*** is a problem solving method that *usually* yields a good solution, which is ideally close to the optimal, but *not necessarily* optimal. An ***exact algorithm***, or just algorithm, is a

problem solving method that yields the optimal solution. An ***optimal solution*** is as good as or better than any other possible solution with respect to the criteria of interest.

This section now defines some concepts underlying heuristics and exact algorithms for two-level logic size optimization. Those concepts will be illustrated graphically on K-maps, but such illustration is only intended to provide the reader with an intuition of the concepts—automated tools do *not* use K-maps.

Recall that a function can be written as a sum-of-minterms equation. A ***minterm*** is a product term that includes all the function's variables exactly once, in either true or complemented form. The ***on-set*** of a Boolean function is the set of minterms that define when the function should evaluate to `1` (i.e., when the function is "on"). Consider the function `F = x'y'z + xyz' + xyz`, whose K-map representation is shown in Figure 6.28. `F`'s on-set is `{x'y'z, xyz, xyz'}`. The ***off-set*** of a Boolean function consists of all the minterms that define when the function should evaluate to `0`. `F`'s off-set is `{x'y'z', x'yz', x'yz, xy'z', xy'z}`. Using compact minterms representation (see Section 2.6), the on-set is {1,6,7} and the off-set is {0,2,3,4,5}.

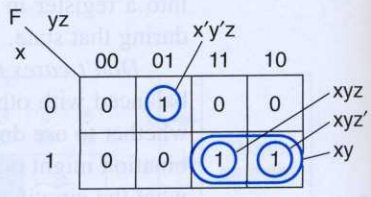


Figure 6.28 Minterms (the three smaller circles) and implicants (all the circles).

An ***implicant*** is a product term that may include fewer than all the function's variables, but is a term that only evaluates to `1` if the function should evaluate to `1`. The function `F` above has four implicants: `x'y'z`, `xyz'`, `xyz`, and `xy`. Graphically, an implicant is any legal (but not necessarily the biggest possible) circle on a K-map, as shown in Figure 6.28. All minterms are obviously implicants, but not all implicants are minterms.

An implicant ***covers*** a minterm if the implicant evaluates to `1` whenever the minterm does. Implicant `xy` covers minterms `xyz'` and `xyz` of function `F`. Graphically, an implicant's circle includes the `1`s of the covered minterms. Replacing minterms in an equation by an implicant that covers those minterms does not change the function. For function `F`, `xyz'+xyz` can be replaced by `xy` because `xy` covers those two minterms. A set of implicants that covers the on-set of a function (and covers no other minterms) is known as a cover of the function or ***function cover***. For the above function, one function cover is `x'y'z + xyz + xyz'`. Another cover is `x'y'z + xy`. Yet another cover is `x'y'z + xyz + xyz'+ xy`.

"Expanding" seems like the wrong word to describe changing `xyz` *to* `xy`*, because the term gets smaller. But* `xy` *covers more minterms, so it is expanded. The K-map visualization should make this point clear—*`xy`*'s circle is* bigger.

Removing a variable from a term is known as ***expanding*** the term, which is the same as expanding the size of a circle on a K-map. For example, the term `xyz` can be expanded by removing `z`, leading to the term `xy`. If the original term is an implicant of a function, a new term obtained by expanding the original term may or may not be an implicant of the function. For example, for function `F` in Figure 6.28, expanding the term `xyz` to the term `xy` results in an implicant of the function. Expanding the term `xyz'` to `xy` also results in an implicant (the same one). But expanding `xyz` to `xz` (by eliminating `y`) does not result in an implicant—`xz` covers minterm `xy'z`, which is not in the function's on-set.

A ***prime implicant*** of a function is an implicant with the property that if any variable were eliminated from the implicant, the result would be a term covering a minterm not in the function's on-set; a prime implicant cannot be expanded into another implicant. Graphically, a prime implicant corresponds to circles that are the largest pos-

sible—enlarging the circle further would result in covering 0s, which changes the function. In Figure 6.28, x'y'z and xy are prime implicants. Removing any variable from implicant x'y'z would result in a term that covers a minterm that is not in the on-set. Removing any variable from implicant xy would also result in a term that covers minterms not in the on-set. On the other hand, xyz is not a prime implicant, because z can be removed from that implicant without changing the function; xy covers minterms xyz and xyz', both of which are in the on-set. Likewise, xyz' is not a prime implicant, because z' can be removed. There is usually no need to cover a function with anything other than prime implicants, because a prime implicant achieves the same function with fewer literals than non-prime implicants. This is equivalent to saying that we should always draw the biggest circles possible in K-maps.

An ***essential prime implicant*** is a prime implicant that is the *only* prime implicant that covers a particular minterm in a function's on-set. Graphically, an essential prime implicant is the only circle (the largest possible, of course, since the circle must represent a prime implicant) that covers a particular 1. A nonessential prime implicant is a prime implicant whose covered minterms are also covered by one or more other prime implicants. Figure 6.29 shows a function G that has four prime implicants, but only two of which are essential. x'y' is an essential prime implicant because it is the only prime implicant that covers minterm x'y'z'. xy is an essential prime implicant because it is the only prime implicant that covers minterm xyz'. y'z is a nonessential prime implicant because both of its covered minterms are covered by other implicants (those other prime implicants may or may not be essential prime implicants). Likewise, xz is not essential.

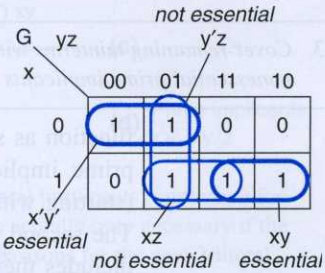


Figure 6.29 Essential prime implicants.

For the earlier function F in Figure 6.28, x'y'z is an essential prime implicant because it is the only prime implicant covering x'y'z. xy is also an essential prime implicant because it is the only prime implicant that covers xyz (and likewise the only prime implicant that covers xyz').

The significance of essential prime implicants is that each essential prime implicant *must* be included in a function's cover, otherwise there would be some required minterms that would not be covered by any prime implicant. On the other hand, each nonessential prime implicant may or may not be included in function's cover. In Figure 6.29, neither of the nonessential prime implicants, y'z and xz, must appear in the function's cover. Of course, in this case, one of them must appear in the function cover in order to cover minterm xy'z (the bottom left 1 in the K-map), but neither of them individually is essential.

The Tabular Method (Quine-McCluskey)

Given the notions of prime implicants and essential prime implicants, an automatable approach for two-level logic size optimization is shown in Table 6.1, known as the ***tabular method***. The method was originally developed by Quine and McCluskey and is commonly referred to as the ***Quine-McCluskey method***. The method begins with a function's minterms, which can be obtained algebraically from any equation describing the

TABLE 6.1 Automatable tabular method for two-level logic size optimization.

	Step	Description
1	*Determine prime implicants*	Starting with minterm implicants, methodically compare all pairs (actually, all pairs whose numbers of uncomplemented literals differ by one) to find opportunities to combine terms to eliminate a variable, yielding new implicants with one less literal. Repeat for new implicants. Stop when no implicants can be combined. All implicants not covered by a new implicant are prime implicants.
2	*Add essential prime implicants to the function's cover*	Find every minterm covered by only one prime implicant, and denote that prime implicant as essential. Add essential prime implicants to the cover, and mark all minterms covered by those implicants as already covered.
3	*Cover remaining minterms with nonessential prime implicants*	Cover the remaining minterms using the minimal number of remaining prime implicants.

function as shown in Chapter 2. The first step of the tabular method is to determine all prime implicants of the function by methodically comparing all pairs of implicants (starting with minterms) to find opportunities to combine terms to eliminate a variable. The second step finds the essential prime implicants among those prime implicants and includes them in the function's cover (because by definition, essential prime implicants must be included in the cover), and notes all the minterms that are covered by those essential prime implicants. The last step covers remaining minterms using the fewest remaining prime implicants. The steps will be illustrated using the following function:

```
F = x'y'z' + x'y'z + x'yz + xy'z + xyz' + xyz
```

Step 1 finds all prime implicants. It begins by considering each minterm as an implicant; function `F` has 6 such implicants. The step compares every pair of implicants to detect opportunities to combine terms to eliminate a variable, as illustrated in Figure 6.30(a). For example, the first implicant `x'y'z'` is compared with each of the other implicants. When compared with the second implicant `x'y'z`, the result is `x'y'z' + x'y'z = x'y'`. On the other hand, when compared with the third implicant `x'yz`, no combining is possible (indicated by the word "No" in the figure); likewise, no combining is possible when compared with the other implicants ("No" is not shown for those). The second implicant is then compared with each of the third, fourth, fifth, and sixth implicants (it has already been compared with the first implicant). And so on for each of the remaining implicants.

In doing these comparisons, one may notice that combining terms to eliminate a variable is only possible if the number of uncomplemented literals in the two implicants differs by exactly one. For example, `x'y'z'` has 0 uncomplemented literals and `x'y'z` has 1 uncomplemented literal (`z`), so comparing those terms could possibly lead to them being combined. On the other hand, comparing `x'y'z'` with `x'yz`, which has 2 uncomplemented literals (`y` and `z`), makes no sense—those two terms can't possibly differ by only 1 literal and thus they could never be combined. Likewise, comparing implicants with the same number of uncomplemented literals makes no sense either because those implicants must have two (or more) variables whose complementing differs; e.g., `x'yz` and `xy'z` each has 2 uncomplemented literals and thus the two terms can't possibly be

Minterms (3-literal implicants)

(0) x'y'z'

(1) x'y'z

(3) x'yz

(5) xy'z

(6) xyz'

(7) xyz

Figure 6.30
opportuniti
number of
implicants,

*In digital desig automation (an computer algorithms in general), skipp items that cann possibly lead to good solution i known as **pruni** think of all poss solutions as bei the leaves of a tree's branches—skip items is like pruning branch of the tree.*

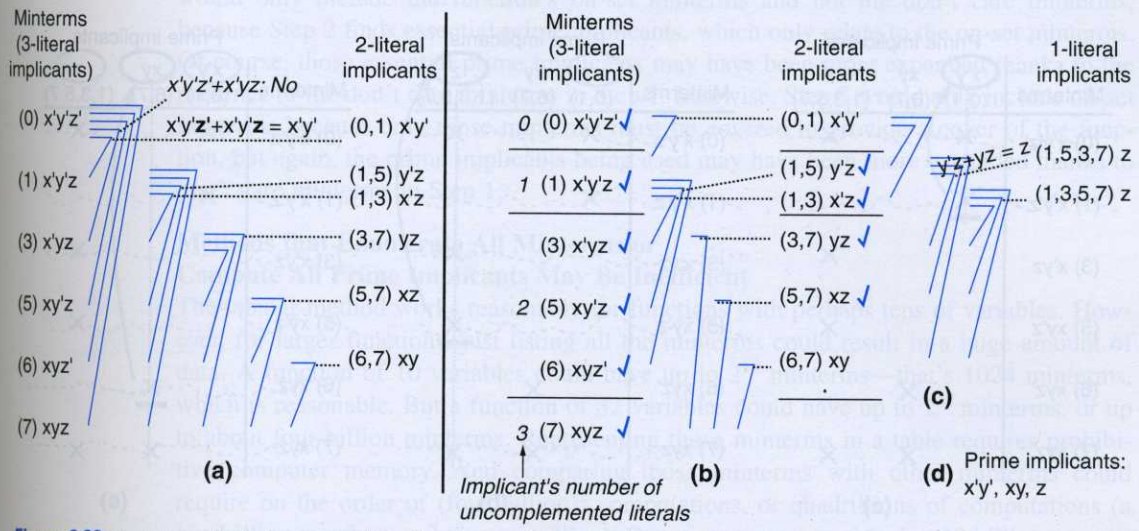


Figure 6.30 First step of tabular method for function F: (a) comparing all pairs of 3-literal implicants (minterms) for opportunities to combine terms to eliminate a variable, (b) comparison of two terms is actually only necessary if the number of uncomplemented literals in the terms differs by one, (c) repeating the comparisons for the new 2-literal implicants, (d) prime implicants are all implicants not covered by a lower-literal implicant.

combined. Thus, the minterms can be grouped according to their number of uncomplemented literals as shown in Figure 6.30(b), and then comparisons between items in groups whose number does not differ by exactly one can be skipped.

*In digital design automation (and in computer algorithms in general), skipping items that cannot possibly lead to a good solution is known as **pruning**; think of all possible solutions as being the leaves of a tree's branches—skipping items is like pruning branches of the tree.*

Comparing a pair may lead to a new implicant. The implicants in the pair cannot possibly be a prime implicants because each can be expanded into the new implicant. For example, `x'y'z'` and `x'y'z` cannot be prime implicants because each can be expanded into implicant `x'y'`. A check mark can be placed next to each such implicant as in Figure 6.30(b). Furthermore, keeping track of which minterms are covered by which implicants will be helpful, so each minterm's number (see compact sum-of-minterms representation in Chapter 2) is shown in parentheses next to the minterm, and each new implicant has in parentheses the list of minterms that the implicant covers.

The pairwise comparisons repeat for the new implicants; e.g., F's 2-literal implicants are compared as in Figure 6.30(c). Such comparisons again only compare implicants whose number of uncomplemented literals differs by exactly one. Furthermore, no comparison is necessary between implicants having a different number of literals, such as between the 3-literal and 2-literal implicants, because such comparisons couldn't possibly lead to terms being combined, due to the different numbers of literals.

The comparisons continue until no new implicants result. All unchecked implicants must be prime implicants, because those implicants could not be expanded into another implicant.

Step 2 determines all essential prime implicants and adds them to the function's cover. Determining essential prime implicants begins by creating a table as in Figure 6.31(a) with one column for each prime implicant (as determined in Step 1) and one row for each minterm. An "X" is placed in the table if the column's prime implicant covers

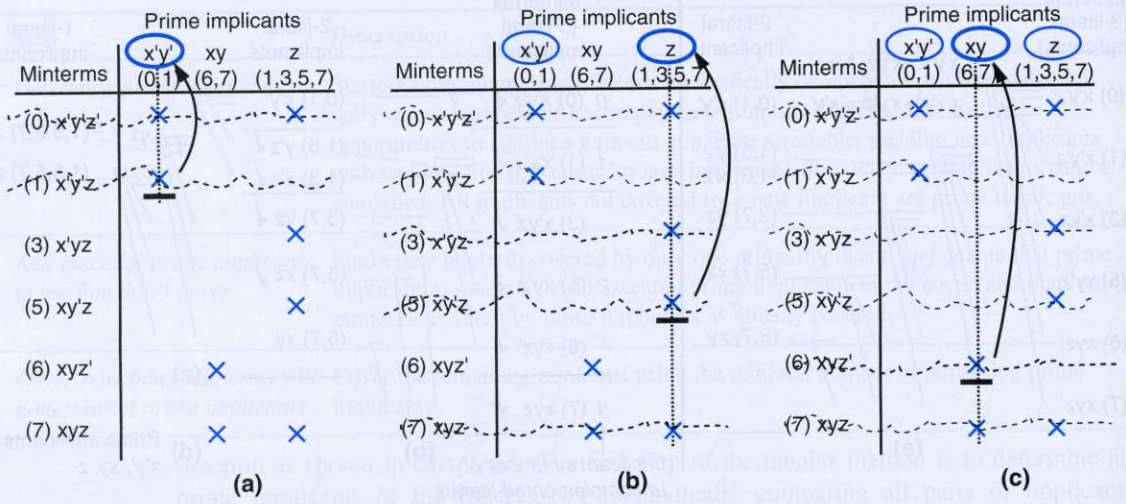


Figure 6.31 Second step of tabular method for function F: (a) the sole X (underlined) in the second row means that only prime implicant x'y' covers minterm x'y'z, so x'y' is an essential prime implicant. The prime implicant is circled to indicate it being added to F's cover, and all its covered minterms' rows are crossed out to indicate those minterms are now covered. (b) The fourth row also has a sole X, so prime implicant z is essential and thus added to the cover, and its covered minterms' rows are crossed out. (c) Prime implicant xy is also essential, so it is added and its rows crossed out. For F, the essential prime implicants cover all the minterms, and thus the method's third step is not needed. The final cover is F = x'y' + xy + z.

the row's minterm; keeping the list of minterm numbers with each prime implicant in Step 1 now makes it easy to determine which minterms are covered by a prime implicant.

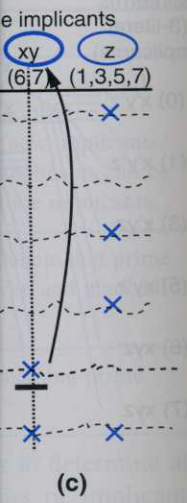
Next, each row is examined to see if only one X exists in the row. Figure 6.31(a) shows that the second row has a sole X, which means that prime implicant x'y' is the only prime implicant that covers minterm x'y'z and is therefore an essential prime implicant. x'y' is thus added to the function's cover, indicated by being circled. Now that x'y' is part of the function's cover, both minterms covered by x'y' are covered (indicated by the vertical dotted line), and thus their rows can be crossed out of the table as shown, because those minterms no longer need to be considered.

The search for rows with a sole X continues. A sole X is found in the fourth row, and thus prime implicant z is essential and so is added to the function's cover, and its covered minterms' rows are crossed out. A sole X is then found in the fifth row, and thus prime implicant xy is essential and is similarly added to the cover and its row crossed out.

Step 3 is to use the fewest remaining prime implicants to cover any remaining minterms. However, all minterms were covered by Step 2; i.e., all rows have been crossed out. Thus, for this function, the essential prime implicants completely cover the function. The final minimized equation is therefore F = x'y' + xy + z.

The reader may find it useful to minimize F using a K-map to help understand the minimization achieved by the tabular method.

Don't care minterms can be introduced into the tabular method by including them as function minterms during Step 1, which finds all prime implicants. However, Step 2



(c)

d row means that
prime implicant is
o indicate those
and thus added to
, so it is added and
thod's third step is

prime implicant in
a prime implicant.
ow. Figure 6.31(a)
plicant x'y' is the
an essential prime
being circled. Now
x'y' are covered
sed out of the table

the fourth row, and
ver, and its covered
ow, and thus prime
w crossed out.
ny remaining mint-
ve been crossed out.
er the function. The

help understand the

y including them as
s. However, Step 2

would only include the function's on-set minterms and not the don't care minterms, because Step 2 finds essential prime implicants, which only relate to the on-set minterms. Of course, those essential prime implicants may have been more expanded thanks to the presence of the don't care minterms in Step 1. Likewise, Step 3 would only include on-set minterms because only those minterms must be covered to provide a cover of the function, but again, the prime implicants being used may have been more expanded thanks to don't care minterms in Step 1.

Methods that Enumerate All Minterms or Compute All Prime Implicants May Be Inefficient

The tabular method works reasonably for functions with perhaps tens of variables. However, for larger functions, just listing all the minterms could result in a huge amount of data. A function of 10 variables could have up to 2^{10} minterms—that's 1024 minterms, which is reasonable. But a function of 32 variables could have up to 2^{32} minterms, or up to about four billion minterms. Representing those minterms in a table requires prohibitive computer memory. And comparing those minterms with other minterms could require on the order of (four billion)^2 computations, or quadrillions of computations (a quadrillion is a thousand times a trillion). Even a computer performing 10 billion computations per second would require 100,000 seconds to perform all those computations, or 27 hours. And for 64 variables, the numbers go up to 2^{64} possible minterms, or quadrillions of minterms, and quadrillions^2 of computations, which could require a month of computation. Functions with 100 inputs, which are not uncommon, would require an absurd amount of memory, and many years of computations. Even computing all prime implicants, without first listing all minterms, is computationally prohibitive for many modern-sized functions.

Iterative Heuristic for Two-Level Logic Size Optimization

Because enumerating all minterms of a function, or even just all prime implicants, is prohibitive in terms of computer memory and computation time for functions with many variables, most automated tools use methods that instead iteratively transform the original function's equation in an attempt to find improvements to the equation. ***Iterative improvement*** means repeatedly making small changes to an existing solution until the decision is made to stop, perhaps because changes are no longer yielding improvements, or perhaps because the tool has run for enough time. As an example of making small changes to an existing solution, consider the equation

```
F = abcdefgh + abcdefgh'+ jklmnop
```

Clearly, this equation can be reduced simply by combining the first two terms and removing variable `h`, resulting in `F = abcdefg + jklmnop`. However, enumerating the minterms, as required in the earlier-described size optimization methods, would have resulted in roughly 1000 minterms and then millions of computations to find the prime implicants. Such enumeration and computation are obviously not necessary to minimize this equation. An iterative improvement approach is better suited to this problem.

A simple iterative improvement heuristic that is reasonably effective uses repeated application of the expand operation. The ***expand*** operation removes a literal from a term. Trying all possible expands of every term may take too much compute time, and thus the heuristic randomly chooses which term to expand and randomly chooses how to expand that term. For example, consider the function `F = x'z + xy'z + xyz`. A heuristic might try to expand the term `x'z` by removing `x'`, or by removing `z`. Note that expanding a term *reduces* the number of literals—the concept that *expanding* a term *reduces* the number of literals in a term may take a while to get accustomed to. Thinking of K-map circles may help, as shown in Figure 6.32—the bigger the circle, the fewer the resulting literals. An expansion is ***legal*** if the new term covers only minterms in the function's on-set, or equivalently, does *not* cover a minterm in the function's off-set. In other words, an expansion is legal if the new term is still an implicant of the function. Figure 6.32(a) shows that expanding term `x'z` to `z` for the given function is legal because the expanded term covers only `1`s, whereas expanding `x'z` to `x'` is not legal because the expanded term covers at least one `0`. If an expansion is legal, the heuristic replaces the original term by the expanded term, and then *searches for and removes any other term covered by the expanded term.* In Figure 6.32(a), the expanded term `z` covers terms `xy'z` and `xyz`, so both those latter terms can be removed.

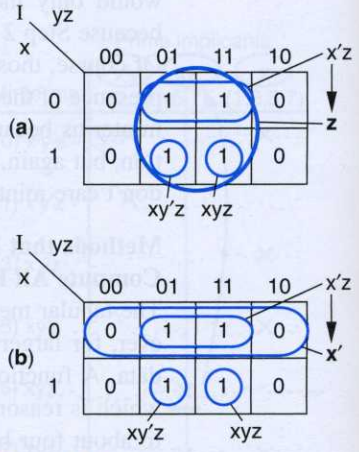


Figure 6.32 Expansions of term `x'z` in the function `F = x'z + xy'z + xyz`: (a) legal, (b) not legal (because the expanded term covers `0`s).

The expand operation was illustrated on a K-map merely to aid in understanding the intuition of the operation—K-maps are nowhere to be found in heuristic two-level logic size optimization tools.

As another example, consider the earlier introduced function

```
F = abcdefgh + abcdefgh'+ jklmnop
```

A heuristic might randomly choose to first try expanding the first term `abcdefgh`. It might randomly try expanding that term to `bcdefgh` (i.e., literal `a` has been removed). However, that term covers the term `a'bcdefgh`, which covers minterms that are not in the function's on-set, so that expansion is not legal. The heuristic might try other expansions, finding them not legal either, until the heuristic comes across the expansion to `abcdefg` (i.e., literal `h` was removed). That term covers only the minterms covered by `abcdefgh` and `abcdefgh'`, both of which are clearly implicants because they appear in the original function, and thus the new term must also be an implicant. Therefore, the heuristic replaces the first term by the expanded term:

```
F = abcdefg + abcdefgh'+ jklmnop
```

The heuristic then searches for terms covered by the new expanded term, and removes such terms:

```
F = abcdefgh + abcdefgh' + jklmnop
F = abcdefg + jklmnop
```

Thus, using just the expand operation, the heuristic improved the equation. There is no guarantee that the heuristic will find the above expansion that yields the size reduction, but for a relatively small amount of compute time—far less than required for exact two-level logic size optimization—the heuristic's random search is very likely to find that expansion.

Example 6.9 Iterative heuristic for two-level logic size optimization using *expand*

Minimize the following equation, which was also minimized in Example 6.4, using repeated random application of the expand operation:

```
F = xyz + xyz' + x'y'z' + x'y'z
```

In other words, the on-set consists of the minterms {7, 6, 0, 1}, and so the off-set consists of the minterms {2, 3, 4, 5}.

Suppose the heuristic randomly chooses to expand term `xyz`. The heuristic may try to expand `xyz` to `xy`. Is that a legal expansion? `xy` covers minterms `xyz'` (minterm 6) and `xyz` (minterm 7), both in the on-set. Thus, the expansion is legal, so the heuristic replaces `xyz` by `xy`, yielding the new equation

```
F = xy + xyz' + x'y'z' + x'y'z
```

The heuristic then looks for implicants covered by the new implicant `xy`. `xyz'` is covered by `xy`, so the heuristic eliminates `xyz'`, yielding

```
F = xy + x'y'z' + x'y'z
```

The heuristic may continue trying to expand that first term. It may try expanding the term from `xy` to `x`. The term `x` covers minterms `xy'z'` (minterm 4), `xy'z` (minterm 5), `xyz'` (minterm 6), and `xyz` (minterm 7). The term `x` thus covers minterms 4 and 5, which are not in the on-set but instead are in the off-set. That expansion is not legal. The heuristic may try expanding `xy` to `y`, but again finds that expansion is not legal.

The heuristic might then try to expand next term, `x'y'z'`, to `x'y'`. That expanded term covers minterms `x'y'z'` (minterm 0) and `x'y'z` (minterm 1), both in the on-set, so the expansion is legal. The heuristic thus replaces the original term by the expanded term

```
F = xy + x'y' + x'y'z
```

The heuristic searches for other terms covered by the expanded term, and finds that `x'y'z` is covered by `x'y'`, so removes `x'y'z` to leave

```
F = xy + x'y'
```

The heuristic may try expanding the term `x'y'` further, but will find that neither possible expansion (`x'`, or `y'`) is legal. Thus, the above equation represents the minimized equation found by the heuristic. Notice that this happens to be the same result as obtained when minimizing the same initial equation in Example 6.4.

In the previous example, even though the heuristic generated the optimally minimized equation, there is no guarantee that the results from the heuristic will always be optimal.

More advanced heuristics utilize additional operations beyond just the expand operation. One such operation is the reduce operation, which can be thought of as the opposite of expand. The ***reduce*** operation adds a literal to a given term. A reduction is legal if the equation with the new term still covers the function. Adding a literal to a term is like reducing the size of a circle on a K-map. Adding a literal to a term reduces the number of minterms covered by the term, hence the name *reduce*. Another operation is ***irredundant***, which removes a term entirely as long as the new equation still covers the function. If so, the removed term was "redundant," hence the name *irredundant*.

A heuristic may iterate among the expand, reduce, irredundant, and other operations, such as in the following heuristic: Try 10 random expansion operations, then 5 random reduce operations, then 2 irredundant operations, and then repeat (iterate) the whole sequence until no improvement occurs from one iteration to the next. Modern two-level size optimization tools differ mainly in their ordering of operations and number of iterations.

Recall that this section stated that modern heuristics don't enumerate all the minterms of a function's on-set, yet the previous example did enumerate all those minterms—actually, the on-set minterms were given in the initial equation. When the on-set minterms are not known, many advanced methods exist to efficiently represent a function's on-set and off-set without enumerating the minterms in those sets, and also to quickly check if a term covers terms in the off-set. Those methods are beyond the scope of the book, and are instead the subject of textbooks on digital design synthesis.

One of the original tools that performed automated heuristics as well as exact two-level logic optimization was called ***Espresso***, developed at the University of California, Berkeley. The algorithms and heuristics in Espresso formed the basis of many modern commercial logic optimization tools.

Multilevel Logic Optimization—Performance and Size Tradeoffs

The previous sections discussed two-level logic size optimization. In practice, the speed of two levels of logic may not be necessary. Three, four, or more levels of logic may be acceptable if those additional levels reduce the amount of required logic. As a simple example, consider the equation

```
F1 = ab + acd + ace
```

This equation can't be minimized for two-level logic. The resulting two-level circuit is shown in Figure 6.33(a). However, algebraic manipulation can be done to yield

```
F2 = ab + ac(d + e) = a(b + c(d + e))
```

That equation can be implemented with the circuit shown in Figure 6.33(b). The multilevel logic implementation implements the same function but results in fewer transistors, at the expense of more gate-delays, as illustrated in Figure 6.33(c). The multilevel implementation thus represents a *tradeoff* when compared with the two-level implementation.

Automated heuristics for multilevel logic optimization iteratively transform the initial function's equation, similar to the iterative improvement used by automatic heuristics for two-level logic size optimization. The multilevel heuristics may optimize one of the criteria (size or delay), possibly at the expense of the other criteria.

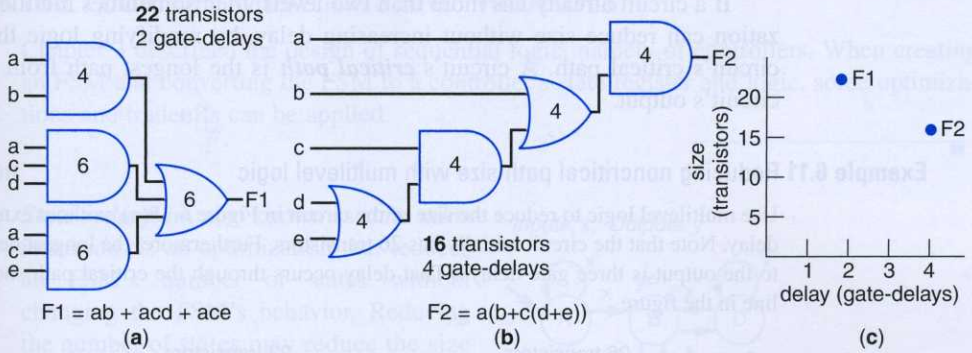


Figure 6.33 Using multilevel logic to tradeoff performance and size: (a) a two-level circuit, (b) multilevel circuit with fewer transistors but more delay, (c) illustration of the size versus delay tradeoff. Numbers inside gates represent transistor counts, estimated as 2 transistors per gate input.

Example 6.10 Multilevel logic optimization

Minimize the following function's circuit size, at the possible expense of increased delay, by algebraically manipulating the initial equation. Plot the tradeoff of the initial and size-optimized circuits with respect to size and delay.

```
F1 = abcd + abcef
```

The circuit corresponding to the equation is shown in Figure 6.34(a). The circuit requires 22 transistors and has a delay of 2 gate-delays.

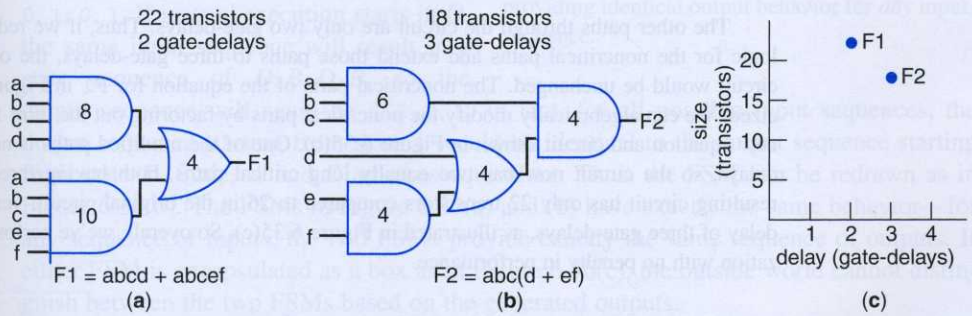


Figure 6.34 Multilevel logic to trade off performance and size: (a) two-level circuit, (b) multilevel circuit with fewer transistors, (c) tradeoff of size versus delay. Numbers inside gates represent transistor counts.

We can algebraically manipulate the equation by factoring out the `abc` term from the two terms, as follows:

```
F2 = abcd + abcef = abc(d + ef)
```

The circuit for that equation is shown in Figure 6.34(b). The circuit requires only 18 transistors, but has a longer delay of 3 gate-delays. The plot in Figure 6.34(c) shows the size and performance for each design.

If a circuit already has more than two levels, then sometimes multilevel logic optimization can reduce size without increasing delay, by modifying logic that exists off the circuit's critical path. A circuit's ***critical path*** is the longest path from any input to the circuit's output.

Example 6.11 Reducing noncritical path size with multilevel logic

Use multilevel logic to reduce the size of the circuit in Figure 6.35(a), without extending the circuit's delay. Note that the circuit initially has 26 transistors. Furthermore, the longest delay from any input to the output is three gate-delays. That delay occurs through the critical path shown by the dashed line in the figure.

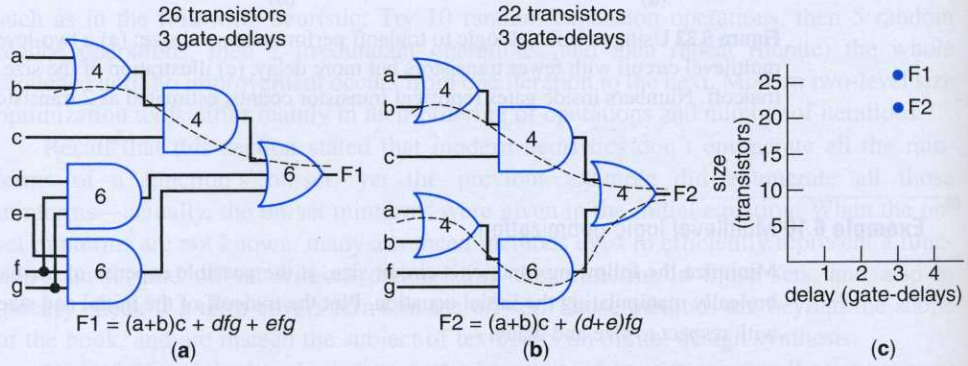


Figure 6.35 Multilevel optimization that reduces size without increasing delay, by altering logic on a noncritical path: (a) original circuit, (b) new circuit with fewer transistors but same delay, (c) illustration of the size optimization with no tradeoff of delay.

The other paths through the circuit are only two gate-delays. Thus, if we reduce the size of the logic for the noncritical paths and extend those paths to three gate-delays, the overall delay of the circuit would be unchanged. The noncritical parts of the equation for `F1` in Figure 6.35(a) are italicized. We can algebraically modify the noncritical parts by factoring out the term `fg`, resulting in the new equation and circuit shown in Figure 6.35(b). One of the modified paths is now also three gate-delays, so the circuit now has two equally long critical paths, both having three gate-delays. The resulting circuit has only 22 transistors compared to 26 in the original circuit, yet still has the same delay of three gate-delays, as illustrated in Figure 6.35(c). So overall, we've performed a size optimization with no penalty in performance.

Generally, multilevel logic optimization makes use of ***factoring***, such as `abc + abd = ab(c+d)`, to reduce the number of gates.

Multilevel logic optimization is probably more commonly used by modern tools than two-level logic optimization. Multilevel logic optimization is also extensively used by automatic tools that map circuits to FPGAs. FPGAs are discussed in Chapter 7.

6.3 SEQUENTIAL LOGIC OPTIMIZATIONS AND TRADEOFFS

Chapter 3 described the design of sequential logic, namely of controllers. When creating an FSM and converting the FSM to a controller's state register and logic, some optimizations and tradeoffs can be applied.

State Reduction

State reduction, also known as ***state minimization***, is an optimization that reduces an FSM's number of states without changing the FSM's behavior. Reducing the number of states may reduce the size of the state register and combinational logic that implement the FSM.

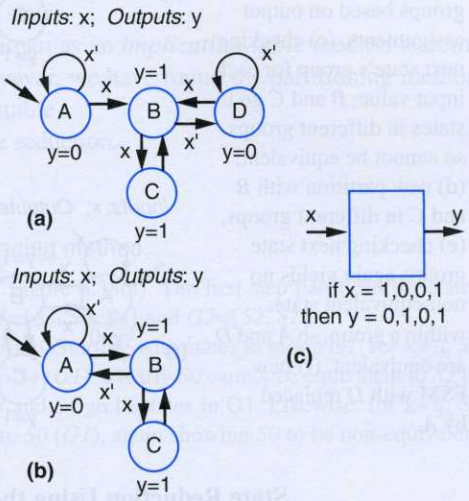


Figure 6.36 Equivalent FSMs: (a) original FSM, (b) equivalent FSM with fewer states, (c) the FSMs are indistinguishable from the outside, providing identical output behavior for *any* input sequence.

Reducing the number of states is possible when the FSM contains states that are equivalent to one another. Consider the FSM of Figure 6.36(a), having input `x` and output `y`. States *A* and *D* are equivalent. Regardless of whether the present state is *A* or *D*, the outputs will be identical for any sequence of inputs. For example, if the present state is *A* and the input sequence for four clock edges is `1, 0, 0, 1`, the state sequence will be *A*, *B*, *D*, *B*, so the output sequence will be `0, 1, 0, 1`. If instead execution starts in *D*, the same input sequence will result in a state sequence of *D*, *B*, *D*, *B*, so the output sequence will again be `0, 1, 0, 1`. In fact, for all possible input sequences, the output sequence starting from state *A* would be identical to the output sequence starting from state *D*. States *A* and *D* are thus equivalent. Thus, the FSM can be redrawn as in Figure 6.36(b). The FSMs in Figure 6.36(a) and (b) have exactly the same behavior—for any sequence of inputs, the two FSMs provide exactly the same sequence of outputs. If either FSM is encapsulated as a box as in Figure 6.36(c), the outside world cannot distinguish between the two FSMs based on the generated outputs.

Two states are equivalent if

1. both states assign the same values to outputs, AND
2. for all possible sequences of inputs, the FSM outputs will be the same starting from either state.

If two states are equivalent, then one of the states can be removed from the FSM, and any transitions pointing to that state can instead point to the other state.

For large FSMs, visual inspection to detect equivalent states is not feasible—a more systematic and automatable approach is needed.

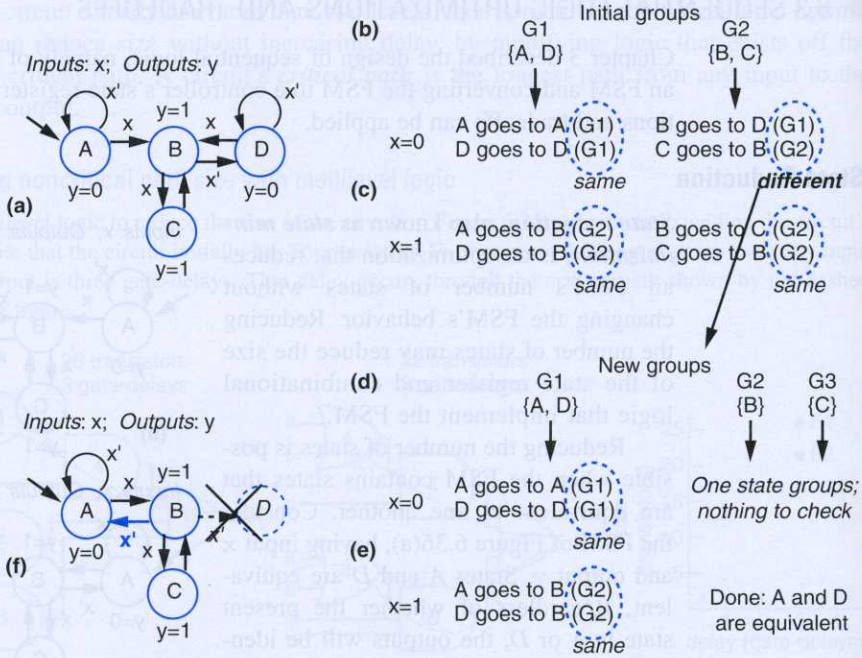


Figure 6.37 The partitioning method for state minimization: (a) original FSM, (b) initial groups based on output assignments, (c) checking next state's group for each input value; *B* and *C* go to states in different groups so cannot be equivalent, (d) new partition with *B* and *C* in different groups, (e) checking next state groups again yields no non-equivalent states within a group, so *A* and *D* are equivalent, (f) new FSM with *D* replaced by *A*.

State Reduction Using the Partitioning Method

A ***partitioning method*** can be used to find groups of equivalent states in an FSM. The method maintains a set of groups where states in different groups *cannot* be equivalent, whereas states in each group *might* be equivalent.

The first step of the method is to partition states into groups based on the values they assign to outputs—states assigning the same values are placed into the same group. For the FSM of Figure 6.37(a), the initial groups are *G1*: {*A*, *D*} (because *A* and *D* each outputs `y = 0`) and *G2*: {*B*, *C*} (because each outputs `y = 1`) as shown in Figure 6.37(b).

The next step involves checking next states. The states within a group are examined. For each possible input value, the next state is listed for each state. If for the same input value, two states in a group go to states in different groups, then those two states cannot possibly be equivalent. Figure 6.37(c) starts with group *G1* (having states *A* and *D*) and shows that for `x = 0`, *A* goes to a state in group *G1*, and *D* goes to a state in *G1*; both go to states in the same group, *G1*. For `x = 1`, *A* goes to a state in *G2*, and *D* goes to a state in *G2*; both go to states in the same group, *G2*. Thus, *A* and *D* might still be equivalent. Figure 6.37(c) proceeds to examine states in group *G2* (having states *B* and *C*) and shows that for `x = 0`, *B* goes to a state in *G1* whereas *C* goes to a state in *G2*. Thus, *B* and *C* cannot possibly be equivalent, because for the same input value, they transition to states that have already been determined to not be equivalent.

Upon detecting that two states in a group are not equivalent, the method partitions the group into subgroups so that states determined to be non-equivalent are in separate groups, as in Figure 6.37(d). The step of checking next states then repeats, as shown in Figure 6.37(e). This time, the step does not find any non-equivalent states in a group. After such a pass through the step, the states within a group are known to be equivalent.

For a group of equivalent states, one of the states can be selected to remain in the FSM. The other states and their outgoing transitions can be removed, Any transitions pointing to those removed states can be redirected to point to the remaining state. Figure 6.37(f) shows that the transition with condition `x'` pointing to state *D* is redirected to point to state *A*. Note that the transitions leaving a removed state need not be replaced; they are guaranteed to already exist on the remaining state, because state equivalence means that the remaining state has the same next state for each input value (i.e., the state has the same transitions) as did the removed state.

An alternative state reduction method known as an ***implication table*** method is commonly found in digital design textbooks. However, we have found the partitioning method to be more intuitive while also being automatable.

We now provide another example of state reduction.

Example 6.12 Minimizing states in an FSM using the partitioning method

This example minimizes the states in the FSM of Figure 6.38(a). The first step partitions the states according to their output assignments, yielding *G1*={*S3*, *S0*, *S4*} and *G2*={*S2*, *S1*}.

The next step examines next states for each group. Starting with states in group *G1*, for `x=0`, *S3* goes to *S3* (*G1*), *S0* goes to *S2* (*G2*), and *S4* goes to *S4* (*G1*). Clearly *S0* cannot be equivalent to *S3* or *S4*, because *S0* goes to a state in *G2*, while the *S3* and *S4* go to states in G1. Likewise, for `x=1`, *S3* goes to *S0* (*G1*), *S0* goes to *S1* (*G2*), and *S4* goes to *S0* (*G1*), again showing *S0* to be non-equivalent to *S3* and *S4*.

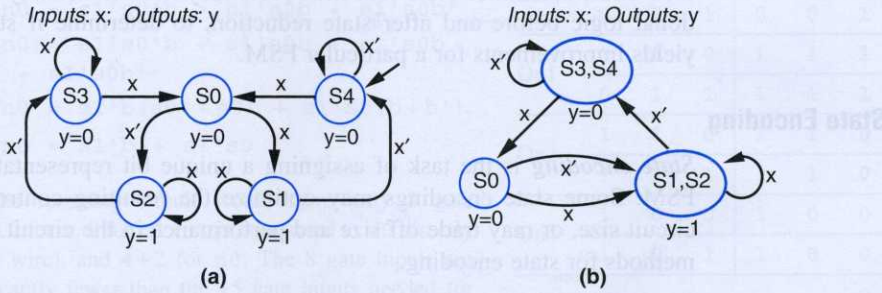


Figure 6.38 FSM state reduction example: (a) original FSM, (b) reduced FSM.

Thus, the method partitions *G1* to yield new groups: *G1*={*S3*,*S4*}, *G2*={*S2*,*S1*}, and *G3*={*S0*}. The method then repeats the step that examines next states. For group *G1*, for `x=0`, *S3* goes to *S3* (*G1*), and *S4* goes to *S4* (*G1*). For `x=1`, *S3* goes to *S0* (*G3*), and *S4* goes to *S0* (*G3*). Thus, *S3* and *S4* still might be equivalent. For group *G2*, for `x=0`, *S2* goes to *S3* (*G1*), and *S1* goes to *S4* (*G1*). For `x=1`, *S2* goes to *S2* (*G2*), and *S1* goes to *S1* (*G2*). Thus, *S1* and *S2* might still be equivalent. Group *G3* has only one state so there is nothing to examine. Thus, this pass through the next state step found no non-equivalent states. Therefore, states *S3* and *S4* (which are in group *G1*) are equivalent, and states *S2* and *S1* (which are in *G2*) are equivalent. The resulting FSM is shown in Figure 6.38(b).

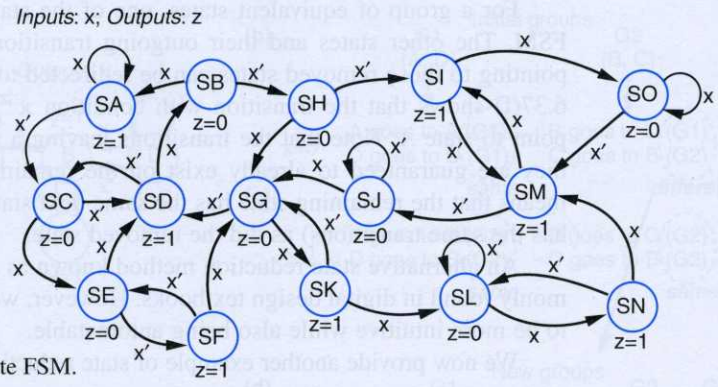


Figure 6.39 A 15-state FSM.

State reduction is typically performed using automated tools. For smaller FSMs, the tools may implement a method similar to the partitioning method. For larger FSMs, the tools may need to resort to heuristics to avoid inordinately large numbers of state comparisons.

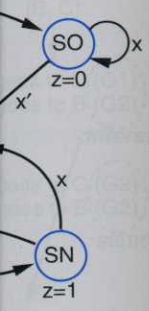
Reducing the number of states does not guarantee a reduction of size of the resulting circuit. One reason is because reducing the states might not reduce the number of required state register bits—reducing the states from 15 down to 12 does not reduce the minimum state register size, which is four in either case. Another reason is because, even if the state reduction reduces the state register size, the combinational logic size could possibly *increase* with a smaller state register, due to the logic having to decode the state bits. Thus, automated state reduction tools may need to actually implement the combinational logic before and after state reduction, to determine if state reduction ultimately yields improvements for a particular FSM.

State Encoding

State encoding is the task of assigning a unique bit representation for each state in an FSM. Some state encodings may optimize the resulting controller circuit by reducing circuit size, or may trade off size and performance in the circuit. We now discuss several methods for state encoding.

Alternative Minimum-Bitwidth Binary Encodings

Previously, we assigned a unique binary encoding to each state in an FSM using the fewest number of bits possible, representing a ***minimum-bitwidth binary encoding***. If there were four states, we used two bits. If there were five, six, seven, or eight states, we used three bits. The encoding represented the state in the controller's state register. There are many ways to map minimum-bitwidth binary encodings to a set of states. Say we are given four states, *A*, *B*, *C*, and *D*. One encoding is *A:*`00`, *B:*`01`, *C:*`10`, *D:*`11`. Another encoding is *A:*`01`, *B:*`10`, *C:*`11`, *D:*`00`. In fact, there are 4*3*2*1 = 4! (four factorial) = 24 possible encodings into two bits (4 encoding choices for the first state, 3 for the next state, 2 for the next, and 1 for the last state). For eight states, there are 8!, or over 40,000, possible encodings into three bits. For *N* states, there are *N!* (*N* factorial) possible encodings—a huge number for any *N* greater than 10 or so. One encoding may result in less combinational logic than another encoding. Automated tools may try several different encodings (but not all *N!* encodings) to reduce combinational logic in the controller.



maller FSMs, the
larger FSMs, the
numbers of state

ze of the resulting
e the number of
oes not reduce the
n is because, even
l logic size could
o decode the state
ment the combina-
duction ultimately

r each state in an
ircuit by reducing
ow discuss several

an FSM using the
nary encoding. If
or eight states, we
tate register. There
states. Say we are
10, *D*:11. Another
! (four factorial) =
tate, 3 for the next
8!, or over 40,000,
al) possible encod-
may result in less
y several different
the controller.

Example 6.13 Alternative binary encoding for three-cycles-high laser timer

Example 3.7 encoded states using a straightforward binary encoding, starting with `00`, then `01`, then `10`, and then `11`. The resulting design had 15 gate inputs (ignoring inverters). We can try instead the alternative binary encoding shown in Figure 6.40.

Table 6.2 provides the truth table for the new encoding, bolding the differences from the original encoding.

The truth table yields the following equations for the three combinational logic outputs of a controller:

```
x = s1 + s0  (note from the table that
  x=1  if s1=1 or s0=1)

n1 = s1's0b' + s1's0b + s1s0b'
  + s1s0b
n1 = s1's0 + s1s0
n1 = s0

n0 = s1's0'b + s1's0b + s1's0b'
n0 = s1's0'b + s1's0b + s1's0b
  + s1's0b'
n0 = s1'b(s0'+s0) + s1's0(b+b')
n0 = s1'b + s1's0
```

The resulting circuit would have only 8 gate inputs: 2 for x, 0 for `n1` (`n1` is connected to `s0` directly with a wire), and 4+2 for `n0`. The 8 gate inputs are significantly fewer than the 15 gate inputs needed for the binary encoding of Example 3.7. This encoding reduces size without any increase in delay, thus representing an optimization.

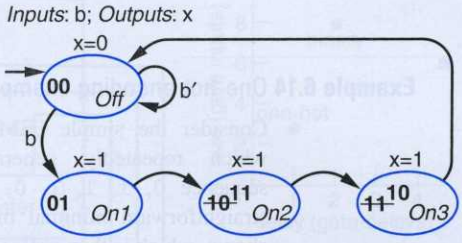


Figure 6.40 Laser timer state diagram with alternative binary state encoding.

TABLE 6.2 Truth table for laser timer controller with alternative encoding.

	Inputs			Outputs		
	s1	s0	b	x	n1	n0
Off	0	0	0	0	0	0
	0	0	1	0	0	1
On1	0	1	0	1	1	**1**
	0	1	1	1	1	**1**
On2	1	**1**	0	1	1	**0**
	1	**1**	1	1	1	**0**
On3	1	**0**	0	1	0	0
	1	**0**	1	1	0	0

One-Hot Encoding

There is no requirement that a set of states be encoded using the fewest number of bits. For example, four states *A*, *B*, *C*, and *D* could be encoded using three bits instead of just two bits, such as *A*:`000`, *B*:`011`, *C*:`110`, *D*:`111`. Using more bits requires a larger state register, but possibly less logic. A popular encoding scheme is called ***one-hot encoding***, wherein the number of bits used for encoding equals the number of states, and each bit corresponds to exactly one state. For example, a one-hot encoding of four states *A*, *B*, *C*, and *D* uses four bits, such as *A*:`0001`, *B*:`0010`, *C*:`0100`, *D*:`1000`. The

main advantage of one-hot encoding is speed—because the state can be detected from just one bit and thus need not be decoded using an AND gate, the controller's next state and output logic may involve fewer gates and/or gates with fewer inputs, resulting in a shorter delay.

Example 6.14 One-hot encoding example

Consider the simple FSM of Figure 6.41, which repeatedly generates the output sequence 0, 1, 1, 1, 0, 1, 1, 1, etc. A straightforward minimal binary encoding is shown, which is then crossed out and replaced with a one-hot encoding.

The binary encoding results in the truth table shown in Table 6.3. The resulting equations are:

```
n1 = s1's0 + s1s0'

n0 = s0'

x = s1 + s0
```

The one-hot encoding results in the truth table shown in Table 6.4. The resulting equations are:

```
n3 = s2

n2 = s1

n1 = s0

n0 = s3

x = s3 + s2 + s1
```

Figure 6.42 shows the resulting circuits for each encoding. The binary encoding yields more gates but, more importantly, requires two levels of logic. The one-hot encoding in this example requires only one level of logic. Notice that the logic to generate the next state is just wires in this example (other examples may require some logic). Figure 6.42(c) illustrates that the one-hot encoding has less delay, meaning a faster clock frequency could be used for that circuit.

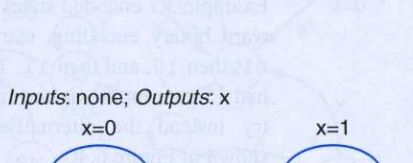


Figure 6.41 FSM for given sequence.

TABLE 6.3 Truth table using binary encoding.

	Inputs		Outputs		
	s1	s0	n1	n0	x
A	0	0	0	1	0
B	0	1	1	0	1
C	1	0	1	1	1
D	1	1	0	0	1

TABLE 6.4 Truth table using one-hot encoding.

	Inputs				Outputs				
	s3	s2	s1	s0	n3	n2	n1	n0	x
A	0	0	0	1	0	0	1	0	0
B	0	0	1	0	0	1	0	0	1
C	0	1	0	0	1	0	0	0	1
D	1	0	0	0	0	0	0	1	1

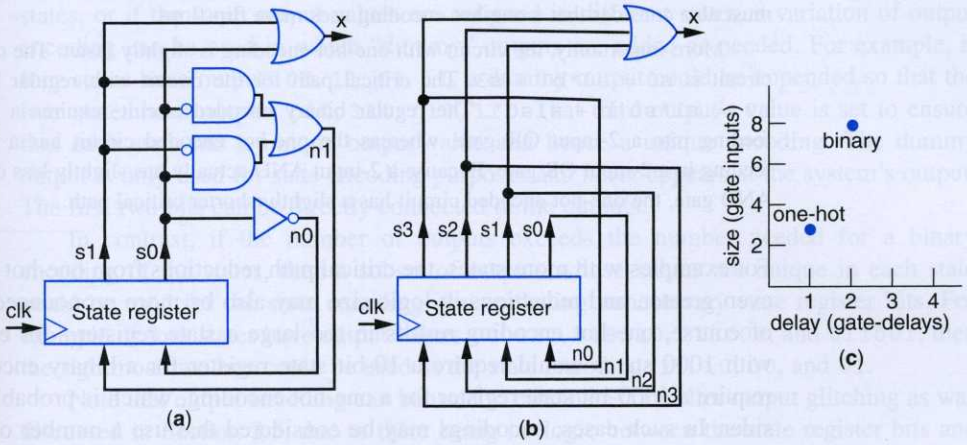


Figure 6.42 One-hot encoding can reduce delay: (a) minimum binary encoding, (b) one-hot encoding, (c) though total sizes may be roughly equal (one-hot encoding uses fewer gates but more flip-flops), one-hot yields a shorter critical path.

Example 6.15 Three-cycles-high laser timer using one-hot encoding

Example 3.7 encoded states using a straightforward binary encoding, starting with `00`, then `01`, then `10`, and then `11`. This example uses a one-hot encoding of the four states, requiring four bits as shown in Figure 6.43.

Table 6.6 shows a truth table for the FSM of Figure 6.43, using the one-hot encoding of the states. Not all rows are shown, since the table would then be too large.

The last step is to design the combinational logic. Deriving equations for each output directly from the table (assuming all other input combinations are don't cares), and minimizing those equations algebraically, results in the following:

```
x = s3 + s2 + s1
n3 = s2
n2 = s1
n1 = s0*b
n0 = s0*b' + s3
```

This circuit would require 3+0+0+2+(2+2) = 9 gate inputs.

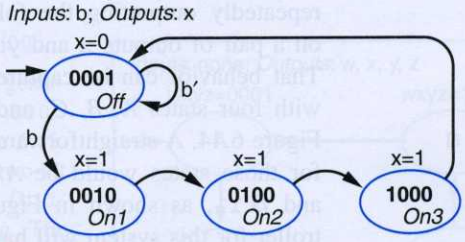


Figure 6.43 One-hot encoding of laser time.

TABLE 6.6 Truth table for laser timer controller with one-hot encoding.

	Inputs					Outputs				
	s3	s2	s1	s0	b	x	n3	n2	n1	n0
Off	0	0	0	1	0	0	0	0	0	1
	0	0	0	1	1	0	0	0	1	0
On1	0	0	1	0	0	1	0	1	0	0
	0	0	1	0	1	1	0	1	0	0
On2	0	1	0	0	0	1	1	0	0	0
	0	1	0	0	1	1	1	0	0	0
On3	1	0	0	0	0	1	0	0	0	1
	1	0	0	0	1	1	0	0	0	1

Thus, the circuit has fewer gate inputs than the original binary encoding's 15 gate inputs—but one must also consider that a one-hot encoding uses more flip-flops.

More importantly, the circuit with one-hot encoding is slightly faster. The critical path for that circuit is `n0 = s0*b' + s3`. The critical path for the circuit with regular binary encoding is `n0 = s1's0'b + s1s0'`. The regular binary encoded circuit requires a 3-input AND gate feeding into a 2-input OR gate, whereas the one-hot encoded circuit has a 2-input AND gate feeding in a 2-input OR gate. Because a 2-input AND actually has slightly less delay than a 3-input AND gate, the one-hot encoded circuit has a slightly shorter critical path.

For examples with more states, the critical path reductions from one-hot encoding may be even greater, and reductions in logic size may also be more pronounced. At some point, of course, one-hot encoding results in too large a state register—for example, an FSM with 1000 states would require a 10-bit state register for a binary encoding, but would require a 1000-bit state register for a one-hot encoding, which is probably too big to consider. In such cases, encodings may be considered that use a number of bits in between that for a binary encoding and that for a one-hot encoding.

Output Encoding

Output encoding uses the output values assigned in a state as the encoding for that state. For example, a problem might require repeatedly outputting the following sequence on a pair of outputs `x` and `y`: `00`, `11`, `10`, `01`. That behavior can be captured using an FSM with four states *A*, *B*, *C*, and *D*, as shown in Figure 6.44. A straightforward binary encoding for those states would be *A*:`00`, *B*:`01`, *C*:`10`, and *D*:`11`, as shown in Figure 6.44. A controller for this system will have a two-bit state register, logic to determine the next state, and logic to generate the output from the present state. In contrast, output encoding would simply use the output values of each state as the encoding of that state, meaning the encoding for the example would be *A*:`00`, *B*:`11`, *C*:`10`, and *D*:`01`. Such an encoding will still result in a two-bit state register and logic to generate the next state, but there won't be logic to generate the output from the present state.

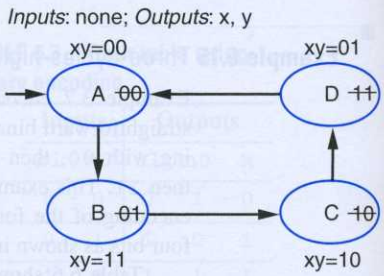


Figure 6.44 FSM for given sequence.

Output encoding may reduce the amount of logic by eliminating the logic that generates the outputs from the present state encoding—that logic is reduced to just wires.

Straightforward output encoding is possible if two conditions are satisfied:

1. The FSM has at least as many outputs as needed for a binary encoding of the states, and
2. The output values assigned by each state are unique.

For example, if the FSM in Figure 6.44 had only the one output `x`, then output encoding would not work because there are too few outputs, so condition 1 above is not satisfied. Or, if the four states had output values of `00`, `11`, `01`, `11`, output encoding would not work because two of the states have output values (i.e., `11`) that are not unique.

If the number of outputs is less than the number needed for a binary encoding of the states, or if the same output values are assigned in different states, a variation of output encoding can be used in which "dummy" outputs are added as needed. For example, if four states have outputs `00`, `11`, `01`, `11`, a dummy output could be appended so that the four states have outputs `000`, `110`, `010`, `111`. That third output's value is set to ensure that each state has a unique output value and hence a unique encoding. The dummy output is only used for state encoding purposes and never appears at the system's output. The first two bits can be directly connected to the outputs.

In contrast, if the number of outputs exceeds the number needed for a binary encoding of the states, then a subset of outputs whose values are unique in each state could be used for the state encoding, thus reducing unnecessary state register bits. For example, if four states have output values `000011`, `000110`, `110000`, and `011001`, then the rightmost two bits could be used as the state encoding: `11`, `10`, `00`, and `01`.

Note that output encoding can be used to eliminate controller output glitching as was discussed in Section 3.5, due to there being no logic between the state register bits and the outputs.

Even if output encoding is not fully used, using output encoding for as many states as possible may still serve to reduce logic. For example, four states with outputs `00`, `11`, `01`, `11` might encode states as `00`, `11`, `01`, and `10`—only the last state's encoding differs from the output.

Example 6.16 Sequence generator using output encoding

Example 3.10 involved design of a sequence generator for the sequence `0001`, `0011`, `1100`, `1000` on a set of four outputs, as shown in Figure 6.45. That example encoded the states using a two-bit binary encoding of *A*:`00`, *B*:`01`, *C*:`10`, and *D*:`11`. This example instead uses output encoding. The four outputs are more than the minimum of two bits needed to encode the four states. Each state's output values are also unique. Thus, output encoding can be considered for this example.

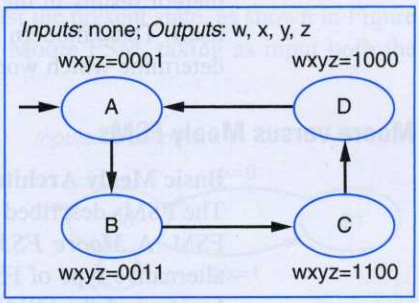


Figure 6.45 Sequence generator FSM.

Table 6.7 shows a partial truth table for the sequence generator using output encoding. Notice that the outputs themselves w, x, y, and z don't need to appear in the table, as they will be the same as `s3`, `s2`, `s1`, and `s0`. We use a partial table to avoid having to show all 16 rows, and we assume that all unspecified rows represent don't cares. The table leads to the following equations for each output:

```
n3 = s1 + s2
n2 = s1
n1 = s1's0
n0 = s1's0 + s3s2'
```

TABLE 6.7 Partial truth table for sequence generator controller using output encoding.

	Inputs				Outputs			
	s3	s2	s1	s0	n3	n2	n1	n0
A	0	0	0	1	0	0	1	1
B	0	0	1	1	1	1	0	0
C	1	1	0	0	1	0	0	0
D	1	0	0	0	0	0	0	1

We obtained those equations by looking at all the `1`s for a particular output, and visually determining a minimal input equation that would generate those `1`s and `0`s for the other shown column entries (all other output values, not shown, are don't cares).

Figure 6.46 shows the final circuit. Notice that there is no output logic—the outputs w, x, y, and z connect directly to the state register.

Compared to the circuit obtained in Example 3.10 using binary encoding, the output encoded circuit in Figure 6.46 actually appears to use more transistors, due to using a wider state register. In other examples, an output encoded circuit might use fewer transistors.

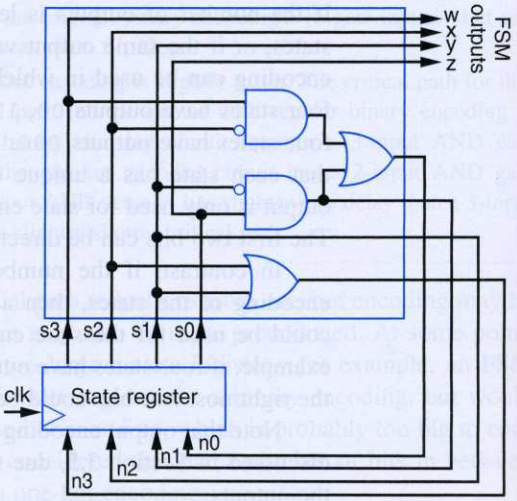


Figure 6.46 Sequence generator controller with output encoding.

Whether one-hot encoding, binary encoding, output encoding, or some variation thereof results in the fewest transistors or a shorter critical path depends on the example itself. Thus, modern tools may try a variety of different encodings for a given problem to determine which works best.

Moore versus Mealy FSMs

Basic Mealy Architecture

The FSMs described in this book have thus far all been a type of FSM known as a Moore FSM. A ***Moore FSM*** is an FSM whose outputs are a function of the FSM's state. An alternative type of FSM is a Mealy FSM. A ***Mealy FSM*** is an FSM whose outputs are a function of the FSM's states *and inputs*. Sometimes a Mealy FSM results in fewer states than a Moore FSM, representing an optimization. Sometimes those fewer states come at the expense of timing complexities that must be handled, representing a tradeoff.

Recall the standard controller architecture of Figure 3.60, reproduced in Figure 6.47. The architecture shows one block of combinational logic, responsible for converting the present state and external inputs into the next state and external outputs.

Because a Moore FSM's outputs are solely a function of the present state (and not of the external inputs), then the architecture can be refined to have two combinational logic blocks: the ***next-state***

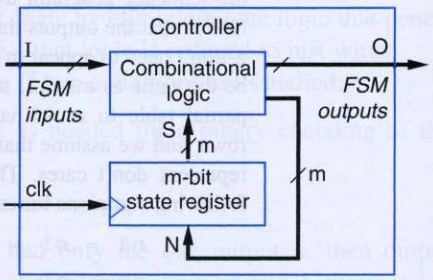


Figure 6.47 Standard controller architecture—general view.

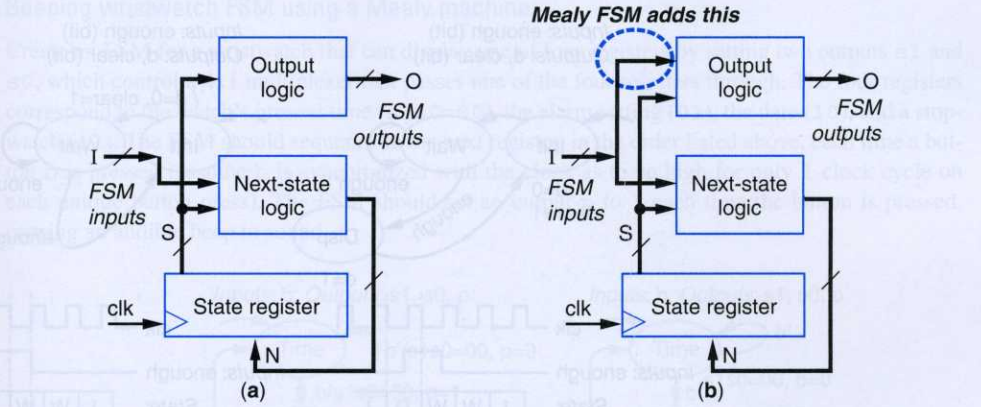


Figure 6.48 Controller architectures for: (a) a Moore FSM, (b) a Mealy FSM.

logic block converts the present state and external inputs into a next state, and the ***output logic*** block converts the present state (but *not* the external inputs) into external outputs, as shown in Figure 6.48(a).

In contrast, a Mealy FSM's outputs are a function of both the present state and the external inputs. Thus, the output logic block for a Mealy FSM takes both the present state *and* the external FSM inputs as input, rather than just the present state, as shown in Figure 6.48(b). The next-stage logic is the same as for a Moore FSM, taking as input both the present state and the external FSM inputs.

Graphically, the FSM output assignments of a Mealy FSM would be listed with each transition, rather than each state, because each transition represents a present state and a particular input value. Figure 6.49 shows a two-state Mealy FSM with an input `b` and an output `x`. When in state *S0* and `b = 0`, the FSM outputs `x = 0` and stays in state *S0*, as indicated by the transition labeled "`b'/x=0`". When in state *S0* and `b = 1`, the FSM outputs `x = 1` and goes to state *S1*. The "/" is used simply to separate the transition's input conditions from the output assignments—the "/" does not mean "divide" here. Because the transition from *S1* to *S0* is taken no matter what the input value, the transition is listed simply as "`/x=0`," meaning there is no input condition, but there is an output assignment.

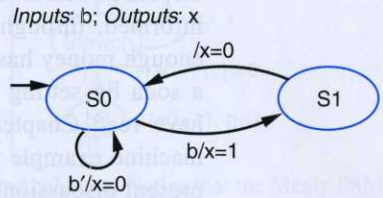


Figure 6.49 A Mealy FSM associates outputs with transitions, not with states.

Mealy FSMs May Have Fewer States

The seemingly minor difference between a Mealy and a Moore FSM, namely that a Mealy FSM's output is a function of the state *and* the current inputs, can lead to fewer states for some behaviors when captured as a Mealy FSM. For example, consider the simple soda dispenser controller FSM in Figure 6.50(a). Setting `d = 1` dispenses a

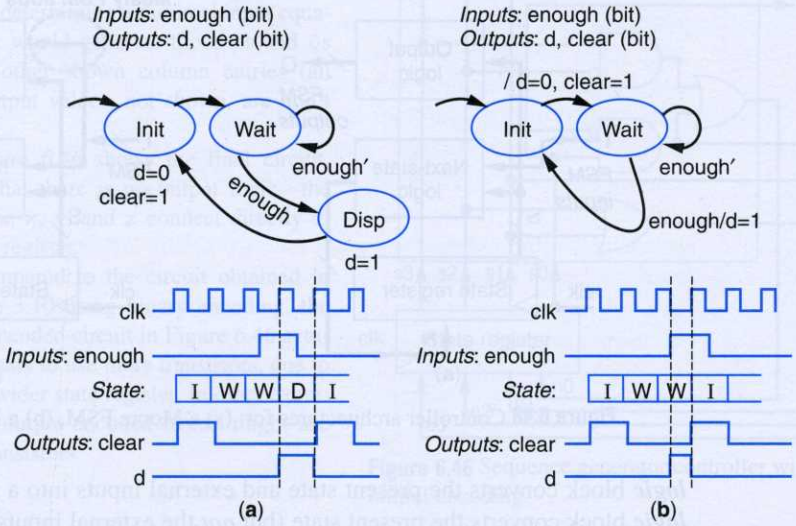


Figure 6.50 FSMs for the soda dispenser controller: (a) Moore FSM has actions in states, (b) Mealy FSM has actions on transitions, resulting in fewer states for this example.

soda. The FSM starts in state *Init*, which sets `d = 0` and sets an output `clear = 1` which clears a device keeping count of the amount of money deposited into the soda dispenser machine. The FSM transitions to state *Wait*, where the FSM waits to be informed, through the `enough` input, that enough money has been deposited. Once enough money has been deposited, the FSM transitions to state *Disp*, which dispenses a soda by setting output `d=1`, and the FSM then returns to state *Init*. (Readers who have read Chapter 5 may notice this example is a simplified version of the soda machine example in Section 5.2; familiarity with that example is not required for the present discussion.).

Like with Moore FSMs, we follow the convention that unassigned outputs in a Mealy FSM state diagram are implicitly assigned `0`.

Figure 6.50(b) shows a Mealy FSM for the same controller. The initial state *Init* has no actions itself, but rather has a conditionless transition to state *Wait* that has the initialization actions `d = 0` and `clear = 1`. In state *Wait*, a transition with condition `enough'` returns to state *Wait* without any actions listed. Another transition with condition `enough` has the action `d = 1`, and takes the FSM back to the *Init* state. Notice that the Mealy FSM does not need the *Disp* state to set `d = 1`; that action occurs on a transition. Thus, the Mealy FSM has fewer states than the Moore FSM for this example.

The Mealy state diagram in Figure 6.50(b) uses a convention similar to the convention used for Moore FSMs in Section 3.4, namely that any outputs not explicitly assigned on a transition are implicitly assigned a `0`. As with Moore FSMs, we still set an output to `0` explicitly if the assignment is key to the FSM's behavior (such as the assignment of `d = 0` in Figure 6.50(b)).

Example 6.17 Beeping wristwatch FSM using a Mealy machine

Create an FSM for a wristwatch that can display one of four registers by setting two outputs `s1` and `s0`, which control a 4x1 multiplexer that passes one of the four registers through. The four registers correspond to the watch's present time (`s1s0=00`), the alarm setting (`01`), the date (`10`), and a stopwatch (`11`). The FSM should sequence to the next register, in the order listed above, each time a button `b` is pressed (assume `b` is synchronized with the clock as to be high for only `1` clock cycle on each unique button press). The FSM should set an output `p` to `1` each time the button is pressed, causing an audible beep to sound.

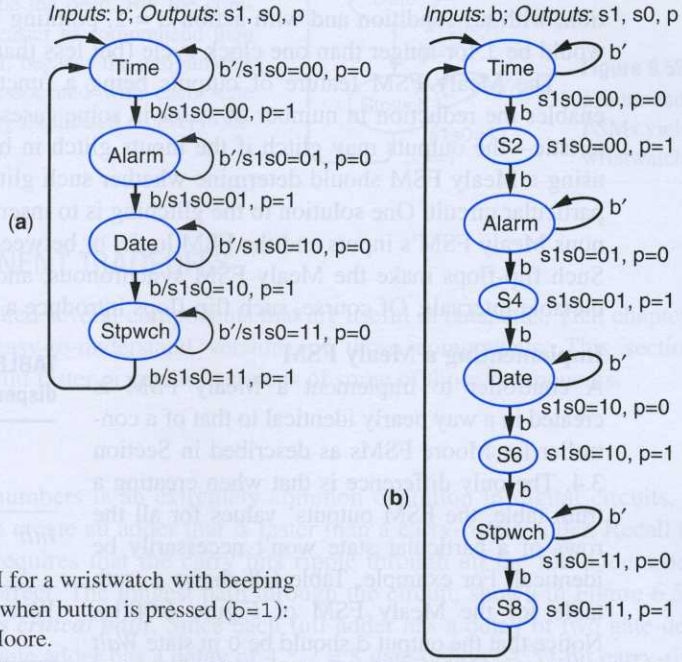


Figure 6.51 FSM for a wristwatch with beeping behavior (`p=1`) when button is pressed (`b=1`): (a) Mealy, (b) Moore.

Figure 6.51(a) shows a Mealy FSM describing the desired behavior. Notice that the Mealy FSM easily captures the beeping behavior, simply by setting `p = 1` on the transitions that correspond to button presses. In the Moore FSM of Figure 6.51(b), we had to add an extra state in between each pair of states in Figure 6.51, with each extra state having the action `p=1` and having a conditionless transition to the next state.

Notice that the Mealy FSM has fewer states than the Moore machine. A drawback is that we aren't guaranteed that a beep will last at least one clock cycle, due to timing issues that we will describe.

Timing Issues with Mealy FSMs

Mealy FSM outputs are not synchronized with clock edges, but rather can change in between clock edges if an input changes. For example, consider the timing diagram shown in Figure 6.50(a) for a soda dispenser's Moore FSM. Note that the output `d` becomes `1` *not right after* the input `enough` became `1`, but rather *on the first clock edge after* `enough` became `1`. In contrast, the timing diagram for the Mealy FSM in Figure

6.50(b) shows that the output d becomes 1 *right after* the input enough becomes 1. Moore outputs are synchronized with the clock; in particular, Moore outputs only change upon entering a new state, which means Moore outputs only change slightly after a rising clock edge loads a new state into the state register. In contrast, Mealy outputs can change not just upon entering a new state, but also at any time that an input changes, because Mealy outputs are a function of both the state and the inputs. We took advantage of this fact to eliminate the *Disp* state from the soda dispenser's Mealy FSM in Figure 6.50(b). Notice in the timing diagram, however, that the d output of the Mealy FSM *does not stay 1 for a complete clock cycle*. If we are unsure as to whether d's high time is long enough, we could include a *Disp* state in the Mealy FSM. That state would have a single transition, with no condition and with action d = 1, pointing back to state *Init*. In that case, d would be 1 for longer than one clock cycle (but less than two cycles).

The Mealy FSM feature of outputs being a function of states and inputs, which enables the reduction in number of states in some cases, also has an undesirable characteristic—the outputs may glitch if the inputs glitch in between clock cycles. A designer using a Mealy FSM should determine whether such glitching could pose a problem in a particular circuit. One solution to the glitching is to insert flip-flops between an asynchronous Mealy FSM's inputs and the FSM logic, or between the FSM logic and the outputs. Such flip-flops make the Mealy FSM synchronous, and the outputs will change at predictable intervals. Of course, such flip-flops introduce a one clock cycle delay.

Implementing a Mealy FSM

A controller to implement a Mealy FSM is created in a way nearly identical to that of a controller for Moore FSMs as described in Section 3.4. The only difference is that when creating a truth table, the FSM outputs' values for all the rows of a particular state won't necessarily be identical. For example, Table 6.8 shows a truth table for the Mealy FSM of Figure 6.50(b). Notice that the output d should be 0 in state *Wait* (s0 = 1) if enough = 0, but should be 1 if enough = 1. In contrast, in a Moore truth table, an output's values are identical within a given state. Given the truth table of Table 6.8, implementing the combinational logic would proceed in the same manner described in Section 3.4.

TABLE 6.8 Mealy truth table for soda dispenser

	Inputs		Outputs		
	s0	enough	n0	d	clear
Init	0	0	1	0	1
	0	1	1	0	1
Wait	1	0	1	0	0
	1	1	0	1	0

Viewing the two "o's" in the word Moore as states may help you remember that a Moore FSM's actions occur in the states, while Mealy is on the transitions.

Combining Moore and Mealy FSMs

Designers often utilize FSMs that are a combination of Moore and Mealy types. Such a combination allows the designer to specify some actions in states, and others on transitions. Such a combination provides the reduced number of states advantage of a Mealy FSM, yet avoids having to replicate a state's actions on every outgoing transition of a state. This simplification is really just a convenience to a designer describing the FSM; the underlying implementation will be the same as for the Mealy FSM having replicated actions on a state's outgoing transitions.

Example 6.18 Beeping wristwatch FSM using a combined Moore/Mealy machine

Figure 6.52 shows a combined Moore/Mealy FSM describing the beeping wristwatch of Example 6.17. The FSM has the same number of states as the Mealy FSM in Figure 6.51(a), because the FSM still associates the beep behavior `p=1` with transitions, avoiding the need for extra states to describe the beep. But the combined FSM is easier to comprehend than the Mealy FSM, because the assignments to `s1s0` are associated with each state rather than being duplicated on every outgoing transition.

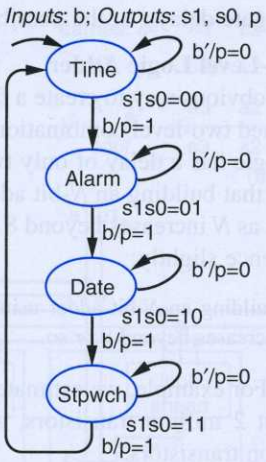


Figure 6.52 Combining Moore and Mealy FSMs yields a simpler wristwatch FSM.

6.4 DATAPATH COMPONENT TRADEOFFS

Chapter 4 created several components that are useful in datapaths. That chapter created the most basic, easy-to-understand versions of those components. This section describes methods to build faster or smaller versions of some of those components.

Faster Adders

Adding two numbers is an extremely common operation in digital circuits, so it makes sense to try to create an adder that is faster than a carry-ripple adder. Recall that a carry-ripple adder requires that the carry bits ripple through all the full-adders before all the outputs are correct. The longest path through the circuit, shown in Figure 6.53, is known as the circuit's ***critical path***. Since each full-adder has a delay of two gate-delays, then a 4-bit carry-ripple adder has a delay of 4 * 2 = 8 gate-delays. A 32-bit carry-ripple adder's delay is 32 * 2 = 64 gate-delays. That's rather slow, but the nice thing about a carry-ripple adder is that it doesn't require very many gates. If a full-adder uses 5 gates, then a 4-bit carry-ripple adder requires only 4 * 5 = 20 gates, and a 32-bit carry-ripple adder would only require 32 * 5 = 160 gates.

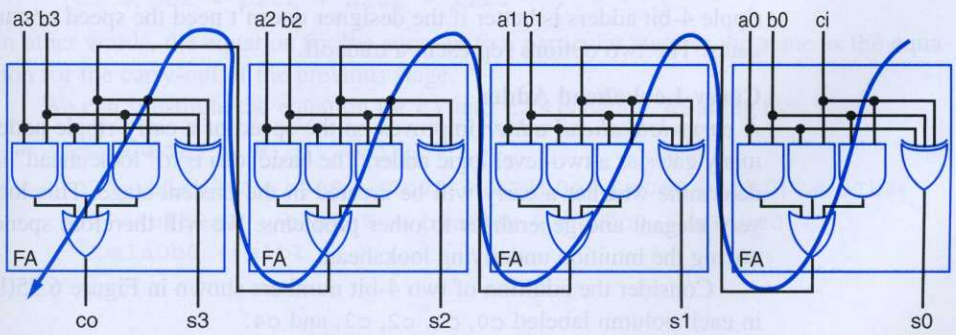


Figure 6.53 4-bit carry-ripple adder, with the longest path (the critical path) shown.

A useful adder would be an adder whose delay is much closer to the delay of just 5 or 6 gate-delays, at the possible expense of more total gates.

Two-Level Logic Adder

One obvious way to create a faster adder at the expense of more gates is to use the earlier-defined two-level combinational logic design process. An adder designed using two levels of logic has a delay of only two gate-delays. That's certainly fast. But recall from Figure 4.24 that building an *N*-bit adder using two levels of logic results in excessively large circuits as *N* increases beyond 8 or so. To be sure you get this point, let's restate the previous sentence slightly:

> Building an *N*-bit adder using two levels of logic results in *shockingly large circuits* as *N* increases beyond 8 or so.

For example, we estimated (in Chapter 4) that a two-level 16-bit adder would require about 2 million transistors, and that a two-level 32-bit adder would require about 100 billion transistors.

On the other hand, building a 4-bit adder using two levels of logic results in a big but reasonably sized adder—about 100 gates, as was shown in Figure 4.25. A larger adder could be built by cascading such fast 4-bit adders together. An 8-bit adder could be built by cascading two fast 4-bit adders together, as shown in Figure 6.54. If each 4-bit adder is built from two levels of logic, then each 4-bit adder has a delay of 2 gate-delays. The 4-bit adder on the right takes 2 gate-delays to generate the sum and carry-out bits, after which the 4-bit adder on the left takes another 2 gate-delays to generate its outputs, resulting in a total delay of 2 + 2 = 4 gate-delays. For a 32-bit adder built from eight 4-bit adders, the delay would be 8 * 2 = 16 gate-delays, and the size would be about 8 * 100 gates = 800 gates. That's much better than the 32 * 2 = 64 gate-delays of a carry-ripple adder, though the improved speed comes at the expense of more gates than the 32 * 5 = 160 gates of the carry-ripple adder. Which design is better? The answer depends on a designer's requirements—the design using two-level logic 4-bit adders is better if the designer needs more speed and can afford the extra gates, whereas the design using carry-ripple 4-bit adders is better if the designer doesn't need the speed or can't afford the extra gates. The two options represent a tradeoff.

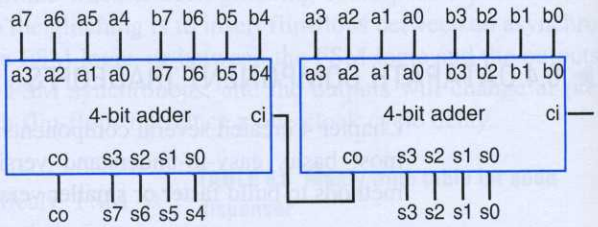


Figure 6.54 8-bit adder built from two fast 4-bit adders.

Carry-Lookahead Adder

A ***carry-lookahead adder*** improves on the speed of a carry-ripple adder without using as many gates as a two-level logic adder. The basic idea is to "look ahead" into lower stages to determine whether a carry will be created in the present stage. This lookahead concept is very elegant and generalizes to other problems. We will therefore spend some time introducing the intuition underlying lookahead.

Consider the addition of two 4-bit numbers shown in Figure 6.55(b), with the carries in each column labeled `c0`, `c1`, `c2`, `c3`, and `c4`.

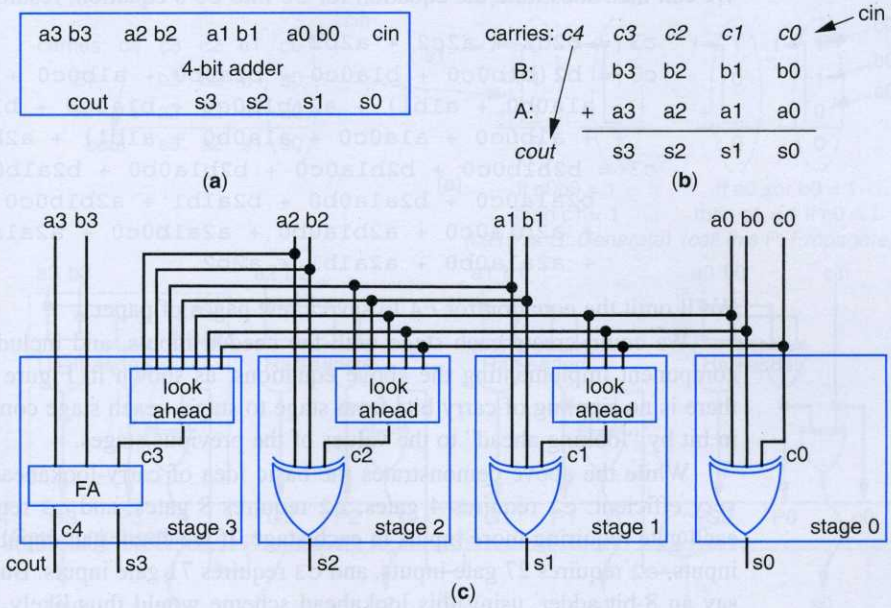


Figure 6.55 Adding two binary numbers by a naive inefficient carry-lookahead scheme—each stage looks at all earlier bits and computes whether the carry-in bit to that stage would be a 1. The longest delay is stage 3, which has 2 logic levels for the lookahead, and 2 logic levels for the full-adder, for a total delay of only four gate-delays.

A Naive Inefficient Carry-Lookahead Scheme. One simple but inefficient carry-lookahead approach is as follows. Recall that the output equations for a full-adder having inputs a, b, and c, and outputs co and s, are

```
s = a xor b xor c
co = bc + ac + ab
```

So we know that the equations for c1, c2, and c3 in a 4-bit adder will be

```
c1 = co0 = b0c0 + a0c0 + a0b0
c2 = co1 = b1c1 + a1c1 + a1b1
c3 = co2 = b2c2 + a2c2 + a2b2
```

In other words, the equation for the carry-in to a particular stage is the same as the equation for the carry-out of the previous stage.

We can substitute the equation for c1 into c2's equation, resulting in:

```
c2 = b1c1 + a1c1 + a1b1
c2 = b1(b0c0 + a0c0 + a0b0) + a1(b0c0 + a0c0 + a0b0) + a1b1
c2 = b1b0c0 + b1a0c0 + b1a0b0 + a1b0c0 + a1a0c0 +
   a1a0b0 + a1b1
```

We can then substitute the equation for c2 into c3's equation, resulting in:

```
c3 = b2c2 + a2c2 + a2b2
c3 = b2(b1b0c0 + b1a0c0 + b1a0b0 + a1b0c0 + a1a0c0 +
     a1a0b0 + a1b1) + a2(b1b0c0 + b1a0c0 + b1a0b0
     + a1b0c0 + a1a0c0 + a1a0b0 + a1b1) + a2b2
c3 = b2b1b0c0 + b2b1a0c0 + b2b1a0b0 + b2a1b0c0 +
     b2a1a0c0 + b2a1a0b0 + b2a1b1 + a2b1b0c0
     + a2b1a0c0 + a2b1a0b0 + a2a1b0c0 + a2a1a0c0
     + a2a1a0b0 + a2a1b1 + a2b2
```

We'll omit the equation for c4 to save a few pages of paper.

We could create each stage with the needed inputs, and include a lookahead logic component implementing the above equations, as shown in Figure 6.55(c). Notice that there is no rippling of carry bits from stage to stage—each stage computes its own carry-in bit by "looking ahead" to the values of the previous stages.

While the above demonstrates the basic idea of carry-lookahead, the scheme is not very efficient. c1 requires 4 gates, c2 requires 8 gates, and c3 requires 16 gates, with each gate requiring more inputs in each stage. If we count gate inputs, c1 requires 9 gate inputs, c2 requires 27 gate inputs, and c3 requires 71 gate inputs. Building a larger adder, say an 8-bit adder, using this lookahead scheme would thus likely result in excessively large size. While the presented scheme is therefore not practical, it serves to introduce the basic idea of carry-lookahead: each stage looks ahead at the inputs to the previous stages and computes for itself whether that stage's carry-in bit should be 1, rather than waiting for the carry-in bit to ripple from previous stages, to yield a 4-bit adder with a delay of only 4 gate-delays.

An Efficient Carry-Lookahead Scheme. A more efficient carry-lookahead scheme is as follows. Consider again the addition of two 4-bit numbers A and B, shown in Figure 6.56(a). Suppose that we add each column's two operand bits (e.g., a0 + b0) using a half-adder, ignoring the carry-in bit of that column. The resulting half-adder outputs (carry-out and sum) provide useful information about the carry for the next stage. In particular:

- If the addition of a0 with b0 results in a carry-out of 1, then c1 will be 1 regardless of whether c0 is a 1 or 0. Why? If we add a0+b0+c0, then 1+1+0=10, while 1+1+1=11 (the "+" symbol represents add here, not OR)—both cases generate a carry-out of 1. Recall that a half-adder computes its carry-out as ab.
- If the addition of a0 with b0 results in a sum of 1, then c1 will be 1 only if c0 is 1. In particular, if we add a0+b0+c0, then 1+0+1=10 and 0+1+1=10. Recall that a half-adder computes its sum as a XOR b.

In other words, c1 will be 1 if a0b0 = 1, OR if a0 XOR b0 = 1 AND c0 = 1. The following equations describe the carry bits (the "+" symbol represents OR here, not add):

```
c1 = a0b0 + (a0 xor b0)c0
c2 = a1b1 + (a1 xor b1)c1
c3 = a2b2 + (a2 xor b2)c2
c4 = a3b3 + (a3 xor b3)c3
```

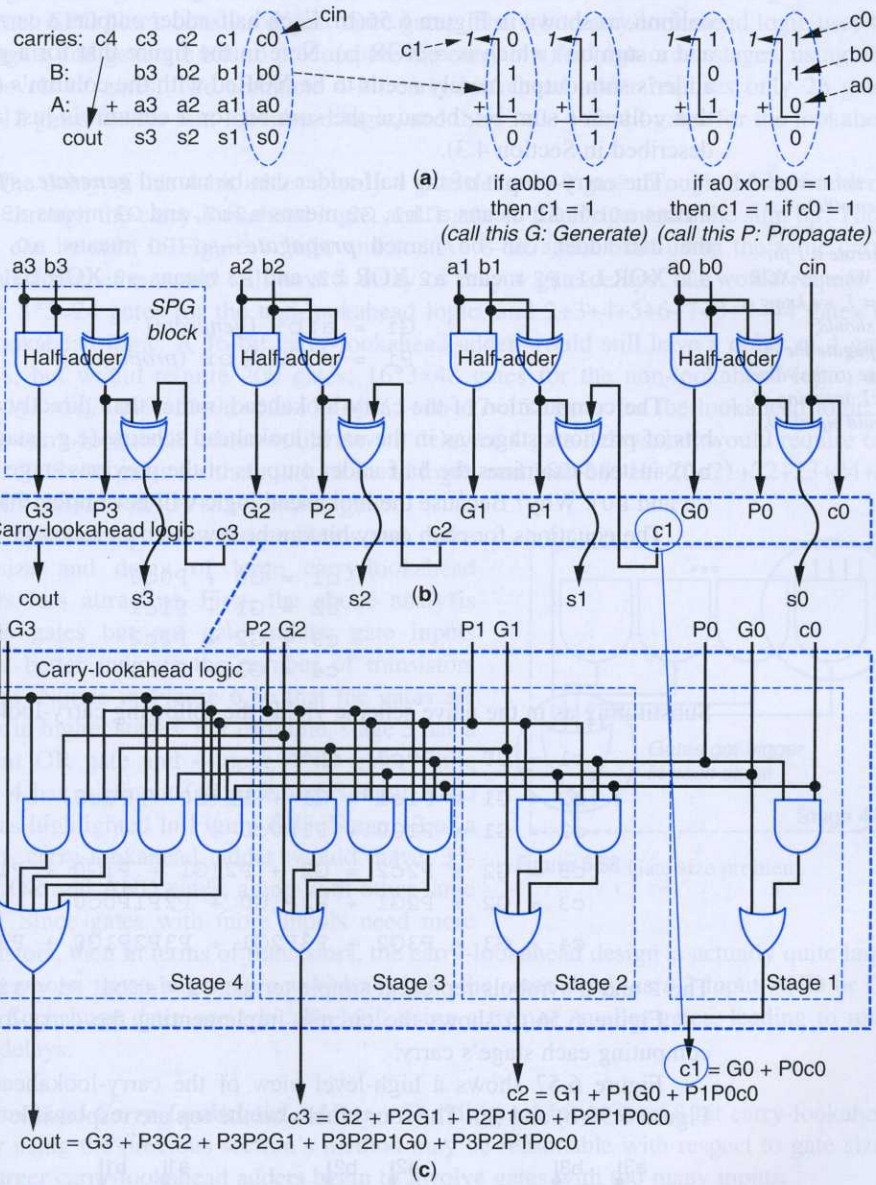


Figure 6.56 Adding two binary numbers using a fast carry-lookahead scheme: (a) idea of using propagate and generate terms, (b) computing the propagate and generate terms and providing them to the carry-lookahead logic, (c) using the propagate and generate terms to quickly compute the carries for each column. The correspondence between `c1` in figures (c) and (b) is shown by two circles connected by the line; similar correspondences exist for `c2` and `c3`.

A half-adder can be included in each stage to add the two operand bits for that column, as shown in Figure 6.56(b). Each half-adder outputs a carry-out bit (which is `ab`) and a sum bit (which is `a` XOR `b`). Note in the figure that for a given column, the half-adder's sum output merely needs to be XORed with the column's carry-in bit to compute that column's sum bit, because the sum bit for a column is just `a` XOR `b` XOR `c` (as described in Section 4.3).

*Why those names? When a0b0=1, we know we should **generate** a 1 for c1. When a0 XOR b0 = 1, we know we should **propagate** the c0 value as the value of c1, meaning c1 should equal c0.*

The carry-output of the half-adder can be named ***generate***, symbolized as `G`—so `G0` means `a0b0`, `G1` means `a1b1`, `G2` means `a2b2`, and `G3` means `a3b3`. The sum output of the half-adder can be named ***propagate***—so `P0` means `a0` XOR `b0`, `P1` means `a1` XOR `b1`, `P2` means `a2` XOR `b2`, and `P3` means `a3` XOR `b3`. In short:

`Gi = aibi` (*generate*)
`Pi = ai` XOR `bi` (*propagate*)

The computation of the carry-lookahead, rather than directly examining the operand bits of previous stages as in the naive lookahead scheme (e.g., stage 1 examining `a0` and `b0`), instead examines the half-adder outputs of the previous stage (e.g., stage 1 examines `G0` and `P0`). Why? Because the lookahead logic will be simpler than in the naive scheme.

The equations for each carry bit can be rewritten as follows:

```
c1 = G0 + P0c0
c2 = G1 + P1c1
c3 = G2 + P2c2
c4 = G3 + P3c3
```

Substituting as in the naive scheme yields the following carry-lookahead equations:

```
c1 = G0 + P0c0
c2 = G1 + P1c1 = G1 + P1(G0 + P0c0)
c2 = G1 + P1G0 + P1P0c0
c3 = G2 + P2c2 = G2 + P2(G1 + P1G0 + P1P0c0)
c3 = G2 + P2G1 + P2P1G0 + P2P1P0c0
c4 = G3 + P3G2 + P3P2G1 + P3P2P1G0 + P3P2P1P0c0
```

The `P` and `G` symbols represent simple terms: `Gi = aibi`, `Pi = ai` XOR `bi`.

Figure 6.56(c) shows the circuits implementing the carry-lookahead equations for computing each stage's carry.

Figure 6.57 shows a high-level view of the carry-lookahead adder's design from Figure 6.56(b) and (c). The four blocks on the top are responsible for generating the sum,

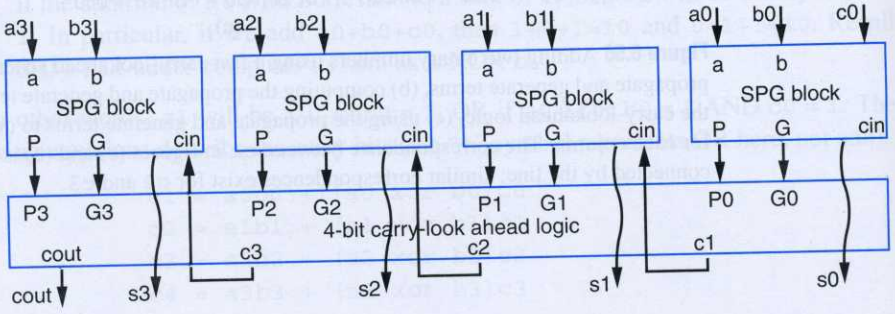


Figure 6.57 High-level view of a 4-bit carry-lookahead adder.

propagate, and generate bits—let's call those "*SPG blocks*." Recall from Figure 6.56(b) that each SPG block consists of just three gates. The 4-bit carry-lookahead logic uses the propagate and generate bits to precompute the carry bits for high-order stages, using only two levels of gates. The complete 4-bit carry-lookahead adder requires only 26 gates: 4*3=12 gates for the non-lookahead logic, and then 2+3+4+5=14 gates for the lookahead logic.

The delay of this 4-bit adder is only 4 gate-delays—1 gate through the half-adder, 2 gates through the carry-lookahead logic, and 1 gate to finally generate the sum bit. Those gates can be seen in Figure 6.56(b) and (c). An 8-bit adder built using the same carry-lookahead scheme would still have a delay of only 4 gate-delays, but would require 68 gates: 8*3=24 gates for the non-lookahead logic, and 2+3+4+5+6+7+8+9=44 gates for the lookahead logic. A 16-bit carry-lookahead adder would still have a delay of 4 gate-delays, but would require 200 gates: 16*3=48 gates for the non-lookahead logic, and 2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17=152 gates for the lookahead logic. A 32-bit carry-lookahead adder would have a delay of 4 gate-delays, but would require 656 gates: 32*3=96 gates for the non-lookahead logic, and 152+18+19+20+21+22+23+24+25+26+27+28+29+30+31+32+33=560 gates.

Unfortunately, there are problems that make the size and delay of large carry-lookahead adders less attractive. First, the above analysis counts gates but not gate inputs; gate inputs would better indicate the number of transistors needed. Notice in Figure 6.56 that the gates are wider in higher stages. For example, stage 3 has a 4-input OR gate and 4-input AND gate, while stage 4 has a 5-input OR gate and 5-input AND gate as highlighted in Figure 6.58. Stage 32 of a 32-bit carry-lookahead adder would have 33-input OR and AND gates, along with other large gates. Since gates with more inputs need more transistors, then in terms of transistors, the carry-lookahead design is actually quite large. Furthermore, those huge gates would not have the same delay as a 2-input AND or OR gate. Such huge gates are typically built using a tree of smaller gates, leading to more gate-delays.

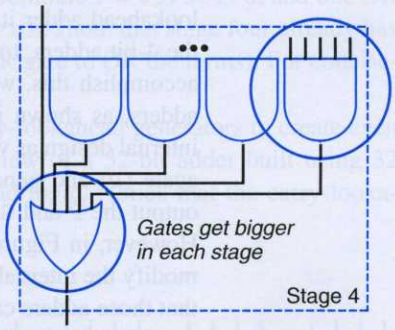


Figure 6.58 Gate size problem.

Hierarchical Carry-Lookahead Adders. Building a 4-bit or even 8-bit carry-lookahead adder using the previous section's method may be reasonable with respect to gate sizes, but larger carry-lookahead adders begin to involve gates with too many inputs.

A larger adder could instead be built by connecting smaller adders in a carry-ripple manner. For example, suppose 4-bit carry-lookahead adders are available. A16-bit adder can be built by connecting four 4-bit carry-lookahead adders, as shown in Figure 6.59. If each 4-bit carry-lookahead adder has a four gate delay, then the total delay of the 16-bit adder is 4+4+4+4=16 gate-delays. Compare this to the delay of a 16-bit carry-ripple adder—if each full-adder has a two gate-delay, then a 16-bit carry-ripple adder has a delay of 16*2=32 gate-delays. Thus, the 16-bit adder built from four carry-lookahead adders connected in a carry-ripple manner is twice as fast as the 16-bit carry-ripple adder. (Actually,

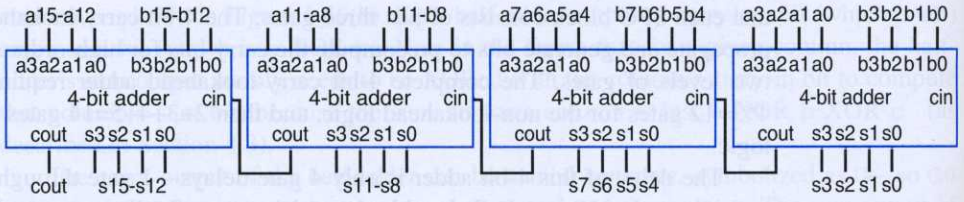


Figure 6.59 16-bit adder implemented using 4-bit adders connected in a carry-ripple manner. Can the delay of the rippling be avoided?

careful observation of Figure 6.53 reveals that the carry-out of a four-bit carry-lookahead adder would be generated in three gate-delays rather than four, resulting in even faster operation of the 16-bit adder built from four carry-lookahead adders—but for simplicity, let's not look inside the components for such detailed timing analysis.) Sixteen gate-delays is good, but can we do better? Can we avoid having to wait for the carries to ripple from the lower-order 4-bit adders to the higher-order adders?

In fact, avoiding the rippling is exactly what was done when developing the 4-bit carry-lookahead adder itself. Thus, we can *repeat the same carry-lookahead process outside* of the 4-bit adders, to quickly provide the carry-in values to the higher-order 4-bit adders. To accomplish this, we add another 4-bit carry-lookahead logic block outside the four 4-bit adders, as shown in Figure 6.60. The carry-lookahead logic block has exactly the same internal design as was shown in Figure 6.56(c). Notice that the lookahead logic needs propagate (`P`) and generate (`G`) signals from each adder block. Previously, each input block output the `P` and `G` signals just by ANDing and XORing the block's `ai` and `bi` input bits. However, in Figure 6.60, each block is a 4-bit carry-lookahead adder. We therefore must modify the internal design of a 4-bit carry-lookahead adder to output its `P` and `G` signals, so that those adders can be used with a second-level carry-lookahead generator.

Thus, let's extend the 4-bit carry-lookahead logic of Figure 6.56 to output `P` and `G` signals. The equations for the `P` and `G` outputs of a 4-bit carry-lookahead adder can be written as follows:

```
P = P3P2P1P0
G = G3 + P3G2 + P3P2G1 + P3P2P1G0
```

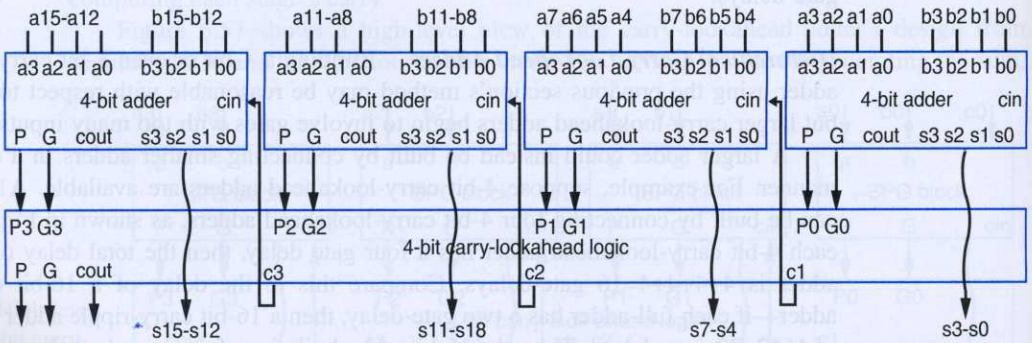


Figure 6.60 16-bit adder implemented using four CLA 4-bit adders and a second level of lookahead.

To understand these equations, recall that propagate meant that the output carry for a column should equal the input carry of the column (hence propagating the carry through the column). For that to be the case for the carry-in and carry-out of a 4-bit adder, the first stage of the 4-bit adder must propagate its input carry to its output carry, the second stage must propagate its input carry to its output carry, and so on for the third and fourth stages. In other words, each internal propagate signal must be `1`, hence the equation `P = P3P2P1P0`.

Likewise, recall that generate meant that the output carry of a column should be a `1` (hence generating a carry of `1`). Generate should thus be `1` if the first stage generates a carry (`G0`) and all the higher stages propagate the carry through (`P3P2P1`), yielding the term `P3P2P1G0`. Generate should also be a `1` if the second stage generates a carry and all higher stages propagate the carry through, yielding the term `P3P2G1`. Likewise for the third stage, whose term is `P3G2`. Finally, generate should be `1` if the fourth stage generates a carry, represented as `G3`. ORing all four of these terms yields the equation `G = G3 + P3G2 + P3P2G1 + P3P2P1G0`.

We would then revise the 4-bit carry-lookahead logic of Figure 6.56(c) to include two additional gates in stage four, one AND gate to compute `P = P3P2P1P0`, and one OR gate to compute `G = G3 + P3G2 + P3P2G1 + P3P2P1G0` (note that stage four already has AND gates for each term, so we need only add an OR gate to OR the terms). For conciseness, we omit a figure showing these two new gates.

We can introduce additional levels of 4-bit carry-lookahead generators to create even larger adders. Figure 6.61 illustrates a high-level view of a 32-bit adder built using 32 SPG blocks and three levels of 4-bit carry-lookahead logic. Notice that the carry-looka-

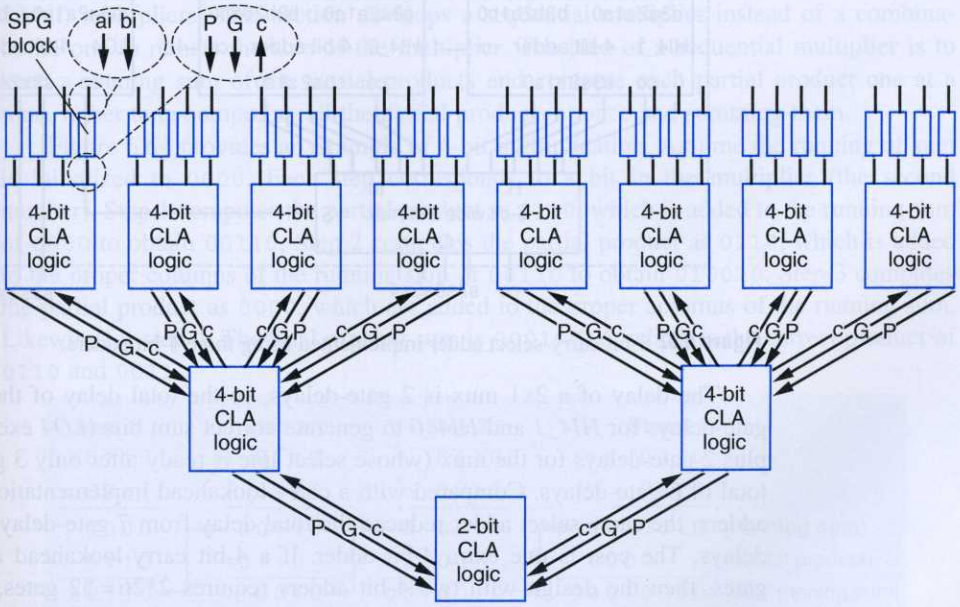


Figure 6.61 View of multilevel carry-lookahead showing the tree structure, which enables fast addition with reasonable numbers and sizes of gates. Each level adds only two gate-delays.

head logic forms a tree. Total delay for the 32-bit adder is only two gate-delays for the SPG blocks, and two gate-delays for each level of carry-lookahead (CLA) logic, for a total of 2+2+2+2 = 8 gate-delays. (Actually, closer examination of gate delays within each component would demonstrate that total delay of the 32-bit adder is actually less than 8 gate-delays.) Carry-lookahead adders built from multiple levels of carry-lookahead logic are known as ***multilevel*** or ***hierarchical carry-lookahead adders***.

In summary, the carry-lookahead approach results in faster additions of large binary numbers (more than 8 bits or so) than a carry-ripple approach, at the expense of more gates. However, by clever hierarchical design, the carry-lookahead gate size is kept reasonable.

Carry-Select Adders

Another way to build a larger adder from smaller adders is known as carry-select. Consider building an 8-bit adder from 4-bit adders. A carry-select approach uses two 4-bit adders for the high-order four bits, labeled *HI4_1* and *HI4_0* in Figure 6.62. *HI4_1 assumes* the carry-in will be `1`, while *HI4_0 assumes* the carry-in will be `0`, so both generate stable output at the same time that *LO4* generates stable output—after 4 gate-delays (assuming the 4-bit adder has a delay of four gate-delays). The *LO4* carry-out value selects among *HI4_1* or *HI4_0*, using a 5-bit-wide 2x1 multiplexer—hence the term ***carry-select adder***.

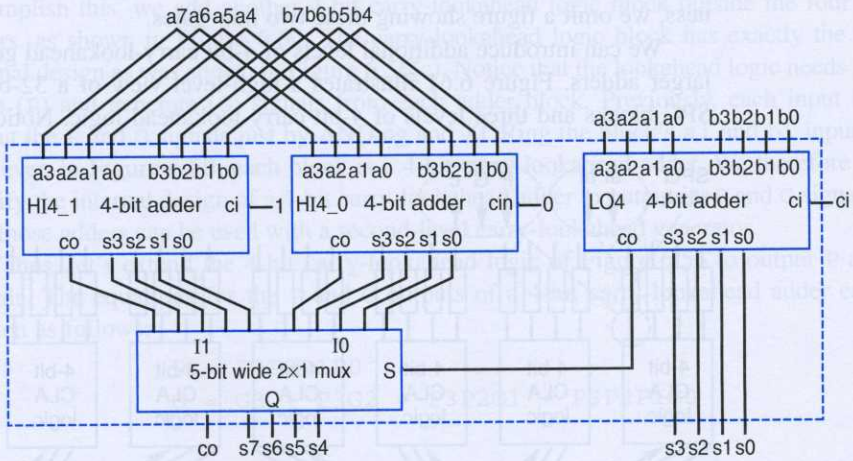


Figure 6.62 8-bit carry-select adder implemented using three 4-bit adders.

The delay of a 2x1 mux is 2 gate-delays, so the total delay of the 8-bit adder is 4 gate-delays for *HI4_1* and *HI4_0* to generate correct sum bits (*LO4* executes in parallel), plus 2 gate-delays for the mux (whose select line is ready after only 3 gate-delays), for a total of 6 gate-delays. Compared with a carry-lookahead implementation using two 4-bit adders, the carry-select adder reduced the total delay from 7 gate-delays down to 6 gate-delays. The cost is one extra 4-bit adder. If a 4-bit carry-lookahead adder requires 26 gates, then the design with two 4-bit adders requires 2*26=52 gates, while the carry-select adder requires 3*26=78 gates, plus the gates for the 5-bit 2x1 mux.

A 16-bit carry-select adder can built using 4-bit carry-lookahead adders by using multiple levels of multiplexing. Each nibble (four bits) has two 4-bit adders, one assuming a carry-in of 1, the other assuming 0. *Nibble0*'s carry-out selects, using a multiplexer, the appropriate adder for *Nibble1*. *Nibble1*'s selected carry-out selects the appropriate adder for *Nibble2*. *Nibble2*'s selected carry-out selects the appropriate adder for *Nibble3*. The delay of such an adder is 6 gate-delays for *Nibble1*, plus 2 gate-delays for *Nibble2*'s selection, plus 2 gate-delays for *Nibble3*'s selection—for a total of only 10 gate-delays. Cascading four 4-bit adders would yield 4+4+4+4 = 16 gates-delays. The speedup of the carry-select version over the cascaded version would be 16 / 10 = 1.6. Total size would be 7*26 = 182 gates, plus the gates for the three 5-bit 2x1 muxes. Carry-select adders provide good speed for reasonable size.

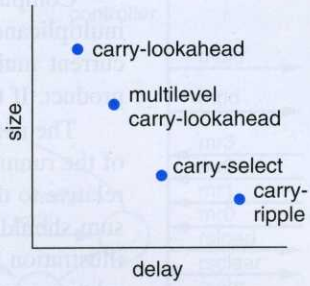


Figure 6.63 Adder tradeoffs.

Figure 6.63 illustrates the tradeoffs among adder designs. Carry-ripple is the smallest but has the longest delay. Carry-lookahead is the fastest but has the largest size. Carry-select is a compromise between the two, involving some lookahead and some rippling. The choice of the most appropriate adder for a design depends on the speed and size constraints of the design.

Smaller Multiplier—Sequential (Shift-and-Add) Style

An array-style multiplier can be fast, but may require many gates for wide-bitwidth (e.g., 32-bit) multipliers. This section develops a sequential multiplier instead of a combinational one to reduce the size of the multiplier. The idea of a sequential multiplier is to keep a running sum of the partial products and compute each partial product one at a time, rather than computing all the partial products at once and summing them.

Figure 6.64 provides an example of 4-bit multiplication. Assume the running of sum is initialized to 0000. Each step corresponds to a bit in the multiplier (the second number). Step 1 computes the partial product as 0110, which is added to the running sum of 0000 to obtain 00110. Step 2 computes the partial product as 0110, which is added to the proper columns of the running sum of 00110 to obtain 010010. Step 3 computes the partial product as 0000, which are added to the proper columns of the running sum. Likewise for step 4. The final running sum is 00010010, which is the correct product of 0110 and 0011.

Step 1	Step 2	Step 3	Step 4	
0110	0110	0110	0110	
× 0011	× 0011	× 0011	× 0011	
0000	00110	010010	0010010	**(running sum)**
+ 0110	+0110	+ 0000	+ 0000	(partial product)
00110	010010	0010010	00010010	(new running sum)

Figure 6.64 Multiplication done by generating a partial product for each bit in the multiplier (the number on the bottom), accumulating the partial products in a running sum.

Computing each partial product is easy, requiring just the ANDing of the current multiplicand bit with every bit in the multiplier to yield the partial product. So if the current multiplicand bit is `1`, the AND creates a copy of the multiplier as the partial product. If the current multiplicand bit is `0`, the AND creates `0` as the partial product.

The next thing to determine is how to add each partial product to the proper columns of the running sum. Notice that the partial product should be moved to the left by one bit relative to the running sum after each step. We can look at this another way—the running sum should be moved to the *right* by one bit after each step. Look at the multiplication illustration in Figure 6.64 until you "see" how the running sum moves one bit to the right relative to each partial product.

Therefore, the running sum can be computed by initializing an 8-bit register to 0. Each step adds the partial product for the current multiplicand bit to the leftmost four bits of the running sum, and shifts the running sum one bit to the right, shifting a `0` into the leftmost bit. So the running sum register should have a clear function, a parallel load function, and a shift right function. A circuit showing the running sum register and an adder to add each partial product to that register is shown in Figure 6.65.

The last thing to be determined is how to control the circuit so that the circuit does the right thing during each step, which is the purpose of controllers. Figure 6.66 shows an FSM describing the desired controller behavior of the sequential multiplier.

In terms of performance, the sequential multiplier requires two cycles per bit, plus 1 cycle for initialization. So a 4-bit multiplier would require 9 cycles, while a 32-bit multi-

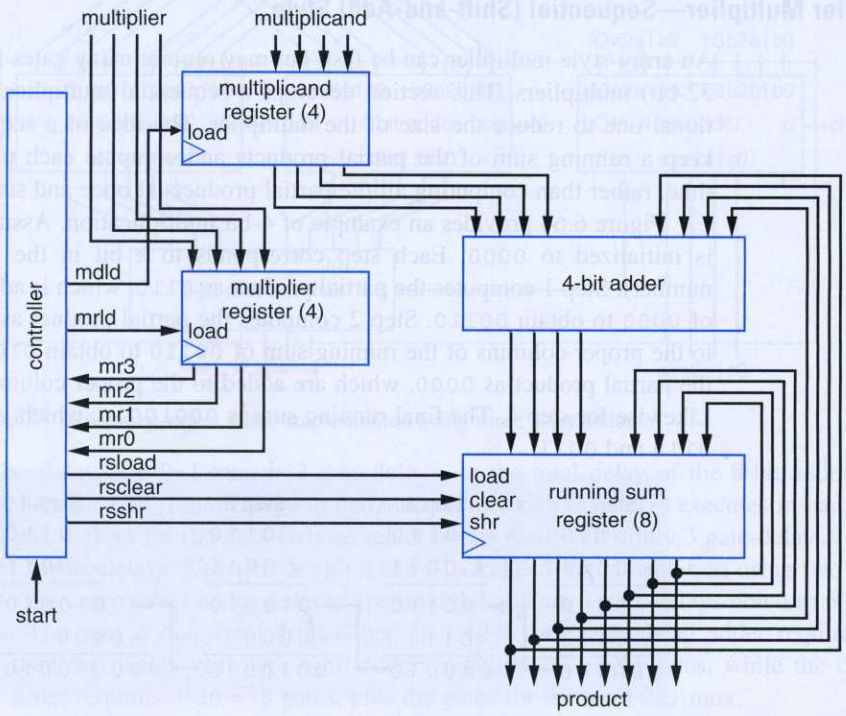


Figure 6.65 Internal design of a 4-bit by 4-bit sequential multiplier.

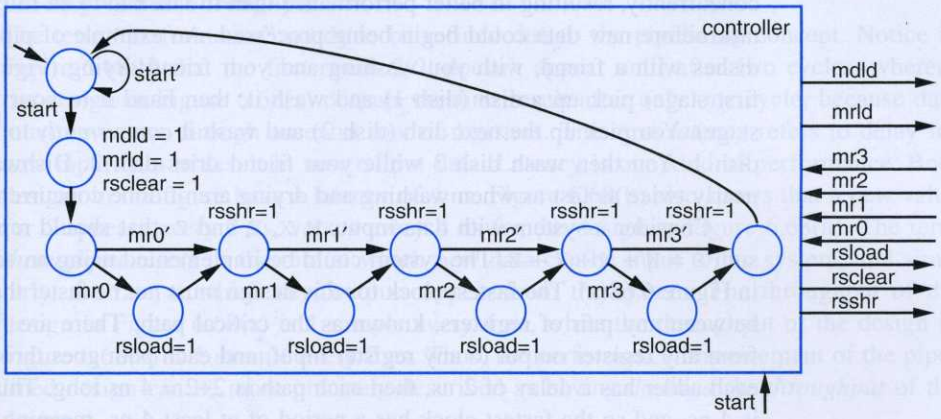


Figure 6.66 FSM describing the controller for the 4-bit multiplier.

plier would require 65 cycles. The longest register-to-register delay is from a register through the adder to a register. If the adder is a carry-lookahead adder having only 4 gate-delays, then the total delay for a 4-bit multiplication would be 9 cycles * 4 gate-delays/cycle = 36 gate-delays. The total delay for a 32-bit multiplication would be 65 cycles * 4 gate-delays/cycle = 260 gate-delays. While slow, notice that this multiplier's size is quite small, requiring only an adder, a few registers, and a state-register and some control logic for the controller. For a 32-bit multiplier, the size would be far smaller than an array-style multiplier requiring 31 adders.

The multiplier's design can be further improved by using a shifter in the datapath, but we omit details of that improved design.

6.5 RTL DESIGN OPTIMIZATIONS AND TRADEOFFS

Chapter 5 described the RTL design process. While creating the datapath during RTL design, several optimizations and tradeoffs can be used to create smaller or faster designs.

Pipelining

Microprocessors continue to become smaller, faster, and less expensive, and thus designers use microprocessors whenever possible to implement desired digital system behavior. But designers continue to choose to build their own digital circuits to implement desired behavior of many digital systems, with a key reason for that choice being *speed*. One method for obtaining speed from digital circuits is through the use of pipelining. ***Pipelining*** means breaking a large task down into a sequence of stages such that data moves through the stages like parts moving through a factory assembly line. Each stage produces output used by the next stage, and all stages operate

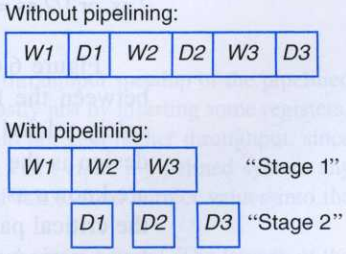


Figure 6.67 Applying pipelining to dishwashing: washing and drying dishes can be done concurrently.

concurrently, resulting in better performance than if data had to be fully processed by the task before new data could begin being processed. An example of pipelining is washing dishes with a friend, with you washing and your friend drying (Figure 6.67). You (the first stage) pick up a dish (dish 1) and wash it, then hand it to your friend (the second stage). You pick up the next dish (dish 2) and wash it *concurrently* to your friend drying dish 1. You then wash dish 3 while your friend dries dish 2. Dishwashing this way is nearly twice as fast as when washing and drying aren't done concurrently.

Consider a system with data inputs `W`, `X`, `Y`, and `Z`, that should repeatedly output the sum `S = W + X + Y + Z`. The system could be implemented using an adder tree as shown in Figure 6.68(a). The fastest clock for this design must not be faster than the longest path between any pair of registers, known as the critical path. There are four possible paths from any register output to any register input, and each path goes through two adders. If each adder has a delay of 2 ns, then each path is 2+2 = 4 ns long. Thus, the critical path is 4 ns, and so the fastest clock has a period of at least 4 ns, meaning a frequency of no more than 1 / 4 ns = 250 MHz.

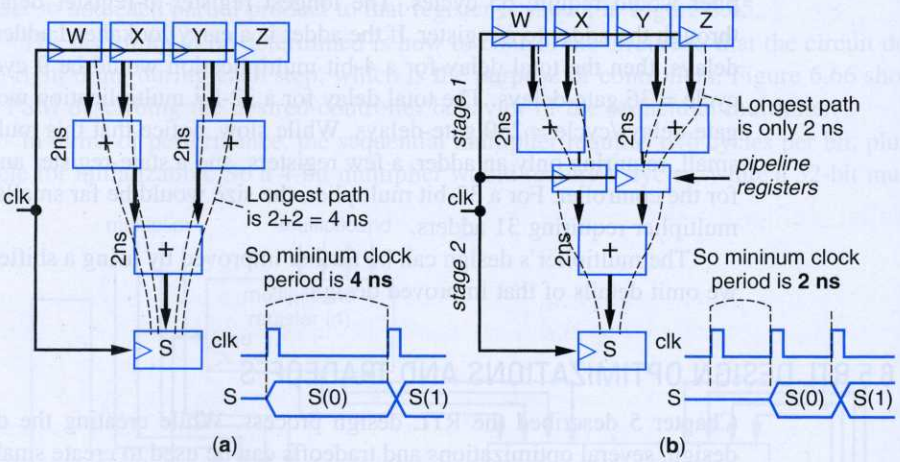


Figure 6.68 Non-pipelined versus pipelined datapaths: (a) four register-to-register paths of 4 ns each, so longest path is 4 ns, meaning minimum clock period is 4 ns, or 1/4 ns = 250 MHz, (b) six register-to-register paths of 2 ns each, so longest path is 2 ns, meaning minimum clock period of 2 ns, or 1/2 ns = 500 MHz.

Figure 6.68(b) shows a pipelined version of this design. We merely add registers between the first and second rows of adders. Since the purpose of these registers is solely related to pipelining, they are known as ***pipeline registers***, though their internal design is the same as any other register. The computations between pipeline registers are known as ***stages***. By inserting those registers and thus creating a two-stage pipeline, the critical path has been reduced from 4 ns down to only 2 ns, and so the fastest clock has a period of at least 2 ns, meaning a frequency of no more than 1/2 ns = 500 MHz. In other words, just by inserting those pipeline registers, we've *doubled the performance* of the design!

Latency versus Throughput

The term "performance" needs to be refined due to the pipelining concept. Notice in Figure 6.68(b) that the first result *S(0)* doesn't appear until after two cycles, whereas the design in Figure 6.68(a) outputs the first result after only one cycle, because data must now pass through an extra row of registers. The term ***latency*** refers to delay for new input data to result in new output data. Latency is one kind of performance. Both designs in the figure have a latency of 4 ns. Figure 6.68(b) also shows that a new value for *S* appears every 2 ns, versus every 4 ns for the design in Figure 6.68(a). The term ***throughput*** refers to the rate at which new data can be input to the system, and similarly, the rate at which new outputs appear from the system. The throughput of the design in Figure 6.68(a) is 1 sample every 4 ns, while the throughput of the design in Figure 6.68(b) is 1 sample every 2 ns. Thus, the performance improvement of the pipelined design can be more precisely described as having *doubled the throughput* of the design.

Example 6.19 Pipelined FIR filter

Recall the 100-tap FIR filter from Example 5.8. We estimated that implementation on a microprocessor would require 4000 ns, while a custom digital circuit implementation would require only 34 ns. That custom digital circuit utilized an adder tree, with seven levels of adders—50 additions, then 25, then 13 (roughly), then 7, then 4, then 2, then 1. The total delay was 20 ns (for the multiplier) plus seven adder-delays (7*2ns = 14ns), for a total delay of 34 ns. We can further improve the throughput of that filter using pipelining. Noticing that the multipliers' delay of 20 ns is roughly equal to the adder tree delay of 14 ns, we decide to insert pipeline registers (50 of them since there are 50 multipliers feeding into 50 adders at the top of the adder tree) between the multipliers and adder tree, thus dividing the computation task into two stages, as shown in Figure 6.69. Those pipeline registers shorten the critical path from 34 ns down to only 20 ns, meaning we can clock the circuit faster and hence improve the throughput. The throughput speedup of the unpipelined design compared to the microprocessor implementation was 4000/34 = 117, while the throughput speedup of the pipelined design is 4000/20 = 200. The additional speedup is obtained mostly just by inserting some registers.

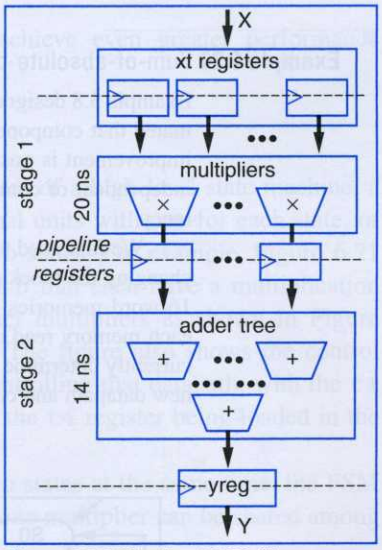


Figure 6.69 Pipelined FIR filter.

Although we could pipeline the adder tree also, that would not gain higher throughput, since the multiplier stage would still represent the critical path. We can't clock a pipelined system any faster than the longest stage, since otherwise that stage would fail to load correct values into the stage's output pipeline registers.

The latency of the non-pipelined design is one cycle of 34 ns, or 34 ns total. The latency of the pipelined design is two cycles of 20 ns, or 40 ns total. Thus, pipelining improves the throughput of this example at the expense of latency, representing a tradeoff.

Concurrency

A key reason for designing a custom digital circuit, rather than writing software that executes on a microprocessor, is to achieve improved performance. A common method of achieving performance in a custom digital circuit is through concurrency. ***Concurrency*** in digital design means to divide a task into several independent subparts, and then to execute those subparts simultaneously. As an analogy, if we have a stack of 200 dishes to wash, we might divide the stack into 10 sub-stacks of 20 dishes each, and then give 10 of our neighbors each a sub-stack. Those neighbors simultaneously go home, wash and dry their respective sub-stacks, and return to us their completed dishes. We would get a ten times speedup in dishwashing (ignoring the time to divide the stack and move sub-stacks from home to home).

Several previous examples used concurrency already. For example, the FIR filter datapath of Figure 5.38 had three multipliers executing concurrently.

The following example uses concurrency to create a faster version of an earlier example.

Example 6.20 Sum-of-absolute-difference component with concurrency

Example 5.8 designed a custom circuit for a sum-of-absolute-difference (SAD) component, and estimated that component to be three times faster than a software-on-microprocessor solution. Further improvement is possible. Notice that comparing one pair of corresponding pixels of two frames is independent of comparing another pair. Thus, such comparisons are an ideal candidate for concurrency.

We first need to be able to read the pixels concurrently. Concurrent reading can be achieved by changing the block memories *A* and *B*, which earlier were 256-byte memories. Instead, we can use 16-word memories where each word is 16 bytes (the total memory size is still 256 bytes). Thus, each memory read corresponds to reading an entire pixel row of a 16x16 block. We can then concurrently determine the differences among all 16 pairs of pixels from *A* and *B*. Figure 6.70 shows a new datapath and controller FSM for a more concurrent SAD component.

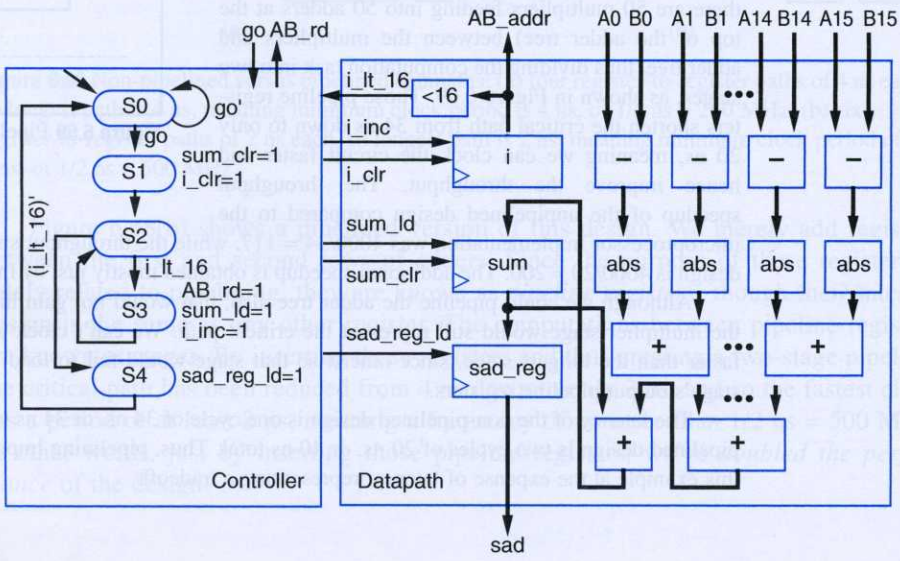


Figure 6.70 SAD datapath using concurrency for speed, along with the controller FSM.

The datapath consists of 16 subtractors operating concurrently on the 16 pixels of a row, followed by 16 absolute value components. The 16 resulting differences feed into an adder tree, whose result gets added with the present sum, for writing back into the sum register. The datapath compares its counter *i* with 16, since there are 16 rows in a block, and so the difference between rows must be computed 16 times. The controlling FSM loops 16 times to accumulate the differences of each row, and then loads the final result into the register *sad_reg*, which connects to the SAD component's output *sad*.

The analysis after Example 5.8 estimated that a software solution would require about six cycles per pixel pair comparison. Since there are 256 pixels in a 16x16 block, the software would require 256 * 6 = 1536 cycles to compare a pair of blocks. The SAD circuit with concurrency instead requires only 1 cycle to compare each row of 16 pixels, which the circuit must do 16 times for a block, resulting in only 16 * 1 = 16 cycles. Thus, the SAD circuit's speedup over software is 1536 / 16 = 96. In other words, the relatively simple SAD circuit using concurrency runs nearly 100 times faster than a software solution. That sort of speedup eventually translates to better-quality digitized video from whatever video appliance is being designed.

Pipelining and concurrency can be combined to achieve even greater performance improvements.

Component Allocation

When the same operation is used in two different states of a high-level state machine, a designer can choose to instantiate either two functional units with one for each state, or one functional unit that will be shared among the two states. For example, Figure 6.71 shows a portion of an HLSM with two states *A* and *B* that each have a multiplication operation. A designer can choose to use two distinct multipliers as shown in Figure 6.71(a) (assume the `t` variables represent registers). The figure also shows the control signals that would be set in each state of the FSM controlling that datapath, with the `t1` register being loaded in the first state (`t1ld=1`), and the `t4` register being loaded in the second state (`t4ld=1`).

However, because a state machine can't be in two states at the same time, the FSM will perform only one multiplication at a time, so the one multiplier can be shared among

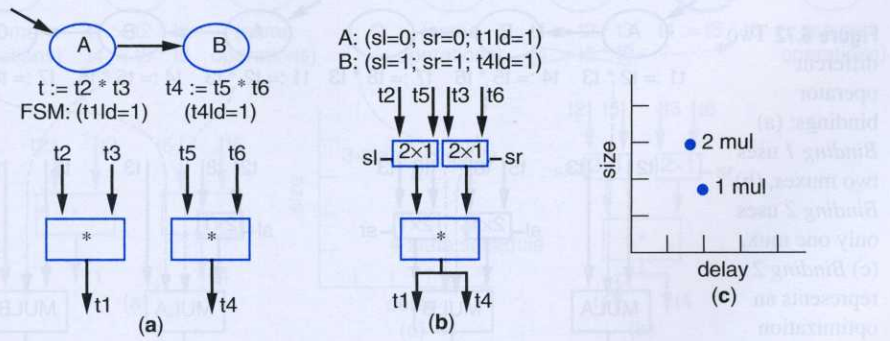


Figure 6.71 Two different component allocations: (a) two multipliers, (b) one multiplier, (c) the one-multiplier allocation represents a tradeoff of smaller size for slightly more delay.

the two states. Because fast multipliers are big, such sharing could save many gates. A datapath with only one multiplier appears in Figure 6.71(b). In each state, the controller FSM would configure the multiplexer select lines to pass the appropriate operands through to the multiplier, and to load the appropriate destination register. So in the first state *A*, the FSM would set the select line for the left multiplexer to `0` (`sl=0`) to let `t2` pass through and would set the select line for the right multiplexer to `0` (`sr=0`) to let `t3` pass through, in addition to setting `t1ld=1` to load the result of the multiplication into the `t1` register. Likewise, the FSM in state *B* sets the muxes to pass `t5` and `t6`, and loads `t4`.

Figure 6.71(c) illustrates that the one-multiplier design would have smaller size at the expense of slightly more delay due to the multiplexers.

The terms "operator" and "operation" refer to behavior, like addition or multiplication. The terms "component" and "functional unit" refer to an item in a circuit, like an adder or a multiplier.

A component library might consist of numerous different functional units that could potentially implement a desired operation. For a multiplication, there may be several multiplier components: *MUL1* might be very fast but large, while *MUL2* might be very small but slow, and *MUL3* may be somewhere in between. There may also be fast but large adders, small but slow adders, and several choices in between. Furthermore, some components might support multiple operations, like an adder/subtractor component, or an ALU. Choosing a particular set of functional units to implement a set of operations is known as ***component allocation***. Automated RTL design tools consider dozens or hundreds of possible component allocations to find the best ones that represent a good tradeoff between size and performance.

Operator Binding

Given an allocation of components, a designer must choose which operations to assign, or ***bind***, to which components. For example, Figure 6.72 shows three multiplication operations, one in state *A*, one in state *B*, and one in state *C*. Figure 6.72(a) shows one possible mapping of multiplication operations to two multipliers, resulting in two multiplexers. Figure 6.72(b) shows an alternative mapping to two multipliers, which results in only one multiplexer, since the same operand (*t3*) is fed to the same multiplier *MULA* in two different states and thus that multiplier's input doesn't require a mux. Thus, the second mapping results in fewer gates, with no performance loss—an optimization, as shown in

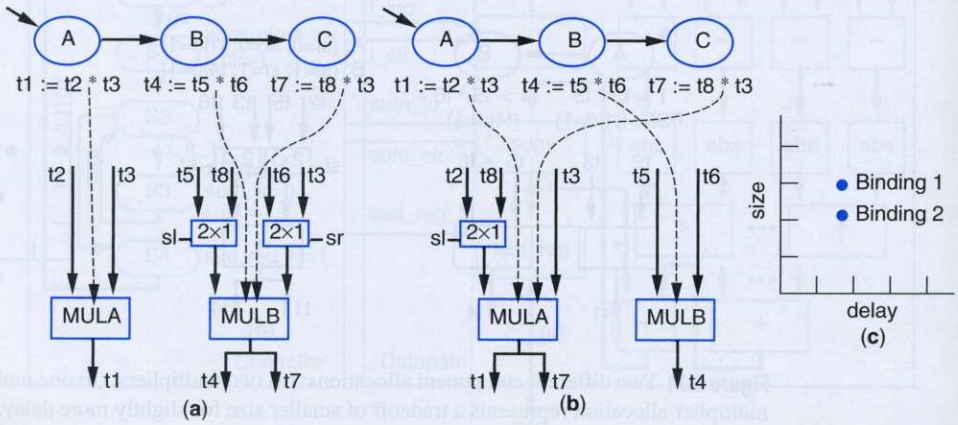


Figure 6.72 Two different operator bindings: (a) *Binding 1* uses two muxes, (b) *Binding 2* uses only one mux, (c) *Binding 2* represents an optimization compared to *Binding 1*.

Figure 6.72(c). Note that binding not only maps operators to components, but also chooses which operand to map to which component input; if we had mapped `t3` to the left operand of *MULA* in Figure 6.72(b), then *MULA* would have required two muxes rather than just one.

Mapping a given set of operations to a particular component allocation is known as ***operator binding***. Automated tools typically explore hundreds of different bindings for a given component allocation.

Of course, the tasks of component allocation and operator binding are interdependent. If only one component is allocated, then all operators must be bound to that component. If two components are allocated, then there are some choices in binding. If many components are allocated, then many more choices for binding exist. Thus, some tools will perform allocation and binding simultaneously, and other tools will iterate between the two tasks. Together, component allocation and operator binding are sometimes referred to as ***resource sharing***.

Operator Scheduling

Given a high-level state machine, additional states may be introduced to enable creating a smaller datapath. For example, consider the HLSM in Figure 6.73(a).The HLSM has three states, with state *B* having two multiplications. Since those two multiplications occur in the same state, and each state will be a single clock cycle, then two multipliers (at least) are needed in the datapath to support the two simultaneous multiplications in state *B*. But what if enough gates exist for only one multiplier? In that case, the operations can be rescheduled so that there is at most only one multiplication performed in any one state, as in Figure 6.73(b). Thus, when components are being allocated, only one multiplier need be allocated as shown, and as was also done in Figure 6.71(b). The result is a smaller but slower design due to the extra state's clock cycle, as illustrated in Figure 6.73(c).

Converting a computation from occurring concurrently in one state to occurring across several states is known as ***serializing*** a computation.

Of course, the inverse rescheduling is also possible. Suppose we started with the HLSM of Figure 6.73(b). If plenty of gates and available and improved performance are

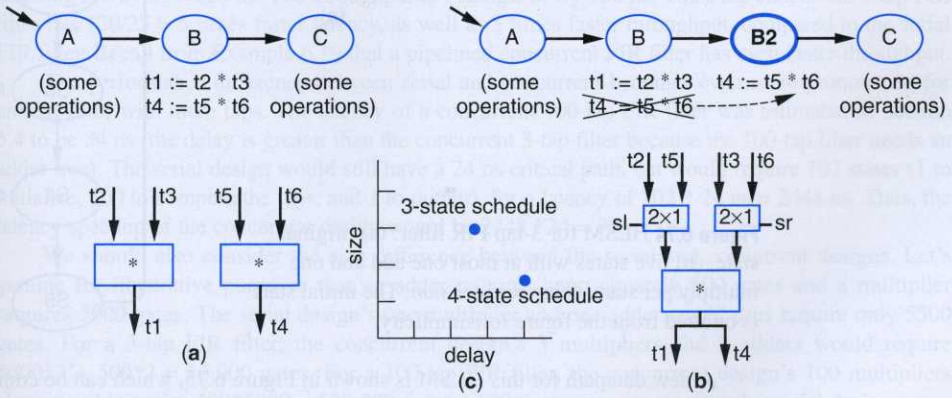


Figure 6.73 Scheduling: (a) initial 3-state schedule requires two multipliers, (b) new 4-state schedule requires only one multiplier, (c) new schedule trades off size for delay (extra state).

desired, the operations can be rescheduled such that the operations of states *B2* and *B* are merged into the one state *B*, as in Figure 6.73(a). The result is a faster but larger design requiring two multipliers instead of one.

Generally, introducing or merging states and assigning operations to those states are together a task known as ***operator scheduling***.

You may have noticed that operator scheduling is interdependent with component allocation, which you may recall was interdependent with operator binding. Thus, the tasks of scheduling, allocation, and binding are all interdependent. Modern tools may combine the tasks and/or may iterate among the tasks several times in search of good designs.

Example 6.21 Smaller FIR filter using operator scheduling

Consider the 3-tap FIR filter of Example 5.10. That design had one state containing the key datapath actions, as shown in Figure 6.74(a). We could reduce the size of the datapath by scheduling the operations across several states, such that at most one multiplication and one addition occurs per state, as shown in Figure 6.74(b). The first state loads the *x* registers with samples—note that the ordering of those actions next to the state doesn't matter, since all the actions occur simultaneously. That state also clears a new register named *sum*, which was introduced to keep track of the intermediate tap sums to be computed in the later states. The second state computes the first tap of the filter result, the next state computes the second tap, and the next state computes the third tap. The last state outputs the result, and then the HLSM returns to the first state again.

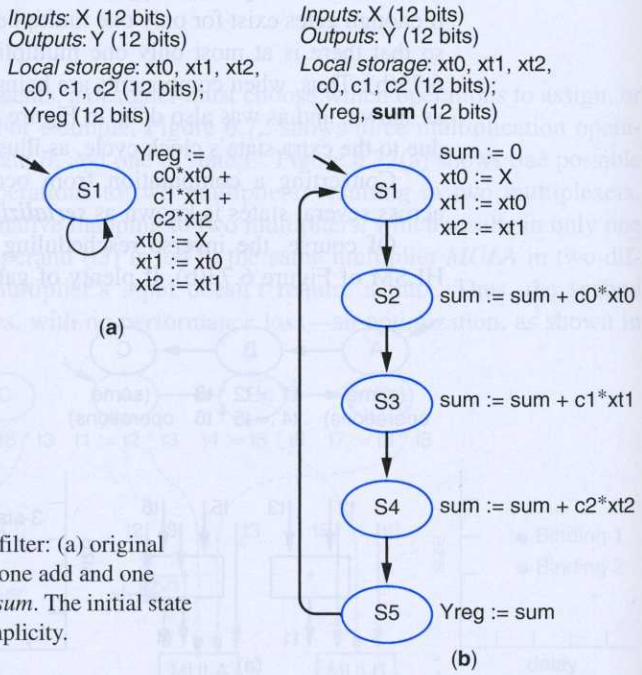


Figure 6.74 HLSM for 3-tap FIR filter: (a) original state, (b) five states with at most one add and one multiply per state, using register *sum*. The initial state is omitted from the figure for simplicity.

A new datapath for this HLSM is shown in Figure 6.75, which can be compared with the original datapath of Figure 5.40 (the new datapath figure only shows control lines that were added to the original datapath). The datapath requires only one multiplier and one adder, because there is at most one multiplication and one addition in any given state in Figure 6.74. The particular configu-

ration of the multiplier, adder, and register in Figure 6.75 is extremely common in signal processing circuits and is known as a ***multiply-accumulate*** (***MAC***) unit. The datapath multiplexes the inputs to the MAC unit.

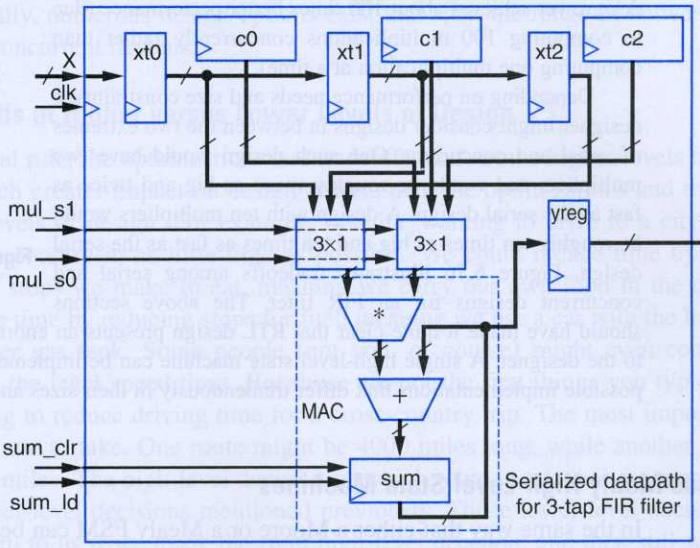


Figure 6.75 Serial FIR filter datapath. The components in the dashed box comprise what is known as a multiply-accumulate (MAC) component.

The performance of the concurrent design of Example 5.10 was estimated assuming 1 ns per gate, 2 ns per adder, and 20 ns per multiplier. The design had a critical path of 20 ns for the multiplier and then 4 ns for two adders in series, for a total of 24 ns. That was also the time between new results being taken in at the inputs and generated at the output: 24 ns. Using the more precise performance measures of latency and throughput defined in Section 6.5, the concurrent design has a latency of 24 ns (delay from input to output) and a throughput of 1 sample every 24 ns. The serial design has a critical path equal to the delay through a mux, multiplier, and adder. Assuming two gate-delays for the mux, we obtain a delay of 2 ns + 20 ns + 2 ns, or 24 ns. The latency from input to output is five states, meaning 5 * 24 ns = 120 ns. The throughput is 1 sample every 120 ns. Thus, the concurrent 3-tap FIR filter has 120/25 = 5 times faster latency, as well as 5 times faster throughput, compared to the serial FIR filter. Recall from Example 6.19 that a pipelined concurrent FIR filter has even faster throughput.

The performance difference between serial and concurrent becomes even more pronounced for an FIR filter with more taps. The latency of a concurrent 100-tap FIR filter was estimated in Section 5.4 to be 34 ns (the delay is greater than the concurrent 3-tap filter because the 100-tap filter needs an adder tree). The serial design would still have a 24 ns critical path, but would require 102 states (1 to initialize, 100 to compute the taps, and 1 to output), for a latency of 102 * 24 ns = 2448 ns. Thus, the latency speedup of the concurrent design would be 2448 / 34 = 72.

We should also consider the size difference between the serial and concurrent designs. Let's assume for illustrative purposes that an adder requires approximately 500 gates and a multiplier requires 5000 gates. The serial design's one multiplier and one adder would thus require only 5500 gates. For a 3-tap FIR filter, the concurrent design's 3 multipliers and 2 adders would require 5000*3 + 500*2 = 16,000 gates. For a 100-tap FIR filter, the concurrent design's 100 multipliers alone would require 100*5000 = 500,000 gates—100 times more gates than the serial design.

Intuitively, these numbers make sense. A concurrent design for 100 taps uses about 100 times more gates (due to using 100 multipliers instead of just 1) compared to a serial design, yet achieves about 100 times better performance (due to computing 100 multiplications concurrently rather than computing one multiplication at a time).

Depending on performance needs and size constraints, a designer might consider designs in between the two extremes of serial and concurrent. One such design would have two multipliers and would be roughly twice as big and twice as fast as the serial design. A design with ten multipliers would be roughly ten times as big and ten times as fast as the serial design. Figure 6.76 illustrates tradeoffs among serial and concurrent designs for an FIR filter. The above sections should have made it quite clear that RTL design presents an enormous range of possible solutions to the designer. A single high-level state machine can be implemented as any of a huge variety of possible implementations that differ tremendously in their sizes and performance.

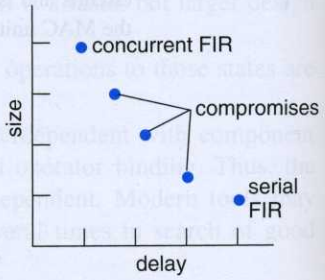


Figure 6.76 FIR design tradeoffs.

Moore versus Mealy High-Level State Machines

In the same way that either a Moore or a Mealy FSM can be created (see Section 6.3), we can create Moore or Mealy high-level state machines. In the case of a high-level state machine, a Moore type can only have actions associated with the states, while a Mealy type can have actions associated with the transitions. As was the case with FSMs, a Mealy type may result in fewer states. Mixing Moore and Mealy types is commonly done in HLSMs.

▶ 6.6 MORE ON OPTIMIZATIONS AND TRADEOFFS

Serial versus Concurrent Computation

Having seen in this chapter numerous examples of tradeoff techniques at various levels of design, we can detect a common theme underlying some of those tradeoffs. The common theme is that of serial versus concurrent computation. ***Serial*** means to perform tasks one at a time. ***Concurrent*** means to perform tasks at the same time.

For example, in combinational logic design, we can reduce logic size by factoring out terms. By factoring out terms, we are essentially serializing the computation, by computing the factored out terms first, and then combining the results with other terms. In datapath component design, we can improve an adder's speed by computing carries concurrently, rather than waiting for the carry to ripple serially. In RTL design, we can schedule operations across several states, serializing the operations to reduce size compared to concurrent operations in a single state. Example 6.20 and Example 6.21 both illustrated serial versus concurrent computation tradeoffs, for an SAD circuit and an FIR circuit, respectively.

Trading off between serial and concurrent computation is a fundamental concept spanning all levels of digital design. As a general rule, a concurrent design is faster but larger, while a serial design is smaller but slower.

Typically, numerous design options exist that span the range in between fully serial and fully concurrent designs.

Optimizations and Tradeoffs at Higher versus Lower Levels of Design

As a general rule, the optimizations and tradeoffs made at the higher levels of design may have a much greater impact on design criteria than the optimizations and tradeoffs made at lower levels of design. For example, imagine wanting to drive to a city on the other side of the country in as little time as possible. We could reduce time by reducing the number of stops we make to eat, meaning we carry our own food in the car. We could also reduce time by reducing stops for fuel, meaning we use a car with the longest driving capacity per gas tank. Some people (not you, of course) might even consider driving faster than the legal speed limit. But those are not the first things you typically think of when trying to reduce driving time for a cross-country trip. The most important decision is which route to take. One route might be 4000 miles long, while another route may be only 2000 miles. The high-level decision of which route to take has far more impact than all the lower-level decisions mentioned previously. Those lower-level decisions are only really useful to us if we made the right high-level decision, and then still want to reduce the time further.

In digital design, optimization/tradeoff decisions at the higher levels (e.g., RTL decisions) may have a much larger impact than decisions at the lower levels (e.g., datapath component decisions or multilevel logic decisions). For example, the RTL decision to build a serial or concurrent FIR filter (Example 6.21) will have a far greater impact on circuit size and performance than the datapath-component-level decision to use a carry-ripple or carry-lookahead adder, or the combinational-logic-level decision to use two-level or multilevel logic. Those lower-level decisions merely tune the size and performance of the higher-level decision. Figure 6.77(a) illustrates this concept. An analogy might be a spotlight shining down on land, illustrated in Figure 6.77(b)—moving the spotlight left or right at high altitude (higher-level decisions) has a larger impact on which land region (possible solutions) is illuminated than do lower-altitude movements (lower-level decisions).

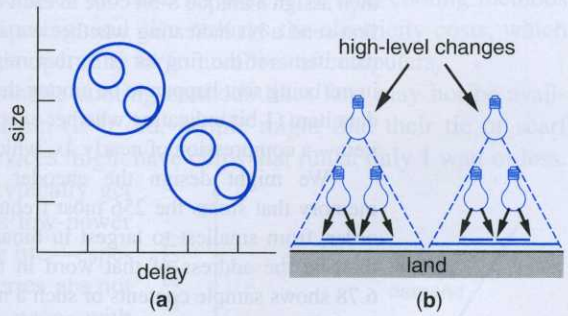


Figure 6.77 Higher- versus lower-level decisions: (a) higher-level decisions (denoted by the larger two circles) focus the design into a region, while lower-level decisions tune within the region, (b) spotlight analogy.

Algorithm Selection

When attempting to implement a system as a digital circuit, perhaps the highest-level design decision, having therefore the most significant impact on design criteria like size, performance, power, etc., is the selection of an algorithm. An ***algorithm*** is a set of steps that solve a problem. The same problem can be solved by different algorithms. Algorithms for the same problem, when implemented as a digital circuit, may result in tremendously different performance and/or size. Some algorithms may simply be better than others (optimization without much tradeoff), while other algorithms may represent tradeoffs among performance, size, and other criteria. Selecting an algorithm for a digital design problem is perhaps the highest level of design, and can have the biggest impact on design criteria. For example, earlier examples showed various implementations of an FIR filter. But there are many other filtering algorithms that are very different from the algorithm used in FIR. Some algorithms may provide higher-quality filtering at the expense of more required computation; others may provide lower quality but need less computation.

The following example illustrates algorithm selection.

Example 6.22 Data compression using different table lookup algorithms

We wish to compress data being sent over a long-distance computer network in order to achieve faster communication by sending fewer bits. One method for such compression is to use short codes for frequently appearing data values. For example, suppose each data item is 32 bits long. We might analyze the data we expect to send and find the 256 most frequently appearing data values. We could then assign a unique 8-bit code to each of those 256 values. When sending data over the network, we first send a bit indicating whether we are about to send an encoded 8-bit data item or a raw 32-bit data item—if the first bit is `1`, that might mean encoded, and a `0` might mean raw. If all the data items being sent happen to be among the top 256 most frequent ones, then we'd be sending 9 bits per data item (1 bit indicating whether encoded, plus 8 bits of encoded data) rather than 32 bits per data item—a compression of nearly 4x, which could translate to about 4 times faster communication.

We might design the encoder using a 256-word memory that stores the 256 most frequent values in sorted order, from smallest to largest in binary. The code would then be the address of that word in the memory. Figure 6.78 shows sample contents of such a memory, in hexadecimal. The contents vary depending on the communicating applications we are considering.

One algorithm for searching a list of values in a memory is known as ***linear search***. Starting at address 0, we compare each memory word's contents with the data item we are looking for (known as the key), incrementing the address and repeating until we find a match, at which point we treat the address at which there was a match as the encoded value. If we get to address 255 and don't find a match, we will transmit the raw data. The linear search algorithm is a slow way to search a sorted list in memory. The algorithm requires 256 reads and compares for data items that aren't in the memory, which may translate to 256 cycles. For data items that are in the memory, we would require on average 128 reads.

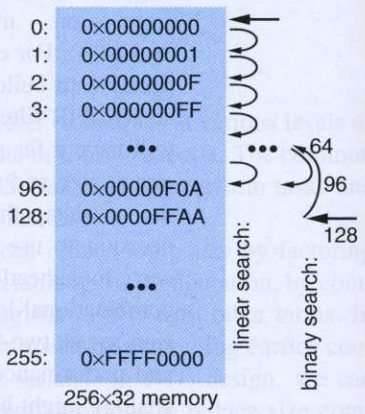


Figure 6.78 Searching a sorted memory for the key 0x00000F0A —linear search requires 97 reads/compares, binary search only 3.

A faster algorithm for searching a list of items in a memory is known as ***binary search***. We first sort the list and then store the list in the memory (we need only sort once). To look up an item, we start in the middle of the memory, meaning address 128, and compare that word's contents with the key. If the contents' value is less than 128, then we know that the key, if it exists in the memory, must be somewhere between 0 and 127. So we go to the middle of that range, meaning address 64, and again compare. If the value there is less than the key, we search 0 to 63; if greater, we search 65–127. So after each comparison, we decrease the remaining possible range of addresses in which the key lies by one half. Halving 256 repeatedly can only be done 8 times: 256, 128, 64, 32, 16, 8, 4, 2, 1. In other words, after at most 8 comparisons, we've either seen the key, or shrunk the range to 1, meaning the key can't be found in the memory. Binary search is 256/8 = 32 times faster than linear search when the key does not exist in the memory, and roughly that much faster when the key exists in the memory too. Yet binary search only requires a slightly smarter controller.

The choice of algorithm makes a big difference in performance for this example—a much bigger difference than is determined by, say, the speed of the comparator being used.

Power Optimization

Power is becoming an important design criteria, both in high-end computing as well as in embedded computing. The unit of power is ***watts***, which represents the energy per second (i.e., joules per second). In high-end computing, like desktop PCs, servers, or video-game consoles, the chips inside a computer consume a lot of power, causing the chips to become very hot. For example, a typical chip inside a PC may consume 60 watts—think about touching a 60-watt light bulb (but don't actually touch one) to understand how hot that is. Designing low-power chips reduces the need for expensive chip cooling methods beyond simple fans in high-end computing, and also reduces the electricity costs, which can be quite significant for companies operating large numbers of computers.

In embedded computing, even simple cooling methods likes fans may not be available—a cell phone does not have a fan (if it did, people might find their tie or scarf getting stuck in that fan). Portable devices might have chips that run at only 1 watt or less.

Furthermore, portable devices typically get their energy from batteries, and thus low-power chips are necessary to extend battery life—especially considering the fact that batteries are not improving fast enough to keep pace with increasing power consumption. By some measures, energy demand per chip is doubling about every three years (going along with Moore's Law). Figure 6.79 plots such energy demands compared to battery energy densities improving at their present rate of only about 8% per year. The increasing gap shown translates to shorter battery lifetimes for a device like a cell phone, or translates to bigger batteries.

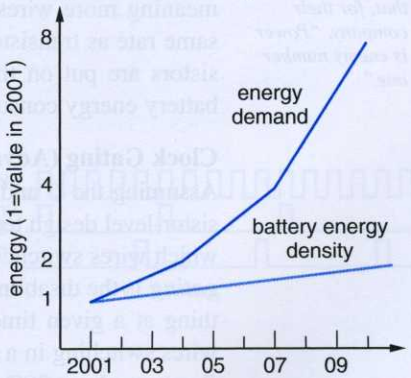


Figure 6.79 Battery energy density is improving more slowly than the increasing energy demands of digital chips.

The most popular IC technology today uses CMOS transistors, and the biggest contributor to power consumption in CMOS is the switching of values from `0` to `1`. The reason for this is that wires aren't perfect, having capacitance (we don't put a capacitor there on purpose—it

is simply a result of the fact that wires aren't perfect conductors of electricity). Switching the wire from `0` to `1` requires charging that capacitor. Switching from `1` back to `0` causes that charge to be discharged to ground. That switching results in power being consumed. This power is known as ***dynamic power***, since this power comes from the changing of signals ("dynamic" means "changing"). Dynamic power consumption of a CMOS wire is proportional to the size of the capacitance (C) of the wire, multiplied by the voltage (V) squared, multiplied by the frequency at which the wire switches (f), namely

$$P = k * CV^2f \quad \text{(equation for CMOS dynamic power consumption)}$$

where k is some constant. Computing the dynamic power of a circuit is achieved by adding up the power computed by the above equation for every wire.

Looking at the above equation, one can clearly see that lowering the voltage will cause the greatest reduction in dynamic power, because of the voltage having a quadratic (squared) contribution to dynamic power. Low-level circuit designers seek to reduce power by creating transistors that operate at the lowest voltage possible, to reduce the V term, and that have the smallest wire capacitance possible, to reduce the C term. Digital designers can therefore choose to utilize gates that operate with a lower voltage.

Unfortunately, lower-voltage gates have a longer delay than higher-voltage gates, resulting in a tradeoff between power and performance.

Another way to reduce the dynamic power consumed by a circuit is to reduce the circuit's clock frequency, which obviously reduces the `f` term for all the clock wires in the circuit, as well as for the many other wires that change on each clock edge (like register wires and the logic connected to those registers' outputs). But again, reducing the clock frequency slows performance, resulting in a tradeoff between power and performance.

The chief technical officer at a major chip design company told me in 2004 that, for their company, "Power is enemy number one."

Contrary to digital design in the 1980s and 1990s, power is a key challenge today. The reason is that IC makers have scaled IC voltage down nearly as low as possible, yet are putting more transistors on each IC every year due to shrinking of transistor sizes, meaning more wires switching on the same IC. And capacitance isn't decreasing at the same rate as transistor sizes. The result is that an IC consumes more power as more transistors are put on the IC, which can result in problems due to too much heat and fast battery energy consumption.

Clock Gating (Advanced Technique)

Assuming the C and V terms have been reduced to the greatest extent possible using transistor-level design techniques, power can be reduced further by reducing f, the frequency at which wires switch. One method for reducing such power is known as clock gating. ***Clock gating*** is the disabling of the clock signal in regions of the chip that are not computing anything at a given time. Clock gating saves power because a significant percentage of the wires switching in a chip are the wires that distribute the clock to all the registers and flip-flops—perhaps 20%–30% of the power consumption is due to the clock signal switching throughout the chip. Clock gating reduces `f` without slowing the clock frequency itself.

In clock gating, the clock signal is disabled by ANDing the clock signal with an enable signal that is set in the state machine. Recall that a register with parallel load internally reloads the same value from the register's flip-flops back into the flip-flops on a rising clock edge. Preventing the clock edge from appearing keeps the same values in the flip-flops, yielding the same net result—the register's contents don't change.

Example

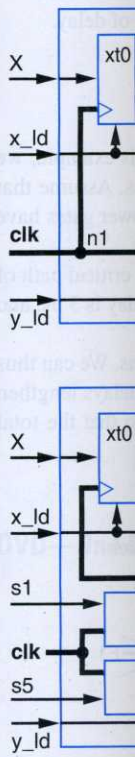


Figure 6.80 C
are only load
reduces the s

Clock gating is not something that digital designers typically do themselves. Rather, modern synthesis tools may allow us to specify clock enable and disable using special commands in each state. However, adding a gate on a clock signal delays the clock signal, resulting in clock signals in different parts of the circuit being slightly different from one another, an effect known as ***clock skew***. The tools therefore automatically perform timing analysis to ensure that the clock skew does not change overall circuit behavior. Furthermore, putting gates on a clock signal can reduce the sharpness of the clock edges, and so tools may use special gates. Nevertheless, the technique is widely used by low-power tools in practice. The next example illustrates clock gating.

Example 6.23 Serial FIR filter with clock gating to reduce power

We designed a serial FIR filter in Example 6.21. A five-state FSM controlled the datapath. The state machine loaded the three *xt* registers only in the first state, state *S1*, and loaded the *yreg* register only in the last state, state *S5*. Yet, the design routed the clock signal to all four registers utilizing four wires, labeled `n1-n4` in Figure 6.80(a). Notice from the timing diagram at the top of the figure that `n1-n4` change identically as the clock signal changes, and remember that every such change consumes dynamic power.

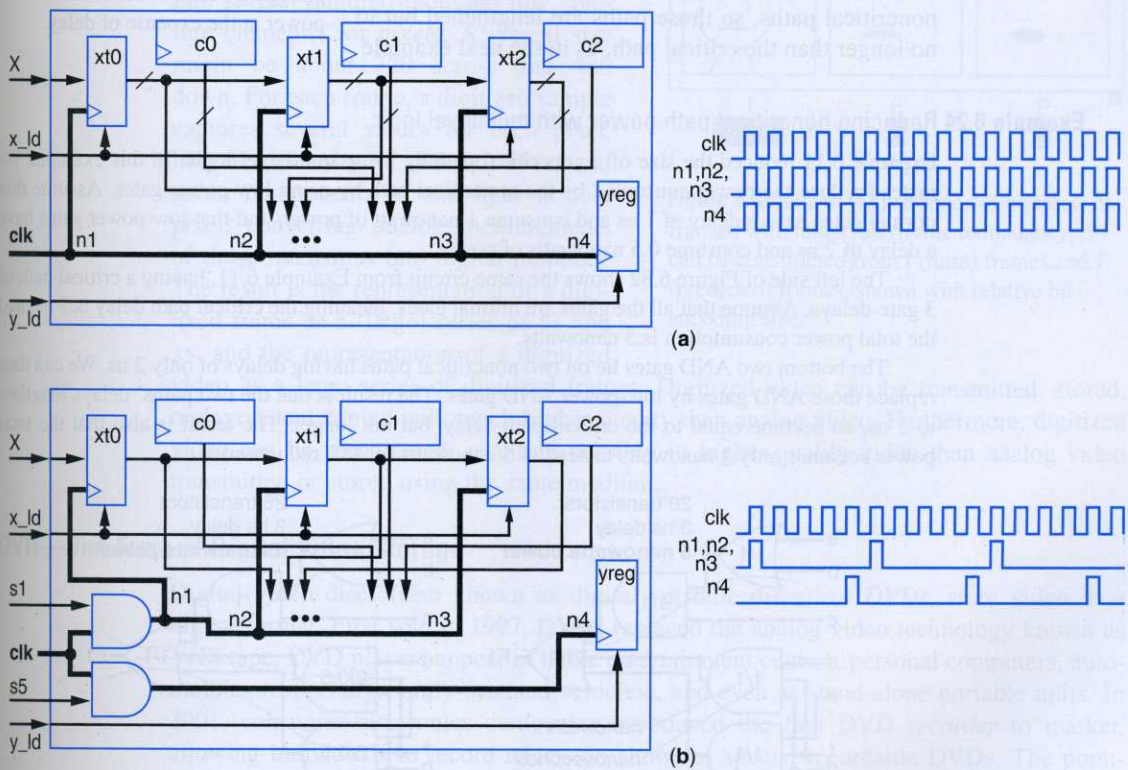


Figure 6.80 Clock gating: (a) the clock signal switches every cycle on all the heavily bolded wires, but the *xt* registers are only loaded in state *S1*, and the *yreg* in state *S5*—so most of the clock switching is wasted; (b) gating the clock reduces the switching on the clock wires.

Figure 6.80(b) shows a design using clock gating. The controller gates the clock to the *xt* registers by setting `s1` to `0` in all states but *S1*. Likewise, the controller gates the clock to the *yreg* register by setting *s5* to `0` in all states but *s5*. Notice the significant decrease in signal switching on the clock's wires `n1`–`n4`, shown at the bottom of Figure 6.80.

Low-power gates on noncritical paths

Not all gates are equally fast. Engineers that build gates from transistors can make a gate faster by increasing the size of the gate's transistors, or by operating the gate at a higher voltage, or by other means. Thus, one two-input AND gate might have a 1 ns delay, while another two-input AND gate might have a 2 ns delay. The latter AND may consume less power, due to its smaller size or lower voltage.

To reduce the power consumed by a circuit, the entire circuit can use low-power gates, at the expense of slower performance, as illustrated in Figure 6.81.

Alternatively, low-power gates can be put only on noncritical paths, so those paths are lengthened but to no longer than the critical path, as in the next example.

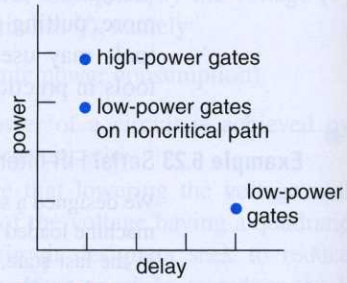


Figure 6.81 Using low-power gates can reduce some power without changing delay, or reduce more power at the expense of delay.

Example 6.24 Reducing noncritical path power with multilevel logic

Example 6.11 reduced the size of a noncritical path by using multilevel logic. In this example, we instead reduce the power consumed by the noncritical path by using low-power gates. Assume that normal gates have a delay of 1 ns and consume 1 nanowatt of power, and that low-power gates have a delay of 2 ns and consume 0.5 nanowatts of power.

The left side of Figure 6.82 shows the same circuit from Example 6.11, having a critical path of 3 gate-delays. Assume that all the gates are normal gates, meaning the critical path delay is 3 ns, and the total power consumption is 5 nanowatts.

The bottom two AND gates lie on two noncritical paths having delays of only 2 ns. We can thus replace those AND gates by low-power AND gates. The result is that the two paths' delays lengthen to 3 ns, so become equal to the critical path delay, but not longer. The result is also that the total power becomes only 4 nanowatts instead of 5 nanowatts (a 20% reduction).

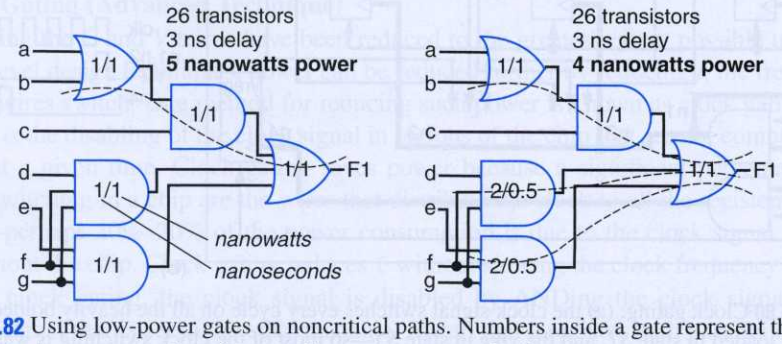


Figure 6.82 Using low-power gates on noncritical paths. Numbers inside a gate represent the gate's delay in nanoseconds and the gate's power consumption in nanowatts.

6.7 PRODUCT PROFILE: DIGITAL VIDEO PLAYER/RECORDER

Digital Video Overview

In the 1990s, the digitization of video became practical due to faster, smaller, and lower-power digital circuits. Previously, video was largely captured, stored, and played using analog methods. Digitized video works by sampling an analog video signal and transforming the samples to digital values. Such digitization is similar to the audio digitization example from Figure 1.1, but with some additional work.

A video is actually a series of quickly displayed still pictures, known as ***frames***, as shown in Figure 6.83(a). One second of video might consist of about 30 frames—the human eyes and brain see such a rapid sequence of frames as a smooth, continuous video.

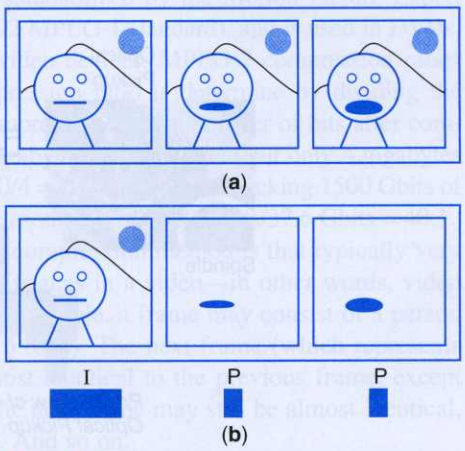


Figure 6.83 Video: (a) is a series of pictures, or frames, with much interframe redundancy, (b) can be constructed from I (intra) frames and P (predicted) frames, shown with relative bit encoding sizes.

A digital display may be divided into several hundred thousand tiny "picture elements," or ***pixels***. A typical size might be about 720 across and 480 down. For each frame, a digitized sample captures several values for each pixel, like the intensity of the red, blue, and green components of the light at that pixel, converting analog measurements of those intensities into digital numbers. The result is the representation of a digitized frame as a (large) series of `0`s and `1`s, and the representation of a digitized video as a large series of digitized frames. Digitized video can be transmitted, stored, replayed, and copied with much higher-quality than analog video. Furthermore, digitized video can be compressed, resulting possibly in higher-quality video than analog video transmitted or stored using the same medium.

DVD—One Form of Digital Video Storage

Digital video discs (also known as digital versatile discs), or ***DVDs***, store video in a digital format. First sold in 1997, DVDs replaced the analog video technology known as VHS tape. DVD players appear in home entertainment centers, personal computers, automobiles (especially family-oriented vehicles), and even as stand-alone portable units. In 2001, consumer electronics companies introduced the first DVD *recorder* to market, allowing individuals to record television shows to special recordable DVDs. The popularity of DVDs compared to the previously popular analog-based VHS technology stems from several advantages, including better-quality video, no deterioration in video quality over time, and the ability to jump directly to particular parts in a video without having to sequentially forward or rewind.

DVDs store large amounts of data on a thin reflective layer of metal. Although the metal layer within a DVD looks flat from our perspective, there are actually billions of tiny pits on the metal layer that store the data. These pits, or lack of pits (called *lands*), store the binary data on the DVD. Figure 6.84 shows how a DVD player reads the information off a DVD. Using a very precise laser, the laser's light is focused onto the metal layer within the DVD. The metal layer reflects the light onto an optical sensor that can detect whether the light is reflected off of a pit or a land. By detecting the different regions, the optical sensor creates a stream of binary values as it reads the DVD.

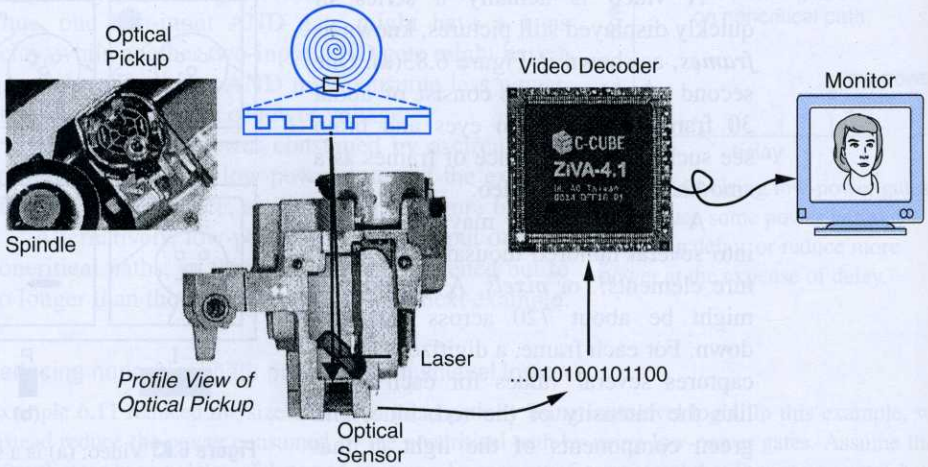


Figure 6.84 How a DVD player reads a DVD. The DVD player's optical pickup element shines a laser on the surface of the DVD. The DVD reflects the laser back to an optical sensor, and the optical sensor uses the intensity of the reflected laser to output the sequence of `0`s and `1`s stored on the DVD. A video decoder circuit converts the binary data to a sequence of frames that humans interpret as a moving picture.

The DVD's binary data is organized into a series of tracks that spiral outward from the center of the DVD. As the DVD player is reading the data, the laser and optical sensor must slowly move outward from the center of the DVD to the outer edge. If a DVD is dual-layered, the data on the disk's second layer is stored in a spiral that moves from the disk's outer to inner edge. The motivation for the second layer's reverse spiral is to prevent the laser and optical sensor from needing to reposition itself to the center of the disk after focusing on the second layer during a layer change. (You may have noticed a DVD pause momentarily at a certain point in a movie during a layer change.)

A single-layer single-sided DVD can store 4.7 gigabytes of data (meaning 37.6 gigabits), but that amount is not enough for a movie unless the data is compressed. Consider a video with a resolution of 720 pixels by 480 pixels, using 24 bits of information per pixel, and displayed at 30 frames per second. One frame would require 720*480*24 = 8,294,400 bits, or about 8 Mbits. One second of video, or 30 frames, would require 30*8,294,400 = 248,832,000 bits, or about 250 Mbits. A 100-minute movie would thus require about 250 Mbits/sec * 100 min * 60 sec/min = 1500 Gbits. But a DVD can only hold 37.6 Gbits. To store a movie, a DVD must store the video in a compressed format.

A DVD is only one of many different digital video storage media. Digitized video may be stored on any storage media capable of storing `0`s and `1`s in some form, such as on tape (used in many digital video cameras), on a flash memory (used in digital cameras and cell phones with video recording capability), on a CD, or on a computer hard drive. All such media are typically still quite limited and thus require compression methods.

MPEG-2 Video Encoding—Sending Frame Differences Using I-, P-, and B-Frames

MPEG-2 video compression was defined and standardized by the Motion Picture Expert Group in 1994 (as an improvement over the 1992 MPEG-1 standard), and is used in DVDs, digital television, and numerous other digital video devices. MPEG-2 compression ratios range from 30:1 to 100:1 or more. The compression ratio is determine by dividing the number of bits of the digitized video before compression, by the number of bits after compression. So if a digitized video requires 400 gigabytes uncompressed but only 4 gigabytes compressed, the compression ratio would be 400/4 = 100:1. Note that packing 1500 Gbits of a movie into 37.6 Gbits would require a compression ratio of 1500 Gbits/37.6 Gbits = 40:1.

The key observation leading to MPEG-2's compression method is that typically very little difference exists between two successive frames in a video—in other words, video typically has much interframe redundancy. For example, a frame may consist of a person standing in front of a mountain, as in Figure 6.83(a). The next frame (which represents perhaps 1/30th of a second later) may be almost identical to the previous frame, except that the person's mouth has opened slightly. The next frame may still be almost identical, with the person's mouth opened slightly more. And so on.

Therefore, MPEG-2 does not merely encode each frame as a distinct picture. Instead, to take advantage of the interframe redundancy, MPEG-2 may choose to encode each frame as one of the following:

- An ***I-frame***, or intracoded frame, is a complete picture.
- A ***P-frame***, or predicted frame, is a frame that merely describes the difference between the current frame and the previous frame. Thus, to derive the picture for this frame, one must combine the P-frame with the previous frame.

For example, Figure 6.83(b) shows P-frames that contain only the differences from the previous frame. A P-frame will obviously require fewer bits than an I-frame. Example frame sizes might be about 8 Mbits for an I-frame, but only 2 Mbits for a P-frame. Thus, instead of representing 30 frames as 30 complete pictures (30 I-frames), a compression method might represent those frames using the following sequence of frames: **I** P P P P P P P P P P P P P P **I** P P P P P P P P P P P P P P. The compression ratio in this example would thus be 8 Mbits * 30 / (2 * 8 Mbits + 28 * 2 Mbits) = 240 / 72 = 3.3:1. Obviously, a picture created by combined predicted frames with a previous frame won't be a perfect representation of the original picture, especially if there is a lot of motion in the video. MPEG-2 thus trades off some quality for compression.

To achieve even further reductions, MPEG-2 uses a third frame type:

- A ***B-frame***, or bidirectional predicted frame, is a frame that can store differences from previous and *future* frames.

B-frames can thus be even smaller than P-frames. An example B-frame size might be just 1 Mbit.

Example 6.25 Computing compression ratios involving I-, P-, and B-frames

Assume a 30-frame MPEG-2 sequence has the following frame sequence: **I** B B P B B P B B P B B P B B **I** B B P B B P B B P B B P B B. Assume average frame sizes of 8 Mbits for I-frames, 2 Mbits for P-frames, and 1 Mbit for B-frames. Compute the compression ratio.

The compression ratio in this example would be 8 Mbits * 30 / (2 * 8 Mbits + 8 * 2 Mbits + 20 * 1 Mbits) = 240 / 52 = 4.6 : 1.

The example sequence of frames is in fact fairly typical for MPEG-2 video, with I-frames occurring about every 12–15 frames.

MPEG-2 video encoders may seek to create about 30 frames per second. With hundreds of thousands of pixels per frame that must be compared with another frame, MPEG-2 encoding requires a large amount of computation to determine which frames should be I, P, and B, and what should be the values for the P- and B-frames. Furthermore, much of that computation will consist of the *same* computation performed between corresponding regions of two frames. Thus, many MPEG-2 encoders utilize custom digital circuits to parallelize those computations at the expense of more hardware size. For instance, Example 6.20 built a sum-of-absolute-differences circuit using more parallelism than in Example 5.9, at the expense of a larger circuit size. Such a circuit would be useful in a video encoder needing to quickly determine whether a frame should be encoded as a P- or B-frame, or instead should be encoded as an I-frame. Additional circuits might compute the actual values of P- and B-frames.

Likewise, an MPEG-2 video decoder might use circuits to quickly recompose I-, P-, and B-frames back into full-picture frames—although decoding MPEG-2 video is easier than encoding because the actual determination of P- and B-frame contents is only done during encoding; decoding merely needs to combine P- and B-frames with their surrounding frames.

Transforming to the Frequency Domain for Further Compression

DCT—Discrete Cosine Transform

We saw in the previous section that sending a frame (P or B) that is just the difference from a previous or future frame can result in some compression. However, the compression ratios achieved were only about 4:1. Recall earlier that a DVD needs perhaps a 40:1 compression ratio to store a full-length movie. Thus, further compression is needed.

MPEG-2 therefore further compresses each I-, P- and B-frame individually. The compression method involves applying what is known as a discrete cosine transform to 8x8 blocks of pixel values within each frame. The discrete cosine transform is also used in the well-known JPEG standard for compressing still images, like those in a digital camera. The ***discrete cosine transform***, or ***DCT***, transforms information from the spatial domain to the frequency domain. (The DCT is similar to another popular technique known as the fast Fourier transform, or FFT, also used for translating to the frequency domain.)

Translating to the frequency domain is a powerful concept, which is widely used in digital signal processing. To understand this concept, consider wanting to digitally store the analog signal shown in Figure 6.85, using the fewest bits possible. The signal is a 1 Hz

cosine wave with an amplitude of 10. To store the signal digitally, we could sample the signal at frequent intervals, perhaps every millisecond, and record the measured signal value as a binary number, perhaps 8 bits wide. One second would thus result in 1000 * 8 = 8000 bits. On the other hand, we could just store the fact that the signal is a cosine wave with a frequency of 1 Hz and an amplitude of 10. If we store each of those numbers as an 8-bit value, then we only need to store 8 + 8 = 16 bits. Sixteen bits is far less than 8000 bits.

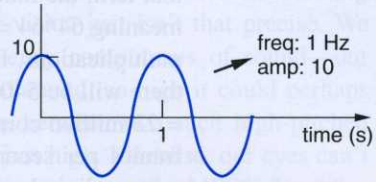


Figure 6.85 Digitizing signals by translating to the frequency domain.

Of course, not all signals that we want to digitize are simple cosine waves. But—and this is the key idea underlying frequency domain representation—*we can approximate any original signal as a sum of cosine waves of different frequencies and amplitudes*. If we break the original signal into small regions, we obtain even better approximation. For example, we might approximate one region as the sum of a 1 Hz cosine wave of amplitude 5 plus a 2 Hz cosine wave of amplitude 3. We might approximate another region as the sum of 50 different cosine waves of different frequencies and amplitudes. The smaller the region we consider, and the more different cosine wave frequencies we consider, the more accurate will be our approximation to the real signal.

Rather than storing the actual frequencies along with the amplitudes of the cosine waves, we could instead decide only to consider using particular frequencies, such as: 1 Hz, 2 Hz, 4 Hz, 8 Hz, 16 Hz, and so on. Then, we can simply send the amplitudes of those particular cosine waves: (5, 3, 0, 0, 0, ...). Let's refer to these amplitudes as coefficients.

The DCT in MPEG-2 converts an input 8x8 block, whose values represent pixel intensities, to an 8x8 block representing the coefficients of predetermined "frequencies." In the video domain, each frequency represents a different block pattern, with low frequency being an almost constant pattern and high frequency being a changing pattern (like a checkerboard). The DCT determines a set of coefficients such that adding the predetermined patterns together with each pattern multiplied by its coefficient yields one resulting pattern very similar to the original input block.

The equation for a two-dimensional DCT applied to an 8x8 block of numbers is:

$$F(u,\ v) = \frac{1}{4}C(u)C(v)\sum_{x=0}^{8}\sum_{y=0}^{8}D[x,\ y]\cos\left(\frac{\pi(2x+1)u}{16}\right)\cos\left(\frac{\pi(2y+1)v}{16}\right)$$

$$C(h) = \begin{cases} \dfrac{1}{\sqrt{2}} & h = 0 \\ 1, & otherwise \end{cases}$$

The input is an 8x8 block, $D[x, y]$. The output is another 8x8 block, with $F(u,v)$ computing the coefficient at row u, column v for the output block.

An MPEG-2 encoder may utilize custom digital circuits for fast DCT computation. Notice that computing each coefficient requires evaluating the rightmost term (let's call that term the inner term) 64 times, and that must be done for each of the 64 coefficients, meaning 64*64 = 4096 evaluations of the term. And that inner term itself requires several multiplications. Furthermore, the DCT operates on 8x8 blocks, but in a 720x480 I-frame there will be 5400 such blocks. Thus, the DCT for one I-frame could require 5400*4096 = 22 million computations of the inner term. And that encoding may have to occur at 30 frames per second. You can begin to see why an MPEG-2 encoder may need to use custom digital circuits to compute the DCT quickly, using extensive parallelism and pipelining to obtain the necessary performance.

The DCT computation can be sped up further by precomputing the cosine terms of the inner term. Notice that the DCT computes two cosines based on the input values of u and x and the input values of v and y. However, because the DCT operates on 8x8 blocks, the variables u, v, x, and y only range in value from 0 to 7. Therefore, we can precompute the 64 possible cosine values needed for the DCT computation and store those values in an 8x8 table, which may be programmed into a ROM. We can then rewrite the DCT transform as follows:

$$F(u,\ v) = \frac{1}{4}C(u)C(v)\sum_{x=0}^{8}\sum_{y=0}^{8}D[x,\ y]\cos[x,\ u]\cos[y,\ v]$$

Using a ROM to store the precomputed cosine values speeds up the computation of the inner term of the DCT.

Quantization

Translating to the frequency domain using the DCT does not directly perform compression—we merely converted an input 8x8 block to an output 8x8 block. That output 8x8 block represents amplitudes of particular cosine wave frequencies. We can achieve compression by rounding those amplitudes, such that we use fewer bits to represent the amplitudes. For example, suppose we use 8 bits to represent the amplitude, meaning we can represent amplitudes ranging from 0 to 255. Suppose we only represent even amplitudes, meaning 2, 4, ..., 254. In that case, we can drop the lowest-order bit, in the representation of the amplitude, resulting in only 7 bits. The decoder would merely append a 0 to the 7-bit number to obtain an 8-bit number again. For example, the 8-bit number 00001111 would be compressed to the 7-bit number 0000111 with an implicit 0 in the eighth bit. The decoder would expand that 7-bit number back to the 8-bit number 00001110—notice that the decoded number is slightly different than the original, being 14 rather than the original 15 (an example of why MPEG-2 compression loses some image quality). We could take this rounding concept further, only representing amplitudes that are multiples of 4 (thus dropping the two lowest-order bits, yielding a 6-bit representation), or are multiples of 8 (dropping the three lowest-order bits, yielding a 5-bit representation). 00001111 might be represented as 00001 with three implicit 0s, thus decoded back to 00001000. The decoded number of 8 is different from the original number 15 due to the rounding.

The rounding described above, achieved by dropping low-order bits to achieve compression, is known as ***quantization***. Notice the tradeoff—more rounding yields more compression, at the expense of accuracy. Fortunately, *humans don't notice such rounding in the high-frequency components of the picture*—our vision just isn't that precise. We also don't notice minor differences in the high-frequency components of sound—our hearing isn't that precise. Think of a very high-pitched sound, so high it could perhaps break glass. You probably couldn't tell the difference between two such high-pitched sounds of slightly different frequencies—they are both just high. Likewise, our eyes can't detect slight rounding of color values in a highly complex scene. So MPEG-2 applies quantization more aggressively on the DCT output block's high-frequency coefficients than on the low-frequency coefficients.

After quantization, the 64 values in the 8x8 block are treated as a list of 64 numbers. Those 64 numbers are then run-length encoded. ***Run-length encoding*** is a compression method that reduces consecutive occurrences of zeros by a number indicating the number of consecutive zeros rather than representing those zeros themselves. For example, consider wanting to represent the following 5 numbers: 0, 0, 0, 0, 24. If each value is 6 bits, the 5 numbers require 5*6 = 30 bits. On the other hand, we could just send a pair of numbers, the first indicating the number of leading zeros, the second indicating the nonzero number. So 0, 0, 0, 0, 24 would be encoded as (4, 24)—4 leading zeros, followed by the number 24. If each value is 6 bits, the run-length encoded version requires only 2*6=12 bits. Any sequence of numbers could similarly be replaced by a sequence of number pairs, each pair replacing a sequence of zeros and a number. The sequence 0, 0, 0, 0, 24, 0, 0, 8, 0, 0, 0, 0, 0, 0, 16 could thus be replaced by three pairs: (4, 24), (2, 8), (6, 16), reducing the number of bits from 15*6=90 down to 6*6=36 bits. Note that the number of zeros at the beginning of the sequence or in between nonzero numbers may be zero, and the last number may be zero. For example, the sequence 2, 0, 0, 63, 2, 0, 0, 0, 0, 0 could be encoded as (0,2), (2,63), (0,2), (4,0).

Run-length encoding achieves good compression only if there are many 0s in the sequence of numbers. Fortunately, the nature of the DCT leads to many 0 numbers (not all cosine frequencies are needed to approximate a signal region, so those frequencies will have 0 coefficients), especially after quantization (many coefficients are just small numbers, which become 0 during quantization). Thus, applying run-length encoding after quantization leads to further compression.

Example 6.26 Computing compression ratios involving quantization and run-length encoding

Continuing Example 6.25, assume that the 30-frame MPEG-2 sequence has the same frame sequence and average sizes as that example, but that each frame is further compressed by DCT conversion to the frequency domain followed by quantization and run-length encoding. Assume the DCT output block consists of 64 8-bit numbers, that quantization reduces the average number size to 5-bit numbers, and that run-length encoding reduces the resulting number sequence size to 30% of its original size.

The compression ratio would be 8 Mbits * 30 / 5/8 * 0.30 *(2 * 8 Mbits + 8 * 2 Mbits + 20 * 1 Mbits) = 240 / 9.7 = 25:1.

Huffman Coding

After run-length encoding, each block consists of a sequence of numbers. Some numbers will occur in that sequence more frequently than others. ***Huffman coding*** is a method of reducing the number of bits required to represent a set of values, by creating shorter encodings for the frequently occurring values, and longer encodings for the less frequent value.

Huffman coding, a form of encoding known as entropy encoding, is another powerful concept in digital data compression. Suppose you wish to represent an original sequence of 16 numbers 0, 3, 3, 31, 0, 3, 5, 8, 9, 7, 15, 14, 3, 0, 3, 0. Assuming 5 bits per number, a straightforward binary encoding would be: `00000 00011 00011 11111 00000 00011 00101`, and so on, for a total of 16*5 = **80 bits**. We can reduce this total by first observing that there are only 9 unique symbols: 0, 3, 5, 7, 8, 9, 14, 15, and 31. We really only need 4 bits to uniquely identify each symbol. We could thus assign the nine unique symbols to 4-bit encodings using the following definitions: 0=`0000`, 3=`0001`, 5=`0010`, 7=`0011`, ..., 31=`1001` (note that the encodings are no longer the binary number representations of the original numbers). Thus, the original sequence of numbers (0, 3, 3, 31, 0, 3, 5, ...) would be encoded as `0000 0001 0001 1001 0000 0001 0010` etc., for a total of 16*4 = **64 bits**. The key observation here is that we can encode numbers using any arbitrary unique bit patterns we desire, as long as the encoder and decoder are both aware of the encoding definitions.

We can take this definition concept a step further, by using encodings of different lengths. Observing that 3 and 0 occur more frequently than the other numbers, we might give 3 and 0 shorter encodings. So we might create the following encoding definitions: 0=`00`, 3=`10`, 5=`010`, 7=`0110`, 8=`0111`, 9=`1100`, 14=`1101`, 15=`1110`, 31=`1111`. How these definitions were created is just beyond the scope of this discussion, though it's really not hard to learn. Notice that the encodings are such that the shorter encodings do not appear at the left of any of the longer encodings. For example, `00` does not appear at the left of any of the longer encodings, like `010`, `0110`, `0111`, etc. This feature allows the decoder to know when it has reached the end of the code word—when the decoder has seen `00`, it knows it has found an encoded 0 (because no other encoding starts with `00`); when it sees `10`, it knows it has found a 3 (because no other encoding starts with `10`). But when the decoder sees `01`, it must look at the next bit, and if it sees `010`, it knows it has found a 5 (because no other encoding starts with `010`). Using this variable-length encoding scheme, the original sequence (0, 3, 3, 31, 0, 3, 5, ...) would be encoded as `00 10 10 1111 00 10 010` etc. We have inserted the spaces just for readability; the actual encoding would just be `0010101111 0010010` etc. The total number of bits would be 4 * 2 (for the four 0s, encoded with the two bits `00`) + 5 * 2 (for the five 3s, encoded with the two bits `10`) + 1*3 (for the one 5, encoded with the three bits `010`) plus 6*4 (for the six remaining numbers 31, 8, 9, 7, 15, and 14, each encoded as 4 bits), totaling **45 bits**—much reduced from the original 80 bits required by the straightforward binary encoding.

Huffman coding achieves good compression when some numbers occur much more frequently than other numbers in the sequence of numbers to be encoded. Fortunately, this is indeed the case after DCT, quantization, and run-length tasks are performed on a block of a frame. For example, there may be plenty of 0s, 1s, 2s, etc., and fewer occurrences of higher numbers.

Example 6.27 Computing compression ratios involving Huffman coding

Continuing Example 6.26, assume that pairs of numbers after quantization and run-length encoding are Huffman coded, and that such encoding reduces the number of bits by 50%.

The compression ratio would thus be 240 / 0.50* 9.7 = 50:1.

Summary

Summarizing MPEG-2 video encoding:

- The use of I-, P-, and B-frames achieves compression by not resending redundant information of successive frames, but rather just sending the differences.
- The DCT transforms 8x8 blocks of frames to the frequency domain, which doesn't achieve compression itself, but rather enables compression in the next steps.
- Quantization achieves further compression by reducing the number of bits needed to represent the DCT coefficients, through rounding.
- Run-length encoding achieves further compression by replacing sequences of zero coefficients by a number indicating the number of such zeros.
- Huffman coding achieves further compression by encoding frequently occurring coefficient numbers with shorter encodings than less frequently occurring coefficient numbers.

The sequence of steps is shown graphically in Figure 6.86.

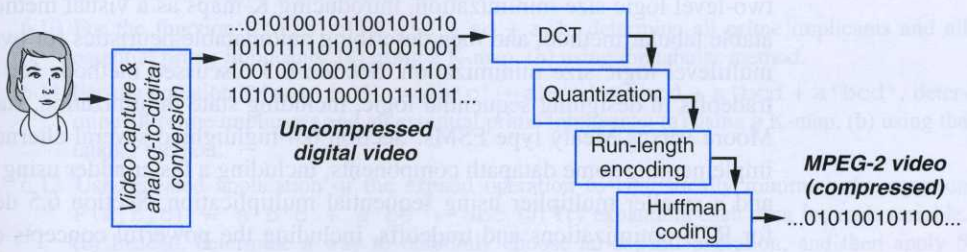


Figure 6.86 MPEG-2 video compression encoding overview.

Our example compression ratio calculations yielded a ratio of about 50:1. In fact, the compression ratio can be varied by varying each of the above steps. We can use fewer I-frames to achieve even higher compression at the cost of degraded video quality, or more I-frames for improved video quality at the cost of more bits. Likewise, we can vary the amount of quantization to trade off quality and compression ratio. Because a typical movie will have some slow-changing scenes and other rapidly changing scenes, and some complex colored frames and other simpler frames, the compression ratio for different parts of a video may actually vary. Notice the permeating presence of *tradeoffs* (primarily between quality and compression ratio) throughout MPEG-2 encoding.

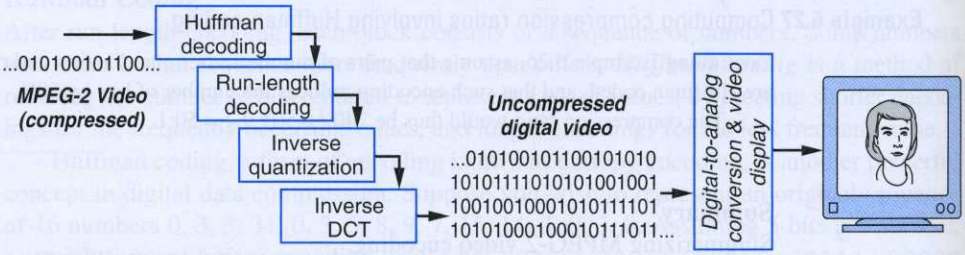


Figure 6.87 MPEG-2 video decoding overview.

An MPEG-2 decoder merely needs to apply the above steps in reverse, as illustrated in Figure 6.87, to convert an MPEG-2 stream of bits back into a series of pictures, or video.

Clearly, MPEG-2 encoding and decoding require a lot of computations performed at speeds fast enough to create smooth-looking, good-quality video. Custom digital circuits can help achieve those required speeds.

▶ 6.8 CHAPTER SUMMARY

Section 6.1 introduced the idea that sometimes a particular design criteria can be improved without hurting other criteria (optimization), but usually improving one criteria is done at the expense of another criteria (tradeoff). Section 6.2 discussed the problem of two-level logic size minimization, introducing K-maps as a visual method and an automatable tabular method, and then describing automatable heuristics for two-level as well as multilevel logic size minimization. Section 6.3 discussed methods for optimization and tradeoffs in designing sequential logic, including state minimization, state encoding, and Moore versus Mealy type FSMs. Section 6.4 highlighted several alternative methods for implementing some datapath components, including a faster adder using carry-lookahead, and a smaller multiplier using sequential multiplication. Section 6.5 described methods for RTL optimizations and tradeoffs, including the powerful concepts of pipelining and concurrency as means of achieving good performance, which is a key purpose of digital design. The section also described the RTL methods of component allocation, operator binding, and operator scheduling. Section 6.6 briefly surveyed some higher-level methods, including the general idea of serial versus concurrent computation, and the selection of efficient algorithms. The section also introduced some basic concepts of power reduction, including clock gating, and the use of low-power gates.

As can be seen from this chapter, there are many methods for improving designs. Yet, this chapter just scratched the surface of such methods. An entire multibillion-dollar-per-year industry exists that specializes in making automated tools for converting behavioral descriptions of desired system functionality into highly optimized circuit implementations—that industry is known as electronic design automation (EDA) or as computer-aided design (CAD). This chapter hopefully gave enough exposure at least to understand the basic idea behind circuit optimization at various levels of design abstraction, ranging from the gate level up to the RTL level and beyond.