Contents

Chapter 1

The Digital Abstraction

1.1 The Digital Revolution

Digital systems are pervasive in modern society. Some uses of digital techonolgoy are obvious - such as a personal computer or a network switch. However, there are also many hidden applications of digital technology. When you speak on the phone, in almost all cases your voice is being digitized and transmitted via digital communications equipment. When you play a music CD, the music, recorded in digital form, is processed by digital logic to correct errors and improve the audio quality. When you watch TV, the image is processed by digital electronics to improve picture quality (and for HDTV the transmission is digital as well). If you have a TiVo (or other PVR) you are recording video in digital form. DVDs are compressed digital video recordings. When you play a DVD you are digitally decompressing and processing the video. Most radios - cell phones, wireless networks, etc... - use digital signal processing to implement their modems. The list goes on.

Most modern electronics uses analog circuitry only at the edge - to interface to a physical sensor or actuator. As quickly as possible, signals from a sensor (e.g., a microphone) are converted into digital form and all real processing, storage, and transmission of information is done in digital form. The signals are converted back to analog form only at the output - to drive an actuator (e.g., a speaker).

Not so long ago the world was not so digital. In the 1960s digital logic was found only in expensive computer systems and a few other niche applications. All TVs, radios, music recordings, and telephones were analog.

The shift to digital was enabled by the scaling of integrated circuits. As integrated circuits become more complex, more sophisticated signal processing became possible. This signal processing was only possible using digital logic. The complexity of the modulation, error correction, compression, and other techniques were not feasible in analog technology. Only digital logic with its ability to perform a complex computation without accumulating noise and its

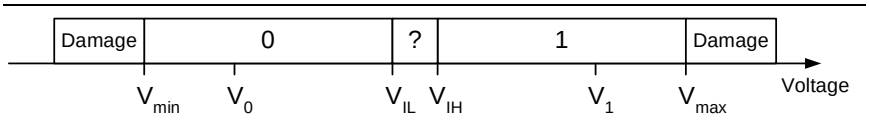


Figure 1.1: Encoding of two symbols, 0 and 1, into voltage ranges. Any voltage in the range labeled 0 is considered a 0 symbol. Any voltage in the range labeled 1 is considerd to be a 1 symbol. Voltages between the 0 and 1 ranges (the ? range) are undefined and represent neither symbol. Voltages outside the 0 and 1 ranges may cause permanent damage to the equipment receiving the signals.

ability to represent signals with arbitrary precision could implement this signal processing.

In this book we will look at how the digital systems that form such a large part of all of our lives function and how they are designed.

1.2 Digital Signals

Digital systems store, process, and transport information in digital form. That is the information is represented as discrete symbols that are encoded into ranges of a physical quantity. Most often we represent information with just two symbols, "0" and "1", and encode these symbols into voltage ranges as shown in Figure 1.1. Any voltage in the ranges labeled "0" and "1" represents a "0" or "1" symbol respectively. Voltages between these two ranges, the region labeled "?" are undefined and represent neither symbol. Voltages outside the ranges, below the "0" range or above the "1" range are not allowed and may permanently damage the system if they occur. We call signal encoded in the manner shown in Figure 1.1 a *binary* signal because it has two valid states.

Table 1.1 shows the JEDEC JESD8-5 standard for encoding a binary digital signal in a system with a 2.5V power supply. Using this standard, any signal with a voltage between -0.3V and 0.7 volts is considered to be a "0" and a signal with a voltage between 1.7V and 2.8V is considered to be a "1". Signals that don't fall into these two ranges are undefined. If a signal is below -0.3V or above 2.8V, it may cause damage[1].

Digital systems are not restricted to binary signals. One can generate a digital signal that can take on three, four, or any finite number of discrete values. However, there are few advantages to using more than two values and the circuits that store and operate on binary signals are simpler and more robust than their multi-valued counterparts. Thus, except for a few niche applications, binary signals are universal in digital systems today.

Digital signals can also be encoded using physical quantities other than voltage. Almost any physical quantity that can be easily manipulated and sensed

[1] The actual specification for V_{max} is $V_{DD} + 0.3$, where V_{DD}, the power supply, is allowed to vary between 2.3 and 2.7V.

Parameter	Value	Description
V_{min}	-0.3V	Absolute minimum voltage below which damage occurs
V_0	0.0V	Nominal voltage representing logic "0"
V_{OL}	0.2V	Maximum output voltage representing logic "0"
V_{IL}	0.7V	Maximum voltage considered to be a logic "0" by a module input
V_{IH}	1.7V	Minimum voltage considered to be a logic "1" by a module input
V_{OH}	2.1V	Minimum output voltage representing logic "1"
V_1	2.5V	Nominal voltage representing logic "1"
V_{max}	2.8V	Absolute maximum voltage above which damage occurs

Table 1.1: Encoding of binary signals for 2.5V LVCMOS logic. Signals with voltage in $[-0.3, 0.7]$ are considered to be a 0 signals with voltage in $[1.7, 2.8]$ are considered to be a 1. Voltages in $[0.7, 1.7]$ are undefined. Voltages outside of $[-.3, 2.8]$ may cause permanent damage.

can be used to represent a digital signal. Systems have been built using electrical current, air or fluid pressure, and physical position to represent digital signals. However, the the tremendous capability of manufacturing complex systems at low cost as CMOS integrated circuits has made voltage signals universal today.

1.3 Digital Signals Tolerate Noise

The main reason that digital systems have become so pervasive, and what distinguishes them from *analog* systems is that they can process, transport, and store information without it being distorted by noise. This is possible because of the discrete nature of digital information. A binary signal represents either a "0" or a "1". If you take the voltage that represents a "1", V_1, and disturb it with a small amount of noise, ϵ, it still represents a "1". There is no loss of information with the addition of noise, until the noise gets large enough to push the signal out of the "1" range. In most systems it is easy to bound the noise to be less than this value.

Figure 1.2 compares the effect of noise on an analog system (Figure 1.2(a)) and a digital system (Figure 1.2(b)). In an analog system information is represented by an analog votage, V. For example, we might represent temperature (in degrees Fahrenheit) with voltage according to the relation $V = 0.2(T - 68)$. So a temperature of 72.5 degrees is represented by a voltage of 900mV. This representation is continuous; every voltage corresponds to a different temperature. Thus, if we disturb the signal V with a noise voltage ϵ, the resulting signal

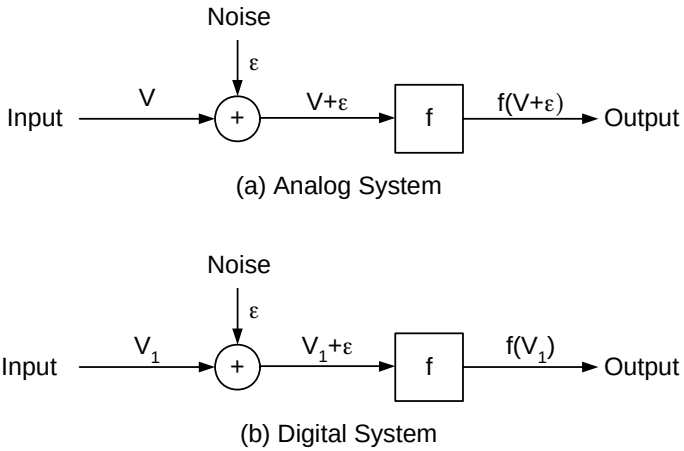


Figure 1.2: Effects of noise in analog and digital systems. (a) In an analog system perturbing a signal V by noise ϵ results in a degraded signal $V + \epsilon$. Operating on this degraded signal with a function f gives a result $f(V + \epsilon)$ that is different from the result of operating on the signal without noise. (b) In a digital system, adding noise ϵ to a signal V_1 representing a symbol, 1, gives a signal $V_1 + \epsilon$ that still represents the symbol 1. Operating on this signal with a function f gives the same result $f(V_1)$ as operating on the signal without the noise.

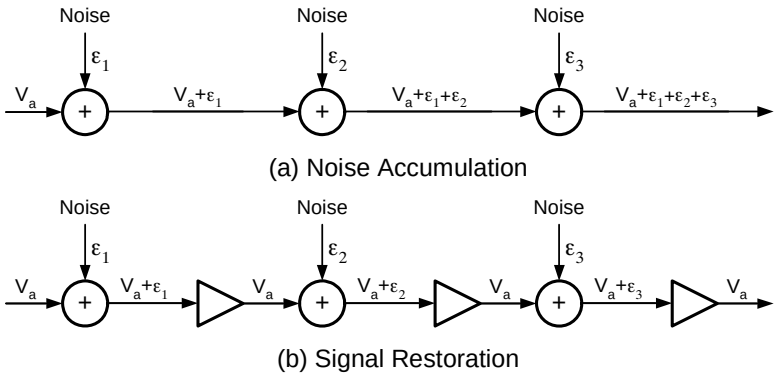


Figure 1.3: Restoration of digital signals. (a) Without restoration signals accumulate noise and will eventually accumulate enough noise to cause an error. (b) By restoring the signal to its proper value after each operation noise is prevented from accumulating.

$V + \epsilon$ corresponds to a different temperature. If $\epsilon = 100$mV, for example, the new signal $V + \epsilon = 1V$ corresponds to a temperature of 73 degrees ($T = 5V + 68$) which is different from the original temperature of 72.5 degrees.

In a digital system, on the other hand, each bit of the signal is represented by a voltage, V_1 or V_0 depending on whether the bit is "1" or "0". If a noise source perturbs a digital "1" signal V_1 for example, as shown in Figure 1.2(b), the resulting voltage $V_1 + \epsilon$ still represents a "1" and applying a function to this noisy signal gives the same result as applying a function to the original signal. Moreover, if a temperature of 72 is represented by a three-bit digital signal with value 010 (see Figure 1.6(c)), the signal still represents a temperature of 72 even after all three bits of the signal are disturbed by noise - as long as the noise is not so great as to push any bit of the signal out of the valid range.

To prevent noise from accumulating to the point where it pushes a digital signal out of the valid "1" or "0" range, we periodically restore digital signals as illustrated in Figure 1.3. After transmitting, storing and retrieving, or operating on a digital signal, it may be disturbed from its nomimal value V_a (where a is 0 or 1) by some noise ϵ_i. Without restoration (Figure 1.3(a)) the noise accumulates after each operation and eventually will overwhelm the signal. To prevent accumulation, we restore the signal after each operation. The restoring device, which we call a *buffer*, outputs V_0 if its input lies in the "0" range and V_1 if its output lies in the "1" range. The buffer, in effect, restores the signal to be a pristine 0 or 1, removing any additive noise.

This capability of restoring a signal to its noiseless state after each operation enables digital systems to carry out complex high-precision processing. Analog systems are limited to performing a small number of operations on relatively

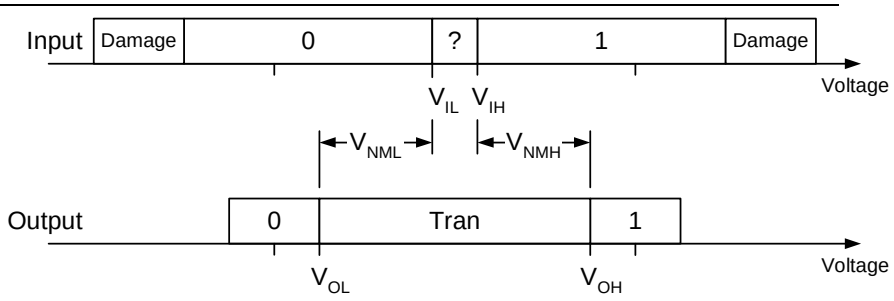


Figure 1.4: Input and output voltage ranges. (Top) Inputs of logic modules interpret signals as shown in Figure 1.1. (Bottom) Outputs of logic modules restore signals to narrower ranges of valid voltages.

low-precision signals because noise is accumulated during each operation. After a large number of operations the signal is swamped by noise. Since all voltages are valid analog signals there is no way to restore the signal between operations. Analog systems are also limited in precision. They cannot represent a signal with an accuracy finer than the background noise level. Digital systems on the other hand can perform an indefinite number of operations and, as long as the signal is restored after each operation, no noise is accumulated. Digital systems can also represent signals of arbitrary precision without corruption by noise.[2]

In practice, buffers, and other restoring logic devices, do not guarantee to output exactly V_0 or V_1. Variations in power supplies, device parameters, and other factors lead the outputs to vary slightly from these nominal values. As illustrated in the bottom half of Figure 1.4, all restoring logic devices guarantee that their 0 outputs fall into a 0 range that is narrower than the input 0 range and similarly for 1 outputs. Specifically, all 0 signals are guaranteed to be less than V_{OL} and all 1 signals are guaranteed to be greater than V_{OH}. To ensure that the signal is able to tolerate some amount of noise, we insist that $V_{OL} < VIL$ and that $V_{IH} < V_{OH}$. For example, the values of V_{OL} and V_{OH} for 2.5V LVCMOS are shown in Table 1.1. We can quantify the amount of noise that can be tolerated as the *noise margins* of the signal:

$$\begin{aligned} V_{NMH} &= V_{OH} - V_{IH}, \\ V_{NML} &= V_{IL} - V_{OL}. \end{aligned} \tag{1.1}$$

While one might assume that a bigger noise margin would be better, this is not necessarily the case. Most noise in digital systems is induced by signal transitions and hence tends to be proportional to the signal swing. Thus, what is really important is the *ratio* of the noise margin to the signal swing, $\frac{V_{NM}}{V_1 - V_0}$

[2]Of course one is limited by analog input devices in acquiring real-world signals of high precision.

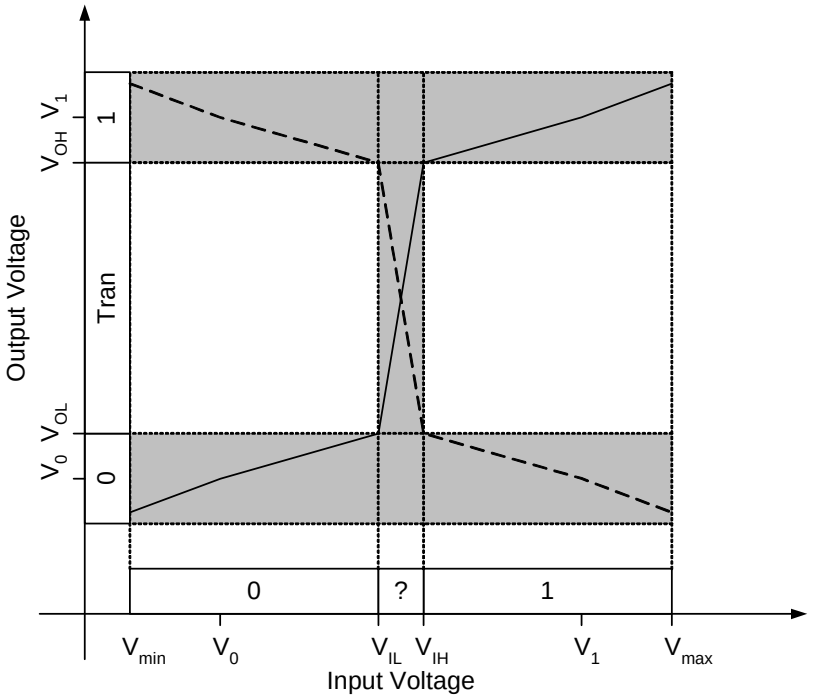


Figure 1.5: DC transfer curve for a logic module. For an input in the valid ranges, $V_{min} \leq V_{in} \leq V_{IL}$ or $V_{IH} \leq V_{in} \leq V_{max}$, the output must be in the valid output ranges $V_{out} \leq V_{OL}$ or $V_{OH} \leq V_{out}$. Thus, all valid curves must stay in the shaded region. This requires that the module have gain > 1 in the invalid input region. The solid curve shows a typical transfer function for a non-inverting module. The dashed curve shows a typical transfer function for an inverting module.

rather than the absolute magnitude of the noise margin. We will discuss noise in more detail in Chapter 5.

Figure 1.5 shows the relationship between DC input voltage and output voltage for a logic module. The horizontal axis shows the module input voltage and the vertical axis shows the module output voltage. To conform to our definition of *restoring* the the transfer curve for all modules must lie entirely within the shaded region of the figure so that a input signal in the valid 0 or 1 range will result in an output signal in the narrower output 0 or 1 range. Non-inverting modules, like the buffer of Figure 1.3 have transfer curves similar to the solid line. Inverting modules have transfer curves similar to the dashed line. In either case, gain is required to implement a restoring logic module. The

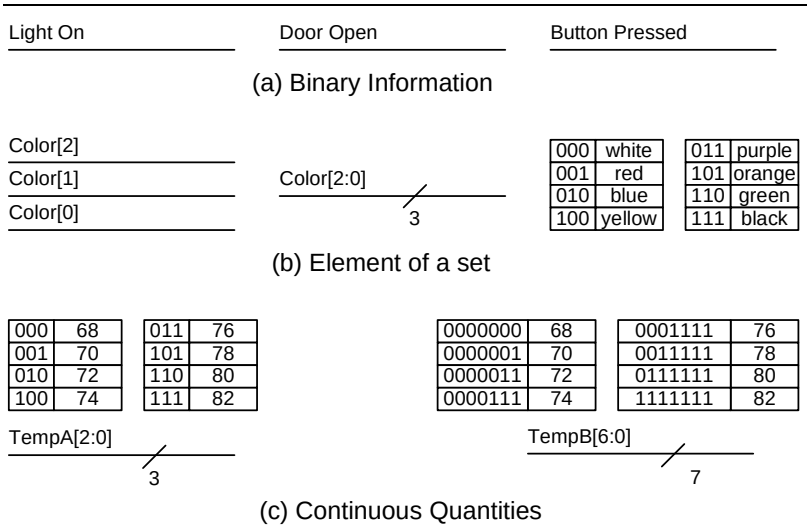


Figure 1.6: Representing information with digital signals. (a) binary-valued predicates are represented by a single-bit signal. (b) elements of sets with more than two elements are represented by a group of signals. In this case one of eight colors is denoted by a three-bit signal `Color[2:0]`. (c) A continuous quantity, like temperature, is *quantized* and the resulting set of values is encoded by a group of signals. Here one of eight temperatures can be encoded as a three-bit signal `TempA[2:0]` or as a seven-bit *thermometer-coded* signal `TempB[6:0]` with at most one transition from 0 to 1.

absolute value of the maximum slope of the signal is bounded by

$$\max \left| \frac{dV_{out}}{dV_{in}} \right| \geq \frac{V_{OH} - V_{OL}}{V_{IH} - V_{IL}}. \tag{1.2}$$

From this we conclude that restoring logic modules must be active elements capable of providing gain.

1.4 Digital Signals Represent Complex Data

Some information is naturally binary in nature and can be represented with a single binary digital signal (Figure 1.6(a)). Truth propositions or predicates fall into this category. For example a single signal can indicate that a door is open, a light is on, a seatbelt is buckled, or a button is pressed.

Often we need to represent information that is not binary in nature: a day of the year, the value and suit of a playing card, the temperature in a

room, a color, etc... We encode information with more than two natural states using a group of binary signals (Figure 1.6(b)). The elements of a set with N elements can be represented by a signal with $n = \lceil \log_2 N \rceil$ bits. For example, the eight colors shown in Figure 1.6(b) can be represented by three one-bit signals, `Color[0]`, `Color[1]`, and `Color[2]`. For convenience we refer to this group of three signals as a single multi-bit signal `Color[2:0]`. In a circuit or schematic diagram, rather than drawing three lines for these three signals, we draw a single line with a slash indicating that it is a multi-bit signal and the number "3" near the slash to indicate that it is composed of three bits.

Continuous quantities, such as voltage, temperature, pressure, are encoded as digital signals by *quantizing* them, reducing the problem to one of representing elements of a set. Suppose for example, that we need to represent temperatures between 68deg F and 82deg F and that it suffices to resolve temperature to an accuracy of 2degF. We quantize this temperature range into eight discrete values as shown in Figure 1.6(c). We can represent this range with binary weighted signals `TempA[2:0]` where the temperature represented is

$$T = 68 + 2\sum_{i=0}^{2} 2^i \texttt{TempA}[i] \tag{1.3}$$

Alternatively we can represent this range with a seven-bit *thermometer-coded* signal `TempB[6:0]`

$$T = 68 + 2\sum_{i=0}^{2} \texttt{TempB}[i] \tag{1.4}$$

Many other encodings of this set are possible. A designer chooses a representation depending on the task at hand. Some sensors (e.g. thermometers) naturally generate thermometer-coded signals. In some applications it is important that adjacent codes differ in only a single bit. At other times cost and complexity are reduced by minimizing the number of bits needed to represent an element of the set. We will revisit digital representations of continuous quantities when we discuss numbers and arithmetic in Chapter 10.

Example: Representing the day of the year. Suppose we wish to represent the day of the year with a digital signal. (We will ignore for now the problem of leap years.) The signal is to be used for operations that include determining the next day (i.e., given the representation of today, compute the representation of tomorrow), testing if two days are in the same month, determining if one day comes before another, and if a day is a particular day of the week.

One approach is to use a $\lceil \log_2 365 \rceil = 9$ bit signal that represents the integers from 0 to 364 where 0 represents January 1 and 364 represents December 31. This representation is compact (you can't do it in less than 9 bits), and it makes it easy to determine if one day comes before another. However, it does

not facilitate the other two operations we need to perform. To determine the month a day corresponds to requires comparing the signal to ranges for each month (January is 0-30, February 31 to 58, etc...), and determining the day of the week requires taking the remainder modulo 7.

A better approach, for our purposes, is to represent the signal as a four-bit month field (January = 1, December = 12) and a five bit day field (1-31). With this representation, for example, July 4 (Independence Day) is 0111 00100_2. The $0111_2 = 7$ represents July and $00100_2 = 4$ represents the day. With this representation we can still directly compare whether one day comes before another and also easily test if two days are in the same month (by comparing the upper four bits.) However, it is even more difficult with this representation to determine the day of the week.

To solve the problem of the day of the week, we use a redundent representation that consists of a four-bit month field (1-12), a five-bit day of the month field (1-31), and a three-bit day of the week field (Sunday = 1,. . . ,Saturday = 7). With this representation, July 4 (which is a Monday in 2005) would be represented as the 12-bit binary number 0111 00100 010. The 0111 means month 7 or July, 00100 means day 4 of the month, and 010 means day 2 of the week or Monday.

Example: Representing subtractive colors. We often pick a representation to simplify carrying out operations on that representation. For example, suppose we wish to represent colors using a *subtractive* system. In a subtractive system we start with white (all colors) and filter this with one or more primary color (red, blue, or yellow) transparent filters. For example, if we start with white use a red filter we get red. If we then add a blue filter we get purple, and so on. By filtering white with the primary colors we can generate derived colors purple, orange, green, and black.

One possible representation for colors is shown in Table 1.2. In this representation we use one bit to denote each of the primary colors. If this bit is set a filter of that primary color is place. We start with white represented as all zeros - no filters in place. Each primary color has exactly one bit set - only the filter of that primary color in place. The derived colors orange, purple, and green, each have two bits set since they are generated by two primary color filters. Finally, black is generated by using all three filters, and hence has all three bits set.

It is easy to see that using this representation, the operation of mixing two colors together (adding two filters) is equivalent to the operation of taking the logical OR of the two representations. For example, if we mix red 001 with blue 100 we get purple 101, and $001 \vee 100 = 101$. [3]

[3] That the symbol $\vee$ denotes the logical OR of two binary numbers. See Chapter 3.

Color	Code
White	000
Red	001
Yellow	010
Blue	100
Orange	011
Purple	101
Green	110
Black	111

Table 1.2: Three-bit representation of colors that can be derived by filtering white light with zero or more primary colors. The representation is chosen so that mixing two colors is the equivalent of OR-ing the representations together.

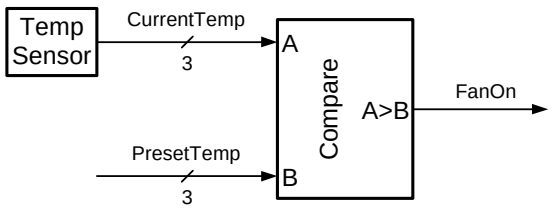


Figure 1.7: A digital thermostat is realized with a comparator. The comparator turns a fan on when the current temperature is larger than a preset temperature.

1.5 Digital Logic Computes Functions of Digital Signals

Once we have represented information as digital signals, we use digital logic circuits to compute logical functions of our signals. That is, the logic computes an output digital signal that is a function of the input digital signal(s).

Suppose we wish to build a thermostat that turns on a fan if the temperature is higher than a preset limit. Figure 1.7 shows how this can be accomplished with a single *comparator*, a digital logic block that compares two numbers and outputs a binary signal that indicates if one is greater than the other. (We will examine how to build comparators in Section 8.5.) The comparator takes two temperatures as input, the current temperature from a temperature sensor, and the preset limit temperature. If the current temperature is greater than the limit temperature the output of the comparator goes high turning the fan on. This digital thermostat is an example of a *combinational logic circuit*, a logic circuit whose output depends only on the current state of its inputs. We will study combinational logic in Chapters 6 to 13.

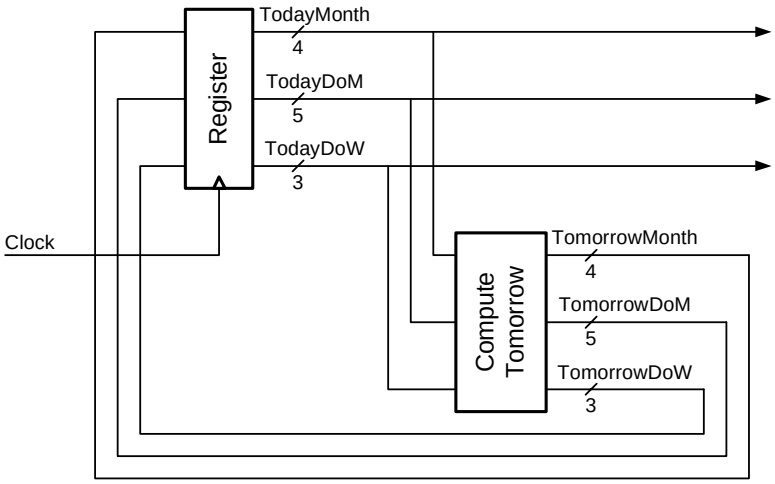


Figure 1.8: A digital calendar outputs the current day in month, day of month, day of week format. A register stores the value of the current day (today). A logic circuit computes the value of the next day (tomorrow).

As a second example, suppose we wish to build a calendar circuit that always outputs the current day in the month, day of month, day of week representation described above. This circuit, shown in Figure 1.8 requires storage. A *register* stores the current day (current month, day of month, and day of week). This register stores the current value, making it available on its output and ignoring its input until the clock rises. When the clock signal rises, the register updates its contents with the value on its input and then resumes its storage function.[4] A logic circuit computes the value of tomorrow from the value of today. This circuit increments the two day fields and takes appropriate action if they overflow. We present the implementation of this logic circuit in Section 9.2. Once a day (at midnight) a *clock* signal rises causing the register to update its contents with tomorrow's value. Our digital calendar is an example of a *sequential* logic circuit. Its output depends not only on current inputs (the clock), but also on internal state (today) which reflects the value of past inputs. We will study sequential logic in Chapters 14 to 19.

We often build digital systems by composing subsystems. Or, from a different perspective, we design a digital system by partitioning it into combinational and sequential subsystems and then designing each subsystem. As a very simple example, suppose we want to modify our thermostat so that the fan does not run on Sundays. We can do this by combining our thermostat circuit with our calendar circuit as shown in Figure 1.9. The calendar circuit is used only for its day of week (DoW) output. This output is compared to the constant

[4] We leave unanswered for now how the register is initially set with the correct date.

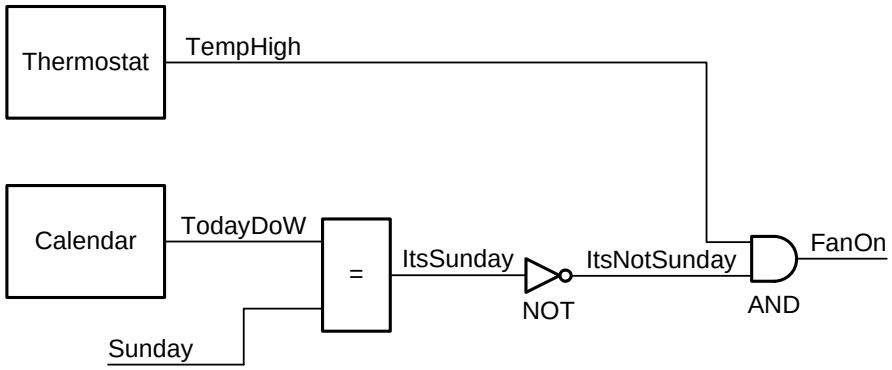


Figure 1.9: By composing our thermostat and calendar circuits we realize a circuit that turns the fan on when the temperature is high except on Sundays when the fan remains off.

Sunday = 1. The output of the comparator is true if today is sunday (ItsSunday) an *inverter*, also called a NOT gate, complements this value. Its output (ItsNotSunday) is true if its not Sunday. Finally, and AND gate combines the inverter output with the output of the thermostat. The output of the AND gate is true only when the temperature is high AND its not Sunday. System-level design — at a somewhat higher level than this simple example — is the topic of Chapters 21 and 20.

1.6 Verilog Is Used to Describe Digital Circuits and Systems

Verilog is a *hardware description language* (HDL) that is used to describe digital circuits and systems. Once a system is described in Verilog we can simulate operation of the circuit using a Verilog simulator. We can also *synthesize* the circuit using a synthesis program (similar to a compiler) to convert the verilog description to a *gate* level description to be mapped to standard cells or an FPGA. Verilog is one of two HDLs in wide use today. (The other is VHDL). Most chips and systems in industry are designed by writing descriptions in one of these two languages.

We will use Verilog througout this book both to illustrate principles and also to teach Verilog coding style. By the end of a course using this book, the reader should be proficient in both reading and writing Verilog.

A Verilog description of our thermostat example is shown in Figure 1.10. The thermostat is described as a Verilog *module* with its code between the keywords `module` and `endmodule`. The first line declares that the module name is `Thermostat` and its interface consists of three signals: `presetTemp`,

```
module Thermostat(presetTemp, currentTemp, fanOn) ;
  input [2:0] presetTemp, currentTemp ; // 3-bit inputs
  output fanOn ;                        // one bit output

  wire fanOn = (currentTemp > presetTemp) ; // compare temps
endmodule
```

Figure 1.10: Verilog description of our thermostat example.

```
# 011 000 -> 0
# 011 001 -> 0
# 011 010 -> 0
# 011 011 -> 0
# 011 100 -> 1
# 011 101 -> 1
# 011 110 -> 1
# 011 111 -> 1
```

Figure 1.11: Result of simulating the Verilog of Figure 1.10 with `presetTemp` = 3 and `currentTemp` sweeping from 0 to 7.

`currentTemp`, and `fanOn`. The second line declares the two temperature signals to be 3-bit wide inputs. The `[2:0]` indicates that `presetTemp` has subsignals `presetTemp[2]`, `presetTemp[1]` and `presetTemp[0]` and similarly for `currentTemp`. The third line declares that `fanOn` is a one bit output. The fourth (non-empty) line describes the whole function of the module. It declares a `wire` for signal `fanOn` and assigns this wire to be true when `currentTemp` is greater than `presetTemp`. The result of simulating this module with `presetTemp` = 3 and `currentTemp` sweeping from 0 to 7 is shown in Figure 1.11.

At first glance, Verilog code looks similar to a conventional programming language like "C" or Java. However, Verilog, or any other HDL, is fundamentally different than a programming language. In a programming language like "C", only one statement is *active* at a time. Statements are executed one at a time in sequence. In Verilog, on the other hand, all modules and all assignment statements in each module are active all of the time. That is all of the statements are executed all of the time.

It is very important in coding Verilog to keep in mind that the code is ultimately being compiled into hardware. Each module instantiated adds a hardware module to the design. Each assignment statement in each module adds gates to each instance of that module. Verilog can be a tremendous productivity multiplier — allowing the designer to work at a much higher level than if she had to manually synthesize gates. At the same time, Verilog can be an impediment if its abstraction causes the designer to lose touch with the end product and

write an inefficient design.

1.7 Outline of this Book

To be written

1.8 Bibliographic Notes

Early calculators Babbage Atanasoff Noyce

1.9 Exercises

1–1 *Gray codes.* A continuous value that has been quantized into N states can be encoded into an $n = \lceil \log_2 N \rceil$ bit signal in which adjacent states differ in at most one bit position. Show how the eight temperatures of Figure 1.6(c) can be encoded into three bits in this manner. Make your encoding such that the encodings of 82 and 68 also differ in just one bit position.

1–2 *Encoding rules.* Equations (1.3) and (1.4) are examples of *decoding* rules that return the value represented by a multi-bit digital signal. Write down the corresponding *encoding* rules. These rules give the value of each bit of the digital signal as a function of the value being encoded.

1–3 *Encoding playing cards.* Suggest a binary representation for playing cards - a set of binary signals that uniquely identifies one of the 52-cards in a standard deck. What different representations might be used to (a) optimize density (minimum number of bits per card), (b) simplify operations such as determining if two cards are of the same suit or rank.

1–4 *Day of the Week.* Explain how to derive the day of the week from the month/day representation of Example ??.

1–5 *Colors.* Derive a representation for colors that supports the operation of additive composition of primary colors. You start with black and add colored light that is red, green, or blue.

1–6 *Colors.* Extend the representation of Exercise 1–5 to support three levels of intensity for each of the primary colored lights. That is each color can be off, weakly on, medium on, or strongly on.

1–7 *Encoding and Decoding.* A 4 core chip is arranged as a 4x1 array of processors where each processor is connected to its east and west neighbors. There are no connections on the ends of the array. The processors' addresses start at 0 on the east-most processor and go up by 1 to address 3 at the west-most processor. Given the current processor's address and

the address of a destination processor, how do you determine whether to go east or west to eventually reach the destination processor?

1–8 *Encoding and Decoding.* A 16 core chip is arranged as a 4x4 array of processors where each processor is connected to its north, south, east, and west neighbors. There are no connections on the edges. Pick an encoding for the address of each processor (0-15) such that when data is moving through the processors it is easy (i.e., similar to Exercise 1–7, above) to determine whether it should move north, south, east, or west at each processor based on the destination address and the address of the current processor.

(a) Draw the array of processors labeling each core with its address according to your encoding.

(b) Describe how to determine the direction the data should move based on current and destination addresses.

(c) How does this encoding or its interpretation differ from simply labeling the processors 0-15 starting at the north-west corner.

1–9 *Noise margins.* Suppose you have a module that uses the encoding described in Table 1.1 but you have freedom to choose either $(V_{OL}, V_{OH}) = (0.3, 2.2)$ or $(0.1, 2.1)$. Which of these output ranges would you choose and why?

1–10 *Circular Gray code.* Come up with a way of encoding the numbers 0-5 onto a 4-bit binary signal so that adjacent numbers differ in only one bit and also so that the representations of 0 and 5 differ in only one bit.

1–11 *Gain of restoring devices.* What is the minimum absolute value of gain for a circuit that restores signals according to the values in Table 1.1.

1–12 *Noise Margins.* Two wires have been placed close together on a chip. They are so close, in fact, that the larger wire (the aggressor) couples to the smaller wire (the victim) and causes the voltage on the victim wire to change. Using the data from Table 1.1, determine the following:

(a) If the victim wire is at V_{OL}, what is the most the aggressor can push it up without causing a problem?

(b) If the victim wire is at V_{OL}, what is the most the aggressor can push it down without causing a problem?

(c) If the victim wire is at V_{OH}, what is the most the aggressor can push it up without causing a problem?

(d) If the victim wire is atV_{OH}, what is the most the aggressor can push it down without causing a problem?

1–13 *Power-supply noise.* Two systems A and B that use the encoding of Table 1.1 send logic signals to one another. Suppose there is a voltage shift between the two systems' power supplies so that all voltages in A are V_N higher than in B (i.e., A voltage of V_x in system A appears as a voltage of $V_x + V_N$ in system B.) Assuming that there are no other noise sources, over what range of V_N will the system operate properly?

1–14 *Proportional signal levels.* A logic device encodes signals with levels proportional to its power supply V_{DD} voltage according to the following table:

Parameter	Value
V_{OL}	$0.1V_{DD}$
V_{IL}	$0.4V_{DD}$
V_{IH}	$0.6V_{DD}$
V_{OH}	$0.9V_{DD}$

Suppose two such logic devices A and B send signals to one another and the supply of device A is $V_{DDA} = 1.0V$. Assuming that there are no other noise sources and that the two devices have a common ground (i.e., $0V$ is the same level in both devices), what is the range of supply voltages for device B, V_{DDB} over which the system will operate properly.

Chapter 2

The Practice of Digital System Design

2.1 The Design Process

As in other fields of engineering, the digital design process begins with a specification. The design then proceeds through phases of concept development, feasibility, partitioning, and detailed design. Most courses, like this one, deal with only the last two steps of this process. To put the design and analysis techniques we will learn in perspective, we will briefly examine the other steps here.

2.1.1 Specification

All designs start with a specification that describes the item to be designed. Depending on the novelty of the object, developing the specification may be a straghtforward process or an elaborate process in itself. The vast majority of designs are evolutionary — the design of a new version of an existing product. For such evolutionary designs, the specification process is one of determining how much better (faster, smaller, cheaper, more reliable, etc...) the new product should be. At the same time, new designs are often constrained by the previous design. For example, a new processor must usually execute the same *instruction set* as the model it is replacing, and a new I/O device must usually support the same standard I/O interface (e.g., a PCI bus) as the previous generation.

On rare occasions, the object being specified is the first of its kind. For such revolutionary developments, the specification process is quite different. There are no constraints of backward compatibility (e.g., of instruction sets and interfaces); although the new object may need to be compatible with one or more *standards*. This gives the designer more freedom, but also less guidance in determining the function, features, and performance of the object.

Whether revolutionary or evolutionary, the specification process is an itera-

tive process — like most engineering processes. We start by writing a *straw man* specification for the object — and in doing so identify a number of questions or open issues. We then iteratively refine this initial specification by gathering information to answer the questions or resolve the open issues. We meet with customers or end users of the product to determine the features they want, how much they value each feature, and how they react to our proposed specification. We commission engineering studies to determine the cost of certain features (e.g., how much die area will it take to reach a certain level of performance, or how much power will be dissipated by adding a branch predictor to a processor). Each time a new piece of information comes in we revise our specification to account for the new information. A history of this revision process is also kept to give a rationale for the decisions made.

While we could continue forever refining our specification, ultimately we must *freeze* the specification and start design. The decision to freeze the specification usually is driven by a combination of schedule pressure (if the product is too late, it will miss a market *window*) and resolution of all critical open issues. Just because the specification is frozen does not mean that it cannot change. If a critical flaw is found after the design starts the specification must be changed. However, after freezing the specification, changes are much more difficult in that they must proceed through an *engineering change control* process. This is a formal process that makes sure that any change to the specification is propagated into all documents, designs, test programs, etc.... and that all people affected by the change *sign off* on it. It also assesses the cost of the change — in terms of both dollars and schedule slippage — as part of the decision process to make the change.

The end product of the specification process is an English language document that describes the object to be designed. Different companies use different names for this document. Many companies call it a product specification or (for chip makers) component specification. A prominent microprocessor manufacturer calls it a *target specification* or TSPEC.[1] It describes the object's function, interfaces, performance, power dissipation, and cost. In short it describes *what* the product does, but not *how* it does it — that's what the design does.

2.1.2 Concept Development and Feasibility

During the concept development phase the high-level design of the system is performed. Block diagrams are drawn, major subsystems are defined, and the rough outline of system operation is specified. More importantly key engineering decisions are made at this stage. This phase is driven by the specification. The concept developed must meet the specification, or if a requirement is too difficult to meet, the specification must be changed.

In the partitioning as well as in the specification of each subsystem, different approaches to the design are developed and evaluated. For example, to build

[1] Often the product specification is accompanied by a business plan for the new product that includes sales forecasts and computes the return on investment for the new product development. However, that is a separate document.

a large communication switch we could use a large crossbar, or we could use a multi-stage network. During the concept development phase we would evaluate both approaches and select the one that best meets our needs. Similarly, we may need to develop a processor that is 1.5× the speed of the previous model. During the concept development phase we would consider increasing clock rate, using a more accurate branch predictor, increasing cache size, and/or increasing issue width. We would evaluate the costs and benefits of these approaches in isolation and in combination.

Technology selection and vendor qualification is also a part of concept development. During these processes, we select what components and processes we are going to use to build our product and determine who is going to supply them to us. In a typical digital design project, this involves selecting suppliers of standard chips — like memory chips and FPGAs, suppliers of custom chips — either an ASIC vendor or a foundry, suppliers of packages, suppliers of circuit boards, and suppliers of connectors. Particular attention is usually paid to components, processes, or suppliers who are new since they represent an element of risk. For example, if we consider using a new optical transceiver or optical switch that has never been built or used before, we need to weigh the probability that it may not work, may not meet specifications, or may not be available when we need it.

A key part of technology selection is making *make vs. buy* decisions about different pieces of the design. For example, you may need to choose between designing your own Ethernet interface, or buying the Verilog for the interface from a vendor. The two (or more) alternatives are evaluated in terms of cost, schedule, performance, and risk. A decision is then made based on the merits of each. Often information needs to be gathered (from design studies, reference checks on vendors, etc...) before making the decision. Too often, engineers favor building things themselves when it is often much cheaper and faster to buy a working design from a vendor. On the other hand, "caveat emptor" applies to digital design. Just because someone is selling a product doesn't mean that it works or meets specification. You may find that the Ethernet interface you purchased doesn't work on certain packet lengths. Each piece of technology acquired from an outside supplier represents a risk and needs to be carefully verified before it is used. This verification can often be a large fraction of the effort that would have been required to do the design yourself.

A large part of engineering is the art of managing technical risk — of setting the level of ambition high enough to give a winning product, but not so high that the product can't be built in time. A good designer makes a few calculated risks in selected areas that give big returns and manages them carefully. Being too conservative (taking no risks, or too few risks) usually results in a non-competitive product. On the other hand, being too aggressive (taking too many risks — particularly in areas that give little return) results in a product that is too late to be relevant. My experience is that far more products fail for being too aggressive (often in areas that don't matter) than too conservative.

To manage technical risks effectively risks must be identified, evaluated, and mitigated. Identifying risks calls attention to them so they can be monitored.

Once we have identified a risk we evaluate it along two axes — importance and danger. For importance, we ask what do we gain by taking this risk. If it doubles our system performance or halves its power it might be worth taking. However, if the gain (compared to a more conservative alternative) is negligible, there is no point taking the risk. For danger, we quantify or classify risks according to how likely they are to succeed. One approach is to assign two numbers between 1 and 5 to each risk, one for importance and one for danger. Risks that are (1,5) — low importance and high danger — are abandoned. Risks that are (5,1) — nearly sure bets with big returns — are kept and managed. Risks that rank (5,5) — very important and very dangerous — are the trickiest. We can't afford to take too many risks, so some of these have to go. Our approach is to reduce the danger of these risks through mitigation — turning a (5,5) into a (5,4) and eventually into a (5,1).

Many designers manage risks informally — following a process similar to the one described here in their heads and then making *gut* decisions as to which risks to take and which to avoid. This is a bad design practice for several reasons. It doesn't work with a large design team (written documents are needed for communication) or for large designs (there are too many risks to keep in one head). Because it is not quantitative, it often makes poor choices. Also, it leaves no written rationale as to why a particular set of risks were chosen and others were avoided.

We often mitigate risks by gathering information. For example, suppose our new processor design calls for a single pipeline stage to check dependencies, rename registers, and issue instructions to eight ALUs (a complex logical function) and we have identified this as both important (it buys us lots of performance) and dangerous (we aren't sure it can be done at our target clock frequency). We can reduce the danger to zero by carrying out the design early and establishing that it can be done. This is often called a *feasibility study* — we establish that a proposed design approach is in fact feasible. We can often establish feasibility (to a high degree of probability) with much less effort than competing a detailed design.

Risks can also be mitigated by developing a backup plan. For example, suppose that one of the (5,5) risks in our conceptual design is the use of a new SRAM part made by a small manufacturer that is not going to be available until just before we need it. We can reduce this risk by finding an alternative component, that while not quite as good is sure to be available when we need it, and designing our system so it can use either part. Then if the new part is not available in time, rather than not having a system at all, we have a system that has just a little less performance — and can be upgraded when the new component is out.

Risks cannot be mitigated by ignoring them and hoping that they go away. This is called *denial* and is a sure-fire way to make a project fail.

With a formal risk management process, identified risks are typically reviewed on a periodic basis (e.g., once every week or two). At each review the importance and danger of the risk is updated based on new information. This review process makes risk mitigation visible to the engineering team. Risks

that are successfully being mitigated, whether through information gathering or backup plans, will have their danger drop steadily over time. Risks that are not being properly managed will have their danger level remain high - drawing attention to them so that they can be more successfully managed.

The result of the concept development phase is a second English language document that describes in detail *how* the object is to be designed. It describes the key aspects of the design approach taken — giving a rationale for each. It identifies all of the outside players: chip suppliers, package suppliers, connector suppliers, circuit-board supplier, CAD tool providers, design service providers, etc.... It also identifies all risks and gives a rationale for why they are worth taking and describes what actions have been done or are ongoing to mitigate them. Different companies use different names for this *how* document. It has been called an implementation specification and a product implementation plan.

2.1.3 Partitioning and Detailed Design

Once the concept phase is complete and design decisions have been made, what remains is to partition the design into modules and then perform the detailed design of each module. The high-level system partitioning is usually done as part of the conceptual design process. A specification is written for each of these high-level modules with particular attention to interfaces. These specifications enable the modules to be designed independently and, if they all conform to the specification, work when plugged together during system integration.

In a complex system the top-level modules will themselves be partitioned into submodules, and so on. The partitioning of modules into sub-modules is often referred to as block-level design since it is carried out by drawing *block diagrams* of the system where each block represents a module or sub-module and the lines between the modules represent the interfaces over which the modules interact.

Ultimately we subdivide a module to the level where each of its submodules can be directly realized using a synthesis procedure. These bottom level modules may be combinational logic blocks — that compute a logical function of their input signals, arithmetic modules — like adders and multipliers, — and finite-state machines that sequence the operation of the system. Much of this course focuses on the design and analysis of these bottom-level modules. It is important to keep in perspective where they fit in a larger system.

2.1.4 Verification

In a typical design project, more than half of the effort goes not into design, but into verifying that the design is correct. Verification takes place at all levels: from the conceptual design down to the design of individual modules. At the highest level, architectural verification is performed on the conceptual design. In this process, the conceptual design is checked against the specification to ensure that every requirement of the specification is satisfied by the implementation.

At the individual module level, *unit tests* are written to verify the functionality of each module. Typically there are far more lines of test code than there are lines of Verilog implementing the modules. After the individual modules are verified they are integrated and the process is repeated for the enclosing subsystem. Ultimately the entire system is integrated and a complete suite of tests are run to validate that the system implements all aspects of the specification.

The verification effort is usually performed according to yet another written document called a *test plan.* [2] In the test plan every feature of the device under test (DUT) is identified and tests are specified to *cover* all of the identified features. Typically a large fraction of tests deal with error conditions - how the system responds to inputs that are outside its normal operating modes - and boundary cases - inputs that are just inside or just outside the normal operating mode.

When time and resources get short engineers are sometimes tempted to take shortcuts and skip some verification. This is almost never a good idea. A healthy philosoply toward verification is: *If it hasn't been tested, it doesn't work.* Every feature, mode, and boundary condition needs to be tested or, chances are the one you skipped will be the one that doesn't work. In the long run the design will get into production more quickly if you complete each step of the verification and resist the temptation to take shortcuts.

2.2 Digital Systems are Built from Chips and Boards

[This section will be revised later with photos of chips, packages, boards, and connectors]

Modern digital systems are implemented using a combination of standard integrated circuits and custom integrated circuits interconnected by circuit boards that in turn are interconnected by connectors and cables.

Standard integrated circuits are parts that can be ordered from a catalog and include memories of all types (SRAM, DRAM, ROM, EPROM, EEPROM, etc...), programmable logic (like the FPGAs we will use in this class), microprocessors, and standard peripheral interfaces. Designers make every effort possible to use a standard integrated circuit to realize a function, since these components can simply be purchased, there is no development cost or effort and usually little risk associated with these parts. However, in some cases, a performance, power, or cost specification cannot be realized using a standard component, and a custom integrated circuit must be designed.

Custom integrated circuits (sometimes called ASICs — for application specific integrated circuits) are chips built for a specific function. Or put differently, they are chips you design yourself because you can't find what you need in a catalog. Most ASICs are built using a *standard cell* design method in which standard

[2] As you can see, most engineers spend more time writing English language documents than writing Verilog or "C" code.

Module	Area (Grids)
One bit of DRAM	2
One bit of ROM	2
One bit of SRAM	24
2-input NAND gate	40
Static Latch	100
Flip-Flop	300
1-bit of a ripple-carry adder	500
32-bit carry-lookahead adder	30,000
32-bit multiplier	300,000
32-bit RISC microprocessor (w/o caches)	500,000

Table 2.1: Area of integrated circuit components in grids

modules (cells) are selected from a catalog and instantiated and interconnected on a silicon chip. Typical standard cells include simple gate circuits, SRAM and ROM memories, and I/O circuits. Some vendors also offer higher-level modules such as arithmetic units, microprocessors, and standard peripherals - either as cells, or as synthesizable RTL (e.g., Verilog). Thus, designing an ASIC from standard cells is similar to designing a circuit board from standard parts. In both cases, the designer selects cells from a catalog and specifies how they are connected. Using standard cells to build an ASIC has the same advantages as using standard parts on a board: no development cost and reduced risk. In rare cases, a designer will design their own non-standard cell at the transistor level. Such custom cells can give significant performance, area, and power advantages over standard-cell logic, but should be used sparingly because they involve significant design effort and are major risk items.

Field programmable gate arrays (FPGAs) are an intermediate point between standard parts and ASICs. They are standard parts that can be programmed to realize an arbitrary function. While significantly less efficient than an ASIC, they are ideally suited to realizing custom logic in less-demanding, low-volume applications. Large FPGAs, like the Xilinx Vertex-II Pro, contain up to 100,000 four-input logic functions (called LUTs), over 1MByte of SRAM, along with several microprocessors and hundreds of arithmetic building blocks. The programmable logic is significantly (over an order of magnitude) less dense, less energy efficient, and slower than fixed standard-cell logic. This makes it prohibitively costly in high-volume applications. However, in low volume applications, the high per-unit cost of an FPGA is attractive compared with the tooling costs for an ASIC which approach 10^6 dollars for 0.13μm technology.

To give you an idea what will fit on a typical ASIC, Table 2.1 lists the area of a number of typical digital building blocks in units of grids (χ^2). A *grid* is the area between the centerlines of adjacent minimum spaced wires in the x and y directions. In a contemporary 0.13μm process, minimum wire pitch $\chi = 0.5\mu$m and one grid is $\chi^2 = 0.25\mu\text{m}^2$. In such a process, there are 4×10^6 grids per mm^2 and 4×10^8 grids on a relatively small 10mm square die - enough room

for 10 million NAND gates. A simple 32-bit RISC processor, which used to fill a chip in the mid-80s, now fits in less than 1mm^2 of area. As described below (Section 2.4) the number of grids per chip doubles every 18 months, so the number of components that can be packed on a chip is constantly increasing.

Chip I/O bandwidth unfortunately does not increase as fast as the number of grids per chip. Modern chips are limited to about 1,000 signal pins by a number of factors and such high pin counts come at a significant cost. One of the major factors limiting pin count and driving cost is the achievable density of printed circuit boards. Routing all of the signals from a high pin-count integrated circuit out from under the chip's package stresses the density of a printed circuit board and often requires additional layers (and hence cost).[3]

Modern circuit boards are laminated from copper-clad glass-epoxy boards interleaved with *pre-preg* glass epoxy sheets. The copper-clad boards are patterned using photolithography to define wires and then laminated together. Connections between layers are made by drilling the boards and electroplating the holes. Boards can be made with a large number of layers — 20 or more is not unusual, but is costly. More economical boards have 10 layers or less. Layers typically alternate between an x signal layer (carrying signals in the x direction), a y signal layer, and a power plane. The power planes distribute power supplies to the chips, isolate the signal layers from one another, and provide a return path for the transmission lines of the signal layers. The signal layers can be defined with minimum wire width and spacing of 3 mils (0.003 inches — about 75μm). Less expensive boards use 5 mil width and spacing rules.

Holes to connect between layers are the primary factor limiting board density. Because of electroplating limits, the holes must have an aspect ratio (ratio of board thickness to hole diameter) no greater than 10:1. A board with a thickness of 0.1 inch requires a minimum hole diameter of 0.01 inch. Minimum hole-to-hole centerline spacing is 25 mils (40 holes per inch). Consider for example the escape pattern under a chip in a 1mm ball-grid-array (BGA) package. With 5 mil lines and spacing, there is room to escape just one signal conductor between the through holes (with 3 mil width and spacing two conductors fit between holes) requiring a different signal layer for each row of signal balls after the first around the periphery of the chip.

Connectors carry signals from one board to another. Right angle connectors connect cards to a *backplane* or *midplane* that carries signals between the cards (Figure ??). Coplanar connectors connect daughter cards to a mother card (Figure ??).

2.3 Computer-Aided Design Tools

The modern digital designer is assisted by a number of computer-aided design (CAD) tools. CAD tools are computer programs that help manage one or more

[3]This routing of signals out from under a package is often referred to as an *escape pattern*, since the signals are *escaping* from under the chip.

aspects of the design process. CAD tools fall into three major categories: capture, synthesis, and verification. They can also be divided into logical, electrical, and physical design tools.

As the name implies, capture tools help *capture* the design. The most common capture tool is a schematic editor. A designer uses the tool to enter the design as a hierarchical drawing showing the connections between all modules and sub-modules. For many designs a textual hardware description language (HDL) like Verilog is used instead of a schematic, and a text editor is used to capture the design. Having done many designs both ways, I find textual design capture far more productive than schematic capture.

Once a design is captured, verification tools are used to ensure that it is correct. A simulator, for example, is often used to test the functionality of a schematic or HDL design. Test scripts are written to drive the inputs and observe the outputs of the design and an error is flagged if the outputs are not as expected. Other verification tools check that a design does not violate simple rules of composition (e.g., only one output driving each wire). Timing tools measure the delay along all possible paths in a design to ensure that they meet timing constraints.

A synthesis tool reduces a design from one level of abstraction to a lower level of abstraction. For example, a logic synthesis tool takes a high-level description of a design in an HDL like Verilog, and reduces it to a gate-level netlist. Logic synthesis tools have largely eliminated manual combinational logic design making designers significantly more productive. A place-and-route tool takes a gate-level netlist and reduces it to a physical design by placing the individual gates and routing the wires between them.

In modern ASICs and FPGAs, a large fraction of the delay and power is due to the wires interconnecting gates and other cells, not due to the gates or cells themselves. To achieve high performance (and low power) requires managing the placement process to ensure that critical signals are short.

2.4 Moore's Law and Digital System Evolution

In 1965, Gordon Moore predicted that the number of transistors on an integrated circuit would double every year. This prediction, that circuit density increases exponentially, has held for 40 years so far, and has come to be known as Moore's law. Over time the doubling every year has been revised to doubling every 18-20 months, but even so, the rate of increase is very rapid. The number of components (or grids) on an integrated circuit is increasing with a compound annual growth rate of over 50% growing by an nearly an order of magnitude roughly every 5 years.

As technology scales, not only does the number of devices increase, but the devices also get faster and dissipate less energy. To first approximation, when the linear dimension L of a semiconductor technology is halved, the area required by a device, which scales as L^2 is quartered, hence we can get four times as many devices in the same area. At the same time, the delay of the

device, which scales with L, is halved — so each of these devices goes twice as fast. Finally, the energy consumed by switching a single device, which scales as L^3, is reduced to one eighth of its original value. This means that in the same area we can do eight times as much work (four times the number of devices running twice as fast) for the same energy.

Moore's law makes the world an interesting place for a digital system designer. Each time the density of integrated circuits increases by an order of magnitude or two (every 5 to 10 years), there is a qualitative change in both the type of systems being designed and the methods used to design them. In contrast, most engineering disciplines are relatively stable — with slow, incremental improvements. You don't see cars getting a factor of 8 more energy efficient every 3 years. Each time such a qualitative change occurs, a generation of designers gets a clean sheet of paper to work on as much of the previous wisdom about how best to build a system is no longer valid. Fortunately, the basic principles of digital design remain invarient as technology scales — however design practices change considerably with each technology generation.

The rapid pace of change in digital design means that digital designers must be students throughout their professional careers, constantly learning to keep pace with new technologies, techniques, and design methods. This continuing education typically involves reading the trade press (EE Times is a good place to start), keeping up with new product announcements from fabrication houses, chip vendors, and CAD tool vendors, and occasionally taking a formal course to learn a new set of skills or update an old set.

2.5 Bibliographic Notes

Moore April 1965 Electronics Magazine.

2.6 Exercises

2–1 Sketch an escape pattern for an BGA package

2–2 Determine the chip area required to implement a particular function

2–3 Given a certain amount of chip area, decide what to do with it

2–4 Risk management exercise.

Chapter 3

Boolean Algebra

We use Boolean algebra to describe the logic functions from which we build digital systems. Boolean algebra is an algebra over two elements: 0 and 1, with three operators: AND, which we denote as $\wedge$, OR, which we denote as $\vee$, and NOT, which we denote with a prime or overbar, e.g., NOT(x) is x' or $\bar{x}$. These operators have their natural meanings: $a \wedge b$ is 1 only if both a and b are 1, $a \vee b$ is 1 if either a or b is 1, and $\bar{a}$ is true only if a is 0.

We write logical expressions using these operators and binary variables. For example, $a \wedge \bar{b}$ is a logic expression that is true when binary variable a is true and binary variable b is false. An instantiation of a binary variable or its complement in an expression is called a *literal*. For example, the expression above has two literals, a and $\bar{b}$. Boolean algebra gives us a set of rules for manipulating such expressions so we can simplify them, put them in *normal form*, and check two expressions for equivalence.

We use the $\wedge$ and $\vee$ notation for AND and OR, and sometimes the Verilog `&` and `|`, to make it clear that Boolean AND and OR are not multiplication and addition over the real numbers. Many sources, including many textbooks, unfortunately use $\times$ or $\cdot$ to denote AND and $+$ to denote OR. We avoid this practice because it can lead students to simplify Boolean expressions as if they were conventional algebraic expressions, that is expressions in the algebra of $+$ and $\times$ over the integers or real numbers. This can lead to confusion since the properties of Boolean Algebra, while similar to conventional algebra, differ in some crucial ways. In particular, Boolean algebra has the property of duality – which we shall discuss below – while conventional algebra does not. One manifestation of this is that in Boolean algebra $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$ while in conventional algebra $a + (b \times c) \neq (a + b) \times (a + c)$.

We will use Boolean algebra in our study of both CMOS logic circuits (Chapter 4) and our study of combinational logic design (Chapter 6).

a	b	$a \wedge b$	$a \vee b$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Table 3.1: Truth tables for AND and OR operations

a	$\bar{a}$
0	1
1	0

Table 3.2: Truth table for NOT operation

3.1 Axioms

All of boolean algebra can be derived from the definitions of the AND, OR, and NOT functions. These are most easily described as truth tables, shown in Tables 3.1 and 3.2. Mathematicians like to express these definitions in the form of *axioms*, a set of mathematical statements that we assert to be true. All of Boolean Algebra derives from the following axioms:

$$\text{Identity}: \quad 0 \wedge x = 0 \qquad 1 \vee x = 1 \tag{3.1}$$

$$\text{Idempotence}: \quad 1 \wedge x = x \qquad 0 \vee x = x \tag{3.2}$$

$$\text{Negation}: \quad \bar{0} = 1 \qquad \bar{1} = 0 \tag{3.3}$$

The *duality* of Boolean algebra is evident in these axioms. The principle of duality states that if a boolean expression is true, then replacing that expression with one where (a) all $\vee$s are replaced by $\wedge$s and vice versa and (b) all 0s are replaced by 1s and vice versa also gives an expression that is true. Since this duality holds in the axioms, and all of Boolean algebra is derived from these axioms, duality holds for all of Boolean algebra.

3.2 Properties

From our axioms we can derive a number of useful properties about Boolean expressions.

x	y	$\overline{(x \wedge y)}$	$\bar{x} \vee \bar{y}$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Table 3.3: Proof of DeMorgan's Law by perfect induction.

Commutative	$x \wedge y = y \wedge x$	$x \vee y = y \vee x$
Associative	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$	$x \vee (y \vee z) = (x \vee y) \vee z$
Distributive	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$	$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
Idempotence	$x \wedge x = x$	$x \vee x = x$
Absorption	$x \wedge (x \vee y) = x$	$x \vee (x \wedge y) = x$
Combining	$(x \wedge y) \vee (x \wedge \bar{y}) = x$	$(x \vee y) \wedge (x \vee \bar{y}) = x$
DeMorgan's	$\overline{(x \wedge y)} = \bar{x} \vee \bar{y}$	$\overline{(x \vee y)} = \bar{x} \wedge \bar{y}$

These properties can all be proved by checking their validity for all four possible combinations of x and y or for all eight possible combinations of x, y, and z. For example, we can prove DeMorgan's Theorem as shown in Table 3.3. Mathematicians call this proof technique *perfect induction*.

This list of properties is by no means exhaustive. We can write down other logic equations that are always true. This set is chosen because it has proven to be useful in simplifying logic equations.

The commutative and associative properties are identical to the properties you are already familiar with from conventional algebra. We can reorder the arguments of an AND or OR operation and an AND or OR with more than two inputs can be grouped in an arbitrary manner. For example, we can rewrite $a \wedge b \wedge c \wedge d$ as $(a \wedge b) \wedge (c \wedge d)$ or as $(d \wedge (c \wedge (b \wedge a)))$. Depending on delay constraints and the library of available logic circuits, there are times when we would use both forms.

The distributive property is also similar to the corresponding property from conventional algebra. It differs, however, in that it applies both ways. We can distribute OR over AND as well as AND over OR. In conventional algebra we cannot distribute + over $\times$.

The next three properties (idempotence, absorption, and combining, have no equivalent in conventional algebra. These properties are very useful in simplifying equations. For example, consider the following logic function:

$$f(a,b,c) = (a \wedge c) \vee (a \wedge b \wedge c) \vee (\bar{a} \wedge b \wedge c) \vee (a \wedge b \wedge \bar{c}) \tag{3.4}$$

First, we apply idempotence twice to triplicate the second term and apply the commutative property to regroup the terms:

$$f(a,b,c) = (a \wedge c) \vee (a \wedge b \wedge c) \vee (\bar{a} \wedge b \wedge c) \vee (a \wedge b \wedge c) \vee (a \wedge b \wedge \bar{c}) \vee (a \wedge b \wedge c) \quad (3.5)$$

Now we can apply the absorption property to the first two terms[1] and the combining property twice - to terms 3 and 4 and to terms 5 and 6 giving:

$$f(a,b,c) = (a \wedge c) \vee (b \wedge c) \vee (a \wedge b) \quad (3.6)$$

In this simplified form it is easy to see that this is the famous *majority function* that is true whenever two or three of its input variables are true.

3.3 Dual Functions

The dual of a logic function, f, is the function f^D derived from f by substituting a $\wedge$ for each $\vee$, a $\vee$ for each $\wedge$ a 1 for each 0, and a 0 for each 1.

For example, if

$$f(a,b) = (a \wedge b) \vee (b \wedge c), \quad (3.7)$$

then

$$f^D(a,b) = (a \vee b) \wedge (b \vee c). \quad (3.8)$$

A very useful property of duals is that the dual of a function applied to the complement of the input variables equals the complement of the function. That is:

$$f^D(\bar{a},\bar{b},\ldots) = \overline{f(a,b,\ldots)}. \quad (3.9)$$

This is a generalized form of DeMorgan's Theorem which states the same result for simple AND and OR functions. We will use this property in Section 4.3 to use dual switch networks to construct the pull-up networks for CMOS gates.

3.4 Normal Form

Often we would like to compare two logical expressions to see if they represent the same function. We could verify equivalence by testing them on every possible input combination — essentially filling out the truth tables and comparing

[1] The estute reader will notice that this gets us back to where we started before making a copy of the second term. However it is useful to demonstrate the absorption property.

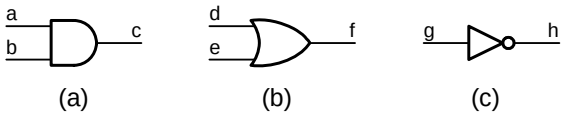


Figure 3.1: Logic symbols for (a) and AND gate, (b) an OR gate, and (c) an Inverter.

them. However an easier approach is to put both expressions into *normal form* — as a sum of product terms.[2]

For example, the normal form for the three-input majority function of Equations (3.4) through (3.6) is:

$$f(a,b,c) = (a \wedge b \wedge \overline{c}) \vee (a \wedge \overline{b} \wedge c) \vee (\overline{a} \wedge b \wedge c) \vee (a \wedge b \wedge c) \tag{3.10}$$

Each product term of a logic expression in normal form corresponds to one row of the truth table for the function. There is a product term for each row that has a 1 in the output column.

We can transform any logic expression into normal form by *factoring* it about each input variable using the identity:

$$f(x_1,\ldots,x_i,\ldots,x_n) = (x_i \wedge f(x_1,\ldots,1,\ldots,x_n)) \vee (\overline{x_i} \wedge f(x_1,\ldots,0,\ldots,x_n)). \tag{3.11}$$

For example, we can apply this method to factor the variable a out from the majority function of Equation (3.6):

$$\begin{aligned} f(a,b,c) &= (a \wedge f(1,b,c)) \vee (\overline{a} \wedge f(0,b,c)) && (3.12)\\ &= (a \wedge (b \vee c \vee (b \wedge c))) \vee (\overline{a} \wedge (b \wedge c)) && (3.13)\\ &= (a \wedge b) \vee (a \wedge c) \vee (a \wedge b \wedge c) \vee (\overline{a} \wedge b \wedge c) && (3.14) \end{aligned}$$

Repeating the expansion about b and c gives the majority function in normal form, Equation (3.10).

3.5 From Equations to Gates

We often represent logical functions using a *logic diagram* - a schematic drawing of gate symbols connected by lines. Three basic gate symbols are shown in

[2]This sum-of-products normal form is often called conjunctive normal form. Because of duality it is equally valid to use a product-of-sums normal form — often called disjunctive normal form.

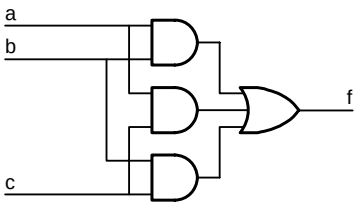


Figure 3.2: Logic diagram for the 3-input majority function.

Figure 3.1. Each gate takes one or more binary inputs on its left side and generates a binary output on its right side. The AND gate (Figure 3.1(a)) outputs a binary signal that is the AND of its inputs - $c = a \wedge b$. The OR gate of Figure 3.1(b) computes the OR of its inputs - $f = d \vee e$. The inverter (Figure 3.1(c)) generates a signal that is the complement of its single input - $h = \overline{g}$. AND gates and OR gates may have more than two inputs. Inverters always have a single input.

Using these three gate symbols we can draw a logic diagram for any boolean expression. To convert from an expression to a logic diagram pick an operator ($\vee$ or $\wedge$) at the top-level of the expression and draw a gate of the corresponding type. Label the inputs to the gate with the subexpression that are arguments to the operator. Repeat this process on the subexpressions.

For example, a logic diagram for the majority function of Equation (3.6) is shown in Figure 3.2. We start by converting the $\vee$ at the top level into a 3-input OR gate at the output. The inputs to this OR gate are the products $a \wedge b$, $a \wedge c$, and $b \wedge c$. We then use three AND gates to generate these three products. The net result is a logic circuit that computes the expression: $f = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$.

Figure 3.3(a) shows a logic diagram for the *exclusive-or* or XOR function, a logic function where the output is high only if exactly one of its inputs is high (i.e., if one input is exclusively high): $f = (a \wedge \overline{b}) \vee (\overline{a} \wedge b)$. The two inverters generate $\overline{b}$ and $\overline{a}$ respectively. The AND gates then form the two products $a \wedge \overline{b}$ and $\overline{a} \wedge b$. Finally, the OR gate forms the final sum.

Because we are frequently complementing signals in logic diagrams, we often drop the inverters and replace them with *inversion bubbles* as shown in Figure 3.3(b). This diagram represents the same function as Figure 3.3(a) we have just used a more compact notation for the inversion of a and b. An inversion bubble may be placed on the input or the output of a gate. In either location it inverts the sense of the signal. Putting an inversion bubble on the input of a gate is equivalent to passing the input signal through an inverter and then connecting the output of the inverter to the gate input.

The exclusive-or function is used frequently enough that we give it its own gate symbol, shown in Figure 3.3(c). It also has its own symbol, $\oplus$, for use in logic expressions: $a \oplus b = (a \wedge \overline{b}) \vee (\overline{a} \wedge b)$.

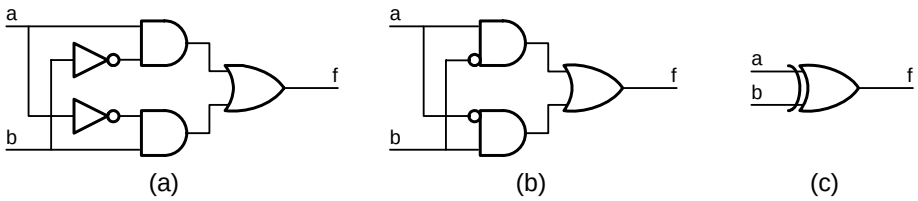


Figure 3.3: The exclusive-or function: (a) logic diagram with inverters, (b) logic diagram with inversion bubbles, (c) gate symbol.

An inversion bubble can be used on the output of a gate as well as the input. Figure 3.4 shows this notation. An AND gate follwed by an inverter (Figure 3.4(a)) is equivalent to an AND gate with an inversion bubble on its output (Figure 3.4(b)). By Demorgan's law, this is also equivalent to an OR gate with inversion bubbles on its input (Figure 3.4(c)). We refer to this gate, that performs the function $f = \overline{(a \wedge b)}$, as a NAND gate (for NOT-AND).

We can apply the same transformation to an OR-gate followed by an inverter (Figure 3.4(d)). We replace the inverter with an inversion bubble to yeild the NOR-gate symbol of Figure 3.4(e) and by applying Demorgan's law we get the alternative NOR-gate symbol of Figure 3.4(f). Because common logic families, such as CMOS, only provide inverting gates, we often use NAND and NOR gates as our primitive building blocks rather than AND and OR.

Figure 3.5 shows how we convert from logic diagrams to equations. Starting at the input, label the output of each gate with an equation. For example, AND gate 1 computes $a \wedge b$ and OR gate 2 computes $c \vee d$ directly from the inputs. Inverter 3 inverts $a \wedge b$ giving $\overline{(a \wedge b)} = \overline{a} \vee \overline{b}$. Note that this inverter could be replaced by an inversion bubble on the input of AND 4. AND 4 combines the output of the inverter with inputs c and d to generate $(\overline{a} \vee \overline{b}) \wedge c \wedge d$. AND 5 combines the outputs of gates 1 and 2 to give $(c \vee d) \wedge a \wedge b$. Finally, OR 6 conbines the outputs of ANDs 4 and 5 to give the final result: $((\overline{a} \vee \overline{b}) \wedge c \wedge d) \vee ((c \vee d) \wedge a \wedge b)$.

3.6 Boolean Expressions in Verilog

In this class you will be implementing digital systems by describing them in a hardware description language named *Verilog* and then compiling your Verilog program to a field-programmable gate array (FPGA). In this section we will introduce Verilog by showing how it can be used to describe logic expressions.

Verilog uses the symbols `&`, `|`, and `~` to represent AND, OR, and NOT respectively. Using these symbols, we can write a Verilog expression for our majority function Equation (3.6) as:

```
assign out = (a & b)|(a & c)|(b & c) ;
```

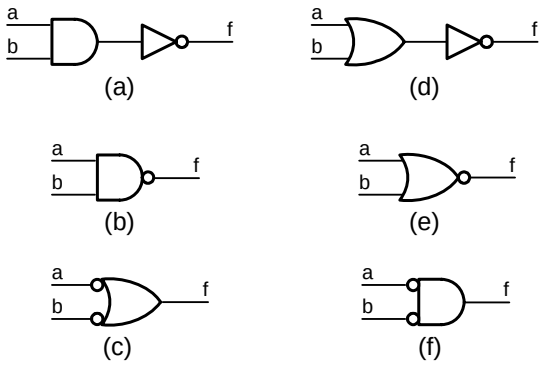


Figure 3.4: NAND and NOR gates: (a) An AND gate followed by an inverter realizes the NAND function. (b) Replacing the inverter with an inversion bubble gives the NAND symbol. (c) Applying Demorgan's law gives an alternate NAND symbol. (d) A OR gate followed by an inverter gives the NOR function. (e) Replacing the inverter with an inversion bubble gives the NOR symbol. (f) Applying Demorgan's law gives an alternate NOR symbol.

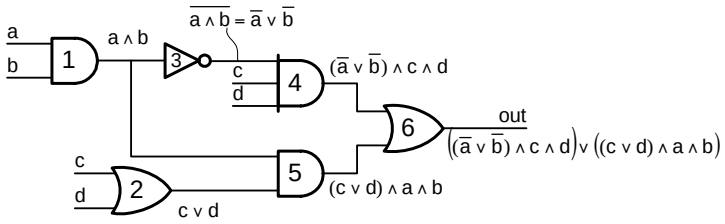


Figure 3.5: Example of converting from a logic diagram to an equation.

```
module Majority(a, b, c, out) ;
  input a, b, c ;
  output out ;

  wire out ;

  assign out = (a & b)|(a & c)|(b & c) ;
endmodule
```

Figure 3.6: Verilog description of a majority gate

The keyword `assign` indicates that this statement describes a combinational logic function that assigns a value to signal `out`. The statement is terminated by a semicolon (`;`).

We can declare the majority gate to be a *module* as shown in Figure 3.6. The first three lines after the comment declares a module named `Majority` with inputs `a`, `b`, and `c`, and output `out`. We then declare that `out` is a wire (don't worry about this). Finally we can insert our `assign` statement to define the function.

To test our majority gate, we can write a test script (Figure 3.7) in Verilog to simulate the gate for all eight possible combinations of input variables. This script declares a three-bit register `count` and instantiates a copy of the majority module with the bits of this register driving the three inputs. The `initial` block defines a set of statements that are executed when the simulator starts. These statements initialize `count` to 0, and then repeats eight times a loop that displays the values of `count` and `out` and then increments `count`. The `#100` inserts 100 units of delay to allow the output of the majority gate to stabilize before displaying it. The result of running this test script is shown in Figure 3.8.

3.7 Bibliographic Notes

Kohavi

3.8 Exercises

3–1 *Prove absorption.* Prove that the absorption property is true by using perfect induction (i.e., enumerate all the possibilities.)

3–2 *Simplify boolean equations.* Reduce the following Boolean expressions to a minimum number of literals.
(a) $(x \vee y) \wedge (x \vee \bar{y})$
(b) $(x \wedge y \wedge z) \vee (\bar{x} \wedge y) \vee (x \wedge y \wedge \bar{z})$
(c) $((y \wedge \bar{z}) \vee (\bar{x} \wedge w)) \wedge (x \wedge \bar{y}) \vee (z \wedge \bar{w})$

```
module test ;
  reg [2:0] count ;     // input - three bit counter
  wire out ;            // output of majority

  // instantiate the gate
  Majority m(count[0],count[1],count[2],out) ;

  // generate all eight input patterns
  initial begin
    count = 3'b000 ;
    repeat (8) begin
      #100
      $display("in = %b, out = %b",count,out) ;
      count = count + 3'b001 ;
    end
  end
endmodule
```

Figure 3.7: Test script to instantiate and exercise majority gate

```
in = 000, out = 0
in = 001, out = 0
in = 010, out = 0
in = 011, out = 1
in = 100, out = 0
in = 101, out = 1
in = 110, out = 1
in = 111, out = 1
```

Figure 3.8: Output from test script of Figure 3.7

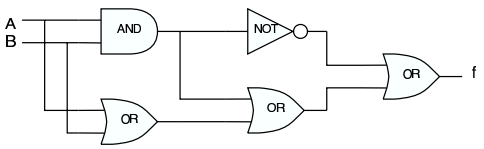


Figure 3.9: Logic circuit for problem 3–7

(d) $(x \wedge y) \vee (x \wedge ((w \wedge z) \vee (w \wedge \bar{z})))$

3–3 *Dual functions.* Find the dual function of each of the following functions and reduce it to normal form.
(a) $f(x, y) = (x \wedge \bar{y}) \vee (\bar{x} \wedge y)$
(b) $f(x, y, z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$

need a few more

3–4 *Normal form.* Rewrite the following Boolean expressions in normal form.
(a) $(x \wedge \bar{y}) \vee (\bar{x} \wedge z)$

need a few more

3–5 *Equation from Schematic.* Write down a Boolean expression for the function computed by the logic circuit of Figure ??.

need a few of these

3–6 *Verilog* Write a Verilog module that implements the logic function

$$f(x, y, z) = (x \wedge y) \vee (\bar{x} \wedge z)$$

And write a test script to verify the operation of your module on all eight combinations of x, y, and z. What function does this circuit realize?

3–7 *Logic Equations.*

(a) Write out the un-simplified logic equation for the circuit of Figure 3.9.

(b) Write the dual form with no simplification.

(c) Draw the circuit for the un-simplified dual form.

(d) Simplify the original equation.

(e) Explain how the inverter and the last OR gate in the original circuit work together to allow this simplification.

Chapter 4

CMOS Logic Circuits

4.1 Switch Logic

In digital systems we use binary variables to represent information and switches controlled by these variables to process information. Figure 4.1 shows a simple switch circuit. When binary variable a is false (0), (Figure 4.1(a)), the switch is open and the light is off. When a is true (1), the switch is closed, current flows in the circuit, and the light is on.

We can do simple logic with networks of switches as illustrated in Figure 4.2. Here we omit the voltage source and light bulb for clarity, but we still think of the switching network as being *true* when its two terminals are connected - i.e., so the light bulb, if connected, would be on.

Suppose we want to build a switch network that will launch a missile only if two switches (activated by responsible individuals) are closed. We can do this as illustrated in Figure 4.2(a) by placing two switches in series controlled by logic variables a and b respectively. For clarity we usually omit the switch symbols and denote a switch as a break in the wire labeled by the variable controlling

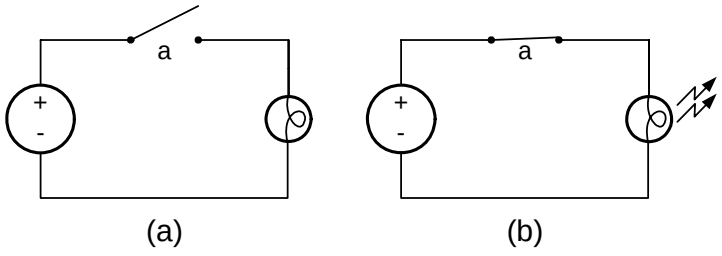


Figure 4.1: A logic variable a controls a switch that connects a voltage source to a light bulb. (a) When $a = 0$ the switch is open and the bulb is off. (b) When $a = 1$ the switch is closed and the bulb is on.

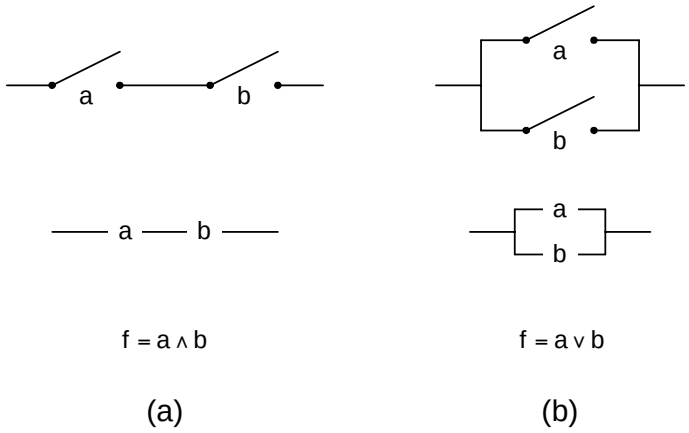


Figure 4.2: AND and OR switch circuits. (a) Putting two switches in series, the circuit is closed only if both logic variable a and logic variable b are true ($a \wedge b$). (b) Putting two switches in parallel, the circuit is closed if either logic variable is true ($a \vee b$). (bottom) For clarity we often omit the switch symbols and just show the logic variables.

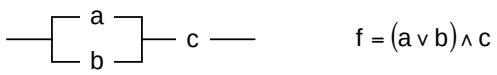


Figure 4.3: An OR-AND switch network that realizes the function $(a \vee b) \wedge c$.

the switch as shown at the bottom of the figure. Only when both a and b are true are the two terminals connected. Thus, we are assured that the missle will only be launched if both a and b agree that it should be launched. Either a or b can stop the launch by not closing its switch. The logic function realized by this switch network is $f = a \wedge b$.[1]

When launching missiles we want to make sure that everyone agrees to launch before going forward. Hence we use an AND function. When stopping a train, on the other hand, we would like to apply the brakes if anyone sees a problem. In that case, we use an OR function as shown in Figure 4.2(b) placing two switches in parallel controlled by binary variables a and b respectively. In this case, the two terminals of the switch network are connected if either a, or b, or both a and b are true. The function realized by the network is $f = a \vee b$.

We can combine series and parallel networks to realize arbitrary logic functions. For example, the network of Figure 4.3 realizes the function $f = (a \vee b) \wedge c$.

[1] Recall from Chapter 3that $\wedge$ denotes the logical AND of two variables and $\vee$ denotes the logical OR of two variables.

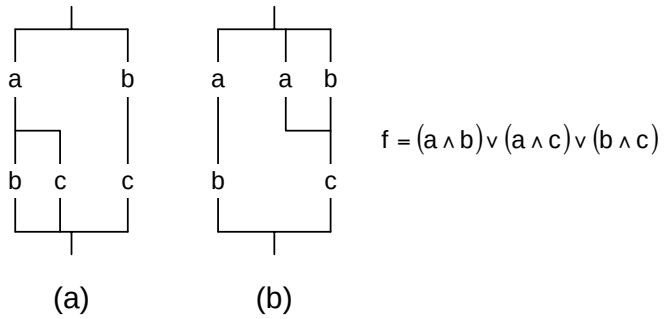


Figure 4.4: Two realizations of a 3-input *majority* function (or 2 out of 3 function) which is true when at least 2 of its 3 inputs is true.

To connect the two terminals of the network c must be true, and either a or b must be true. For example, you might use a circuit like this to engage the starter on a car if the key is turned c and either the clutch is depressed a or the transmission is in neutral b.

More than one switch network can realize the same logical function. For example, Figure 4.4 shows two different networks that both realize the three-input *majority* function. A majority function returns true if the majority of its inputs are true; in the case of a three-input function, if at least two inputs are true. The logic function realized by both of these networks is $f = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$.

There are several ways to analyze a switch network to determine the function it implements. One can enumerate all 2^n combinations of the n inputs to determine the combinations for which the network is connected. Alternatively, one can trace all paths between the two terminals to determine the sets of variables, that if true, make the function true. For a series-parallel network, one can also reduce the network one-step at a time by replacing a series or parallel combination of switches with a single switch controlled by an AND or OR of the previous switches expressions.

Figure 4.5 shows how the network of Figure 4.4(a) is analyzed by replacement. The original network is shown in Figure 4.5(a). We first combine the parallel branches labeled b and c into a single switch labeled $b \vee c$ (Figure 4.5(b)). The series combination of b and c is then replaced by $b \wedge c$ (Figure 4.5(c)). In Figure 4.5(d) the switches labeled a and $b \vee c$ are replaced by $a \wedge (b \vee c)$. The two parallel branches are then combined into $[a \wedge (b \vee c)] \vee (b \wedge c)$ (Figure 4.5(e)). If we distribute the AND of a over $(b \vee c)$ we get the final expression in Figure 4.5(f).

So far we have used only positive switches in our network - that is switches that are closed when their associated logic variable or expression is true (1). The set of logic functions we can implement with only positive switches is very limited

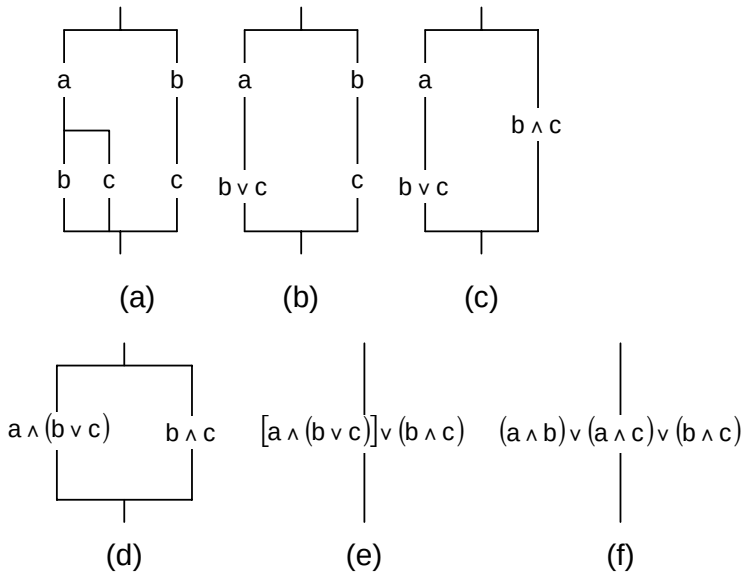


Figure 4.5: We can analyze any series parallel switch network by repeatedly replacing a series or parallel subnetwork by a single switch controlled by the equivalent logic equation.

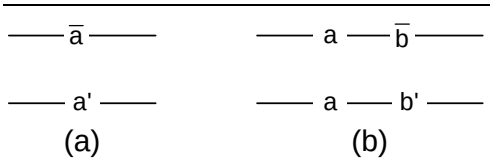


Figure 4.6: A negated logic variable is denoted by a prime a' or an overbar $\bar{a}$. (a) This switch network is closed (true) when variable $a = 0$. (b) A switch network that realized the function $a \wedge \bar{b}$.

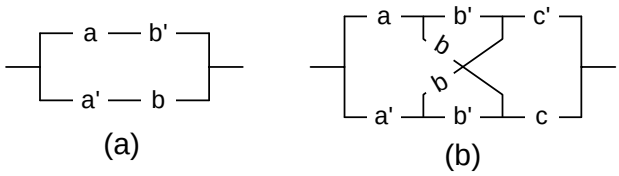


Figure 4.7: Exclusive-or (XOR) switch networks are true (closed) when an odd number of their inputs are true. (a) A two input XOR network. (b) A three-input XOR network.

(monotonically increasing functions). To allow us to implement all possible functions we need to introduce negative switches - switches that are closed when their controlling logic variable is false (0). As shown in Figure 4.6(a)we denote a negative switch by labeling its controlling variable with either a prime a' or an overbar $\bar{a}$. Both of these indicate that the switch is closed when a is false (0). We can build logic networks that combine positive and negative switches. For example, Figure 4.6(b) shows a network that realizes the function $f = a \wedge \bar{b}$.

Often we will control both positive and negative switches with the same logic variable. For example, Figure 4.7(a) shows a switch network that realizes the two-input exclusive-or (XOR) function. The upper branch of the circuit is connected if a is true and b is false while the lower branch is connected if a is false and b is true. Thus this network is connected (true) if exactly one of a or b is true. It is open (false) if both a and b are true.

This circuit should be familar to anyone who has ever used a light in a hallway or stairway controlled by two switches: one at either end of the hall or stairs Changing the state of either switch changes the state of the light. Each switch is actually two switches – one positive and one negative – controlled by the same variable: the position of the switch control. [2]. They are wired exactly as shown in the figure – with switches $a, \bar{a}$ at one end of the hall, and $b, \bar{b}$ at the other end.

In a long hallway, we sometimes would like to be able to control the light from the middle of the hall as well as from the ends. This can be accomplished with the three-input XOR network shown in Figure 4.7(b). An n-input XOR function is true is an odd number of the inputs are true. This three-input XOR network is connected if exactly one of the inputs a, b or c is true or if all three of them are true. To see this is so, you can enumerate all eight combinations of a, b, and c or you can trace paths. You cannot, however, analyze this network by replacement as with Figure 4.5 because it is not a series-parallel network. If you want to have more fun analyzing non-series-parallel networks, see Exercises 4–3 and 4–4.

In the hallway application, the switches associated with a and c are placed at either end of the hallway and the switches associated with b are placed in

[2]Electricians call these three-terminal, two switch units *three-way switches*.

the center of the hall. As you have probably observed, if we want to add more switches controlling the same light, we can repeat the four-switch pattern of the b switches as many times as necessary, each time controlled by a different variable[3].

4.2 A Switch Model of MOS Transistors

Most modern digital systems are built using CMOS (Complementary Metal Oxide Semiconductor) field-effect transistors as switches. Figure 4.8 shows the physical structure and schematic symbol for a MOS transistor. A MOS transistor is formed on a semiconductor substrate and has three terminals[4]: the gate, source, and drain. The source and drain are identical terminals formed by diffusing an impurity into the substrate. The gate terminal is formed from polycrystalline silicon (called *polysilicon* or just *poly* for short) and is insulated from the substrate by a thin layer of oxide. The name MOS, a holdover from the days when the gate terminals were metal (aluminum), refers to the layering of the gate (metal), gate oxide (oxide) and substrate (semiconductor).

Figure 4.8(d), a top view of the MOSFET, shows the two dimensions that can be varied by the circuit or logic designer to determine transistor performance[5]: the device *width* W and the device *length* L. The gate length L is the distance that charge carriers (electrons or holes) must travel to get from the source to the drain and thus is directly related to the speed of the device. Gate length is so important that we typically refer to a semiconductor process by its gate length. For example, most new designs today (2003) are implemented in 0.13μm CMOS processes - i.e., CMOS processes with a minimum gate length of $0.13mu$m. Almost all logic circuits use the minimum gate length supported by the process. This gives the fastest devices with the least power dissipation.

The channel width W controls the strength of the device. The wider the device the more charge carriers that can traverse the device in parallel. Thus the larger W the lower the on resistance of the transistor and the higher the current the device can carry. A large W makes the device faster by allowing it to discharge a load capacitance more quickly. Alas, reduced resistance comes at a cost - the gate capacitance of the device also increases with W. Thus as W increases it takes longer to charge or discharge the gate of a device.

Figure 4.8(c) shows the schematic symbols for an n-channel MOSFET (NFET) and a p-channel MOSFET (PFET). In an NFET the source and drain are n-type semiconductor in a p-type substrate and the charge carriers are electrons.

[3]Electricians call this four-terminal, four-switch unit where the connections are straight through when the variable is false (switch handle down) and crossed when the variable is true (switch handle up) a *four-way switch*. To control one light with $n \geq 2$ switches requires two three-way switches and $n-2$ four-way switches. Of course, one can always use a four-way switch as a three-way switch by leaving one terminal unconnected.

[4]The substrate is a fourth terminal that we will ignore at present.

[5]The gate oxide thickness is also a critical dimension, but it is set by the process and cannot be varied by the designer. In contrast, W and L are determined by the mask set and hence can be adjusted by the designer.

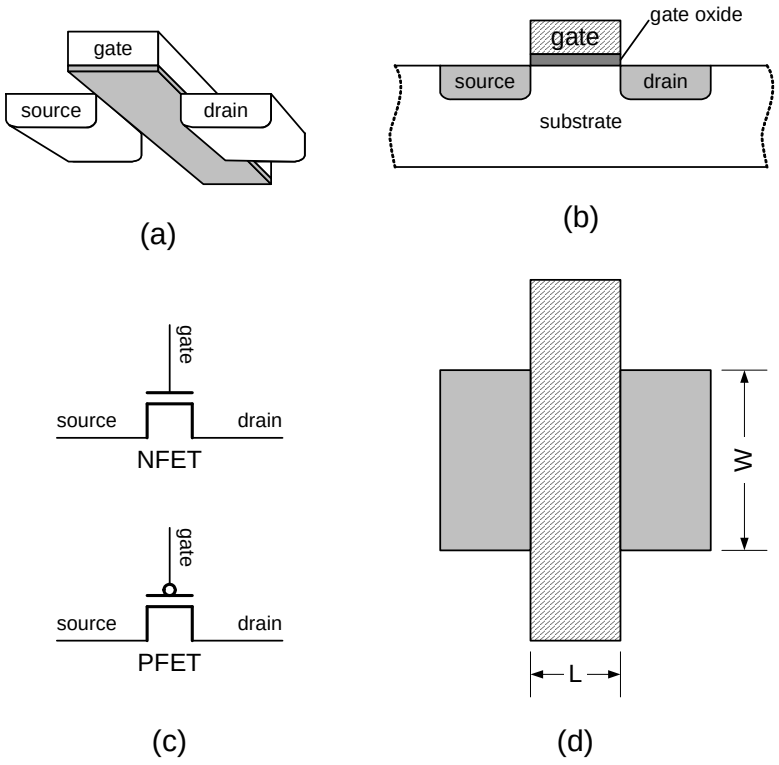


Figure 4.8: A MOS field-effect transistor (FET) has three terminals. Current passes between the source and drain (identical terminals) when the device is on. The voltage on the gate controls whether the device is on or off. (a) The structure of a MOSFET with the substrate removed. (b) A side view of a MOS FET. (c) Schematic symbols for a n-channel FET (NFET) and a p-channel FET (PFET). (d) Top view of a MOSFET showin its width W and length L.

In a PFET the types are reversed - the source and drain are p-type in a n-type substrate (usually a n-well diffused in a p-type substrate) and the carriers are holes.

If you haven't got a clue what n-type and p-type semiconductors, holes, and electrons are, don't worry we will abstract them away shortly. Bear with us for the moment.

Figure 4.9 illustrates a simple digital model of operation for an n-channel FET[6]. As shown in Figure 4.9(a), when the gate of the NFET is a logic 0, the source and drain are isolated from one another by a pair of p-n junctions (back-to-back diodes) and hence no current flows from drain to source, $I_{DS} = 0$. This is reflected in the schematic symbol in the middle panel. We model the NFET in this state with a switch as shown in the bottom panel.

When the gate terminal is a logic 1 *and* the source terminal is a logic zero, as shown in Figure 4.9(b), the NFET is turned on. The positive voltage between the gate and source induces a negative charge in the *channel* beneath the gate. The presence of these negative charge carriers (electrons) makes the channel effectively n-type and forms a conductive region between the source and drain. The voltage between the drain and the source accelerates the carriers in the channel, resulting in a current flow from drain to source, I_{DS}. The middle panel shows the schematic view of the on NFET. The bottom panel shows a switch model of the on NFET. When the gate is 1 and the source is 0, the switch is closed.

It is important to note that if the source[7] is 1, the switch will *not* be closed even if the gate is 1 because there is no net voltage between the gate and source to induce the channel charge. The switch is not open in this state either - because it will turn on if either terminal drops a threshold voltage below the 1 voltage. With source = 1 and gate = 1, the NFET is in an undefined state (from a digital perspective). The net result is that an NFET can reliably pass only a logic 0 signal. To pass a logic 1 requires a PFET

Operation of a PFET, illustrated in Figure 4.10 is identical to the NFET with the 1s and 0s reversed. When the gate is 0 and the source is 1 the device is on. When the gate is 1 the device is off. When the gate is 0 and the source is 0 the device is in an undefined state. Because the source must be 1 for the device to be reliably on, the PFET can reliably pass only a logic 1. This nicely complements the NFET which can only pass a 0.

The NFET and PFET models of Figures 4.9 and 4.10 accurately model the function of most digital logic circuits. However to model the delay and power of logic circuits we must complicate our model slightly by adding a resistance in series with the source and drain and a capacitance from the gate to ground as shown in Figure 4.11[8]. The capacitance on the gate node is proportional to

[6] A detailed discussion of MOSFET operation is far beyond the scope of these notes. Consult a textbook on semiconductor devices for more details.

[7] Physically the source and drain are identical and the distinction is a matter of voltage. The source of an NFET (PFET) is the most negative (positive) of the two non-gate terminals.

[8] In reality there is capacitance on the source and drain nodes as well - usually each has a capacitance equal to about half of the gate capacitance (depending on device size and

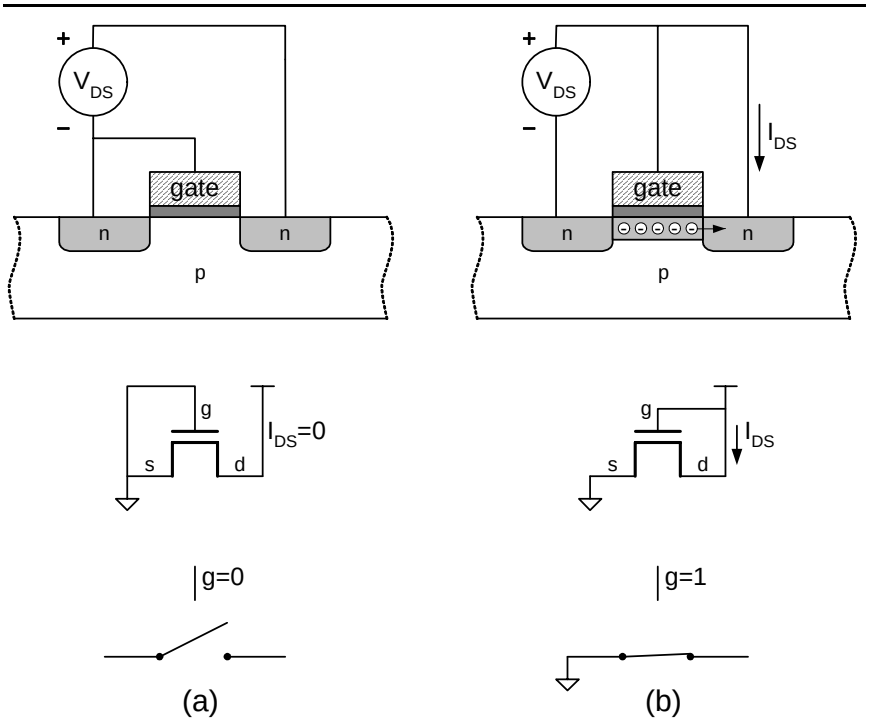


Figure 4.9: Simplified operation of a n-channel MOSFET. (a) When the gate is at the same voltage as the source, no current flows in the device because the drain is isolated by a reverse-biased p-n junction (a diode). (b) When a positive voltage is applied to the gate, it induces negative carriers in the *channel* beneath the gate, effectively inverting the p-type silicon to become n-type silicon. This connects the source and drain allowing a current I_{DS} to flow. The top panel shows what happens physically in the device. The middle panel shows the schematic view. The bottom panel shows a switch model of the device.

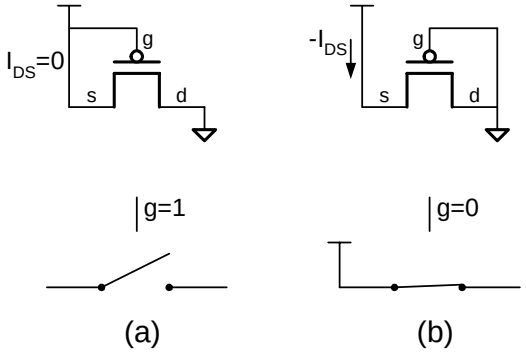


Figure 4.10: A p-channel MOSFET operates identically to an NFET with all 0s and 1s switched. (a) When the gate is high the PFET is off regardless of source and drain voltages. (b) When the gate is low and the source is high the PFET is on and current flows from source to drain.

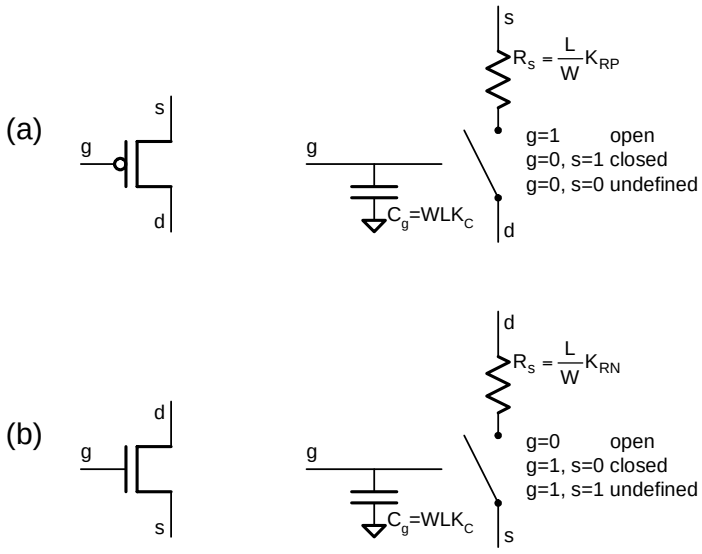


Figure 4.11: A p-channel MOSFET operates identically to an NFET with all 0s and 1s switched. (a) When the gate is high the PFET is off regardless of source and drain voltages. (b) When the gate is low and the source is high the PFET is on and current flows from source to drain.

Parameter	Value	Units
K_C	2×10^{-16}	Farads/$L^2_{\min}$
K_{RN}	2×10^4	Ohms/square
K_P	2.5	
$K_{RP} = K_P K_{RN}$	5×10^4	Ohms/square
$\tau_N = K_C K_{RN}$	4×10^{-12}	seconds
$\tau_P = K_C K_{RP}$	1×10^{-11}	seconds

Table 4.1: Device parameters for a typical 0.13μm CMOS process.

the area of the device, WL. The resistance, on the other hand is proportional to the aspect ratio of the device L/W.

For convenience, and to make our discussion independent of a particular process generation, we will express W and L in units of $L_{\min}$ the minimum gate length of a technology. For example, in an 0.13μm technology, we will refer to a device with $L = 0.13\mu$m and $W = 1.04\mu$m device as a $L = 1$, $W = 8$ device, or just as a $W = 8$ device since $L = 1$ is the default. In some cases we will scale W by $W_{\min} = 8L_{\min}$ that is we will refer to a minimum sized $W/L = 8$ device as a unit-sized device and size other device in relation to this device.

Table 4.2 gives typical values of K_C, K_{RN}, and K_{RP} for an 0.13μm technology. The key parameters here are τ_N and τ_P, the basic time constants of the technology. As technology scales, K_C (expressed as Farads/$L^2_{\min}$) remains roughly proportional to gate length and can be approximated as

$$K_C \approx 1.5 \times 10^{-9} L_{\min}. \tag{4.1}$$

Where $L_{\min}$ is expressed in m. The resistances remain roughly constant as technology scales causing both time constants to also scale linearly with $L_{\min}$.

$$\tau_N \approx 3 \times 10^{-5} L_{\min}. \tag{4.2}$$

$$\tau_P = K_P \tau_N \approx 7.5 \times 10^{-5} L_{\min}. \tag{4.3}$$

4.3 CMOS Gate Circuits

In Section 4.1 we learned how to do logic with switches and in Section 4.2 we saw that MOS transistors can, for most digital purposes, be modeled as switches. Putting this information together we can see how to make logic circuits with transistors.

geometry). For the purposes of these notes, however, we'll lump all of the capacitance on the gate node.

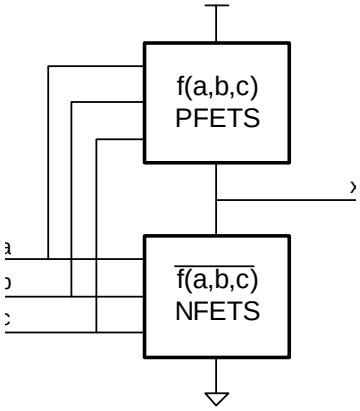


Figure 4.12: A CMOS gate circuit consists of a PFET switch network that pulls the output high when function f is true and an NFET switch network that pulls the output low when f is false.

A well-formed logic circuit should support the digital abstraction by generating an output that can be applied to the input of another, similar logic circuit. Thus, we need a circuit that generates a voltage on its output — not just connects two terminals together. The circuit must also must be restoring, so that degraded input levels will result in restored output levels. To achieve this, the voltage on the output must be derived from a supply voltage, not from one of the inputs.

A *static CMOS gate* circuit realizes a logic function f while generating a restoring output that is compatible with its input as shown in Figure 4.12. When function f is true, a PFET switch network connects output terminal x to the positive supply (V_{DD}). When function f is false, output x is connected to the negative supply by an NFET switch network. This obeys our constraints of passing only logic 1 (high) signals through PFET switch networks and logic 0 (low) signals through NFET networks. It is important that the functions realized by the PFET network and the NFET network be complements. If the functions should overlap (both be true at the same time), a short circuit across the power supply would result drawing a large amount of current and possibly causing permanent damage to the circuit. If the two functions don't cover all input states (there are some input states where neither is true), then the output is undefined in these states.[9]

Because NFETs turn on with a high input and generate a low output and PFETs are the opposite, we can only generate *inverting* (sometimes called monotonically decreasing) logic functions with static CMOS gates. A positive (neg-

[9] We will see in Chapter 23 how CMOS circuits with unconnected outputs can be used for storage.

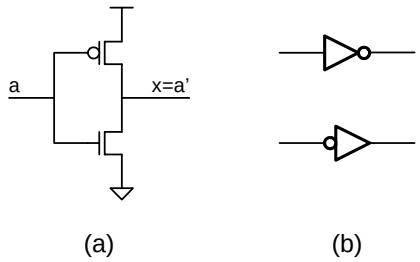


Figure 4.13: A CMOS inverter circuit. (a) A PFET connects x to 1 when $a = 0$ and an NFET connects x to 0 when $a = 1$. (b) Logic symbols for an inverter. The bubble on the input or output denotes the NOT operation.

ative) transition on the input of a single CMOS gate circuit can either cause a negative (positive) transition on the output or no change at all. Such a logic function where transitions in one direction on the inputs cause transitions in just a single direction on the output is called a *monotonic* logic function. If the transitions on the outputs are in the opposite direction to the transitions on the inputs its a monotonic decreasing or inverting logic function. If the transitions are in the same direction, its a monotonic increasing function. To realize a non-inverting or non-monotonic logic function requires multiple stages of CMOS gates.

We can use the principle of duality, Equation (3.9), to simplify the design of gate circuits. If we have an NFET pulldown network that realizes a function $f_n(x_1, \ldots, x_n)$, we know that our gate will realize function $f_p = \overline{f_n(x_1, \ldots, x_n)}$. By duality we know that $f_p = \overline{f_n(x_1, \ldots, x_n)} = f_n^D(\overline{x_1}, \ldots, \overline{x_n})$. So for the PFET pullup network, we want the dual function with inverted inputs. The PFETs give us the inverted inputs, since they are "on" when the input is low. To get the dual function, we take the pulldown network and replace ANDs with ORs and vice-versa. In a switch network, this means that a series connection in the pulldown network becomes a parallel connection in the pullup network and vice-versa.

The simplest CMOS gate circuit is the inverter, shown in Figure 4.13(a). Here the PFET network is a single transistor that connects output x to the positive supply whenever input a is low — $x = \overline{a}$. Similarly the NFET network is a single transistor that pulls output x low whenever the input is high.

Figure 4.13(b) shows the schematic symbols for an inverter. The symbol is a rightward facing triangle with a *bubble* on its input or output. The triangle represents an amplifier — indicating that the signal is restored. The bubble (sometimes called an *inversion bubble*) implies negation. The bubble on the input is considered to apply a NOT operation to the signal before it is input to the amplifer. Similarly a bubble on the output is considered to apply a NOT operation to the output signal after it is amplified. Logically, the two symbols

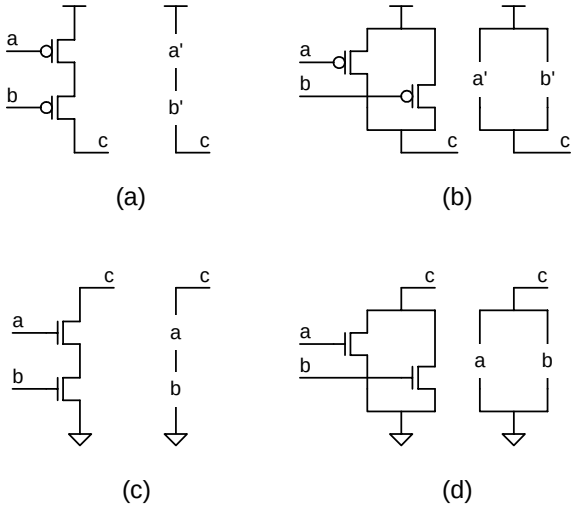


Figure 4.14: Switch networks used to realize NAND and NOR gates. (a) Series PFETs connect the output c high when all inputs are low, $f = \overline{a} \wedge \overline{b} = \overline{a \vee b}$. (b) Parallel PFETs connect the output if either input is low, $f = \overline{a} \vee \overline{b} = \overline{a \wedge b}$. (c) Series NFETs pull the output low when both inputs are high, $f = \overline{a \vee b}$. (d) Parallel NFETs pull the output low when either input is true, $f = \overline{a \wedge b}$.

are equivalent. It doesn't matter if we consider the signal to be inverted before or after amplification. We choose one of the two symbols to obey the *bubble rule* which states that:

> **Bubble Rule:** Where possible signals that are output from a gate with an inversion bubble on its output shall be input to a gate with an inversion bubble on its input.

Schematics drawn using the bubble rule are easier to read than schematics where the polarity of logic signals changes from one end of the wire to the other. We shall see many examples of this in Chapter 6.

Figure 4.14 shows some example NFET and PFET switch networks that can be used to build NAND and NOR gate circuits. A parallel combination of PFETs (Figure 4.14(b)) connects the output high if either input is low, so $f = \overline{a} \vee \overline{b} = \overline{a \wedge b}$. Applying our principle of duality, this switch network is used in combination with a series NFET network (Figure 4.14(c)) to realize a NAND gate. The complete NAND gate circuit is shown in Figure 4.15(a) and two schematic symbols for the NAND are shown in Figure 4.15(b). The upper symbol is an AND symbol (square left side, half-circle right side) with an inversion bubble on the output — indicating that we AND inputs a and b and then invert the output, $f = \overline{a \wedge b}$. The lower symbol is an OR symbol (curved

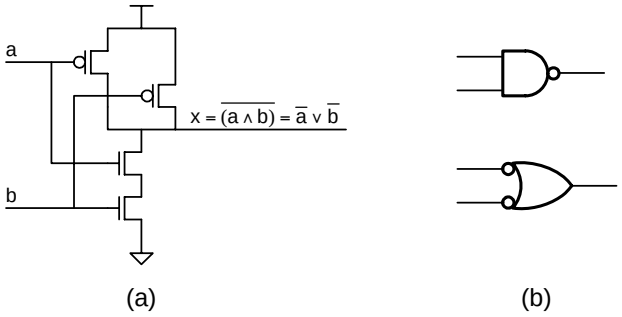


Figure 4.15: A CMOS NAND gate. (a) Circuit diagram — the NAND has a parallel PFET pull-up network and a series NFET pull-down network. (b) Schematic symbols — the NAND function can be though of as an AND with an inverted output (top) or an OR with inverted inputs (bottom).

left side, pointy right side) with inversion bubbles on all inputs — the inputs are inverted and then the inverted inputs are ORed, $f = \overline{a} \vee \overline{b}$. By DeMorgan's law (and duality), these two functions are equivalent. As with the inverter, we select between these two symbols to observe the bubble rule.

A NOR gate is constructed with a series network of PFETs and a parallel network of NFETs as shown in Figure 4.16(a). A series combination of PFETs (Figure 4.14(a)) connects the output to 1 when a and b are both low, $f = \overline{a} \wedge \overline{b} = \overline{a \vee b}$. Applying duality, this circuit is used in combination with a parallel NFET pulldown network (Figure 4.14(d)). The schematic symbols for the NOR gate are shown in Figure 4.16(b). As with the inverter and the NAND, we can choose between inverted inputs and inverted outputs depending on the bubble rule.

We are not restricted to building gates from just series and parallel networks. We can use arbitrary series-parallel networks, or even networks that are not series-parallel. For example, Figure 4.17(a) shows the transistor-level design for an AND-OR-Invert (AOI) gate. This circuit compute the function $f = \overline{(a \wedge b) \vee c}$. The pull-down network has a series connection of a and b in parallel with c. The pull-up network is the dual of this network with a parallel connection of a and b in series with c.

Figure 4.18 shows a majority-invert gate. We cannot build a single-stage majority gate since it is a monotonic increasing function and gates can only realize inverting functions. However we can build the complement of the majority function as shown. The majority is an interesting function in that it is its own dual. That is, $\mathrm{maj}(\overline{a}, \overline{b}, \overline{c}) = \overline{\mathrm{maj}(a, b, c)}$. Because of this we can implement the majority gate with a pull-up network that is identical to the pull-down network as shown in Figure 4.18(a). The majority function is also a *symmetric* logic function in that the inputs are all equivalent. Thus we can permute the inputs to the PFET and NFET networks without changing the function.

A more conventional implementation of the majority-invert gate is shown

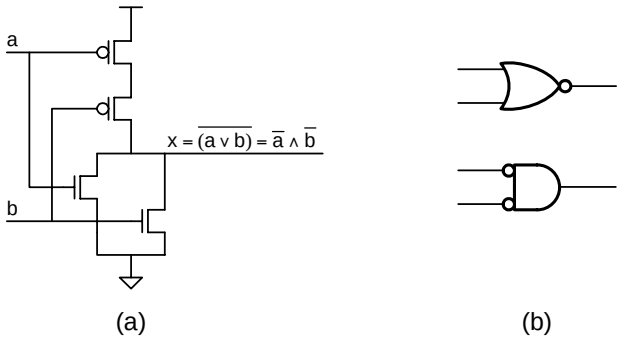


Figure 4.16: A CMOS NOR gate. (a) Circuit diagram — the NOR has a series PFET pull-up network and a parallel NFET pull-down network. (b) Schematic symbols — the NOR can be thought of as an OR with an inverted output or an AND with inverted inputs.

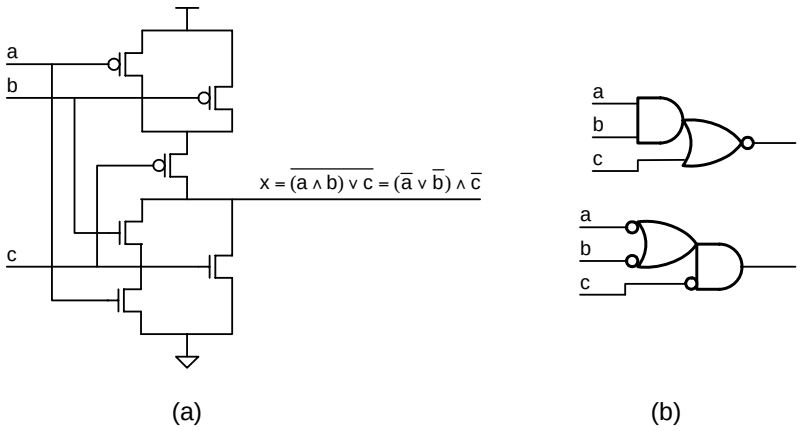


Figure 4.17: An AND-OR-Invert (AOI) gate. (a) Transistor-level implementation uses a parallel-series NFET pull-down network and its dual series-parallel PFET pull-up network. (b) Two schematic symbols for the AOI gate.

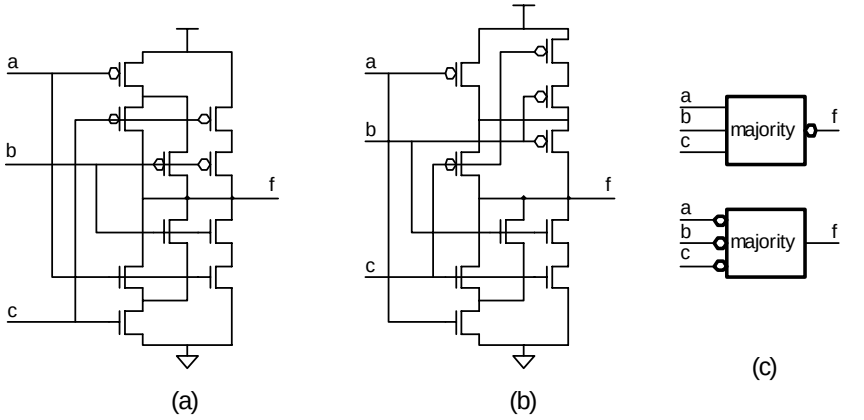


Figure 4.18: A majority-invert gate. The output is false if 2 or 3 of the inputs are true. (a) Implementation with symmetric pull-up and pull-down networks. (b) Implementation with pull-up network that is a dual of the pull-down network. (c) Schematic symbols — the function is still majority whether the inversion is on the input or the output.

in Figure 4.18(b). The NFET pull-down network here is the same as for Figure 4.18(a) but the PFET pull-up network has been replaced by a dual network — one that replaces each series element with a parallel element and vice-versa. The parallel combination of b and c in series with a in the pulldown network, for example, translates to a series combination of b and c in parallel with a in the pull-up network. A PFET pull-up network that is the dual of the NFET pull-down network will always give a switching function that is the complement of the pull-down network because of Equation (3.9).

Figure 4.18(c) shows two possible schematic symbols for the majority-invert gate. Because the majority function is self-dual, it doesn't matter whether we put the inversion bubbles on the inputs or the output. The function is a majority either way. If at least 2 out of the 3 inputs are high the output will be low — a majority with a low-true output. It is also the case that if at least 2 of the 3 inputs are low the output will be high — a majority with low-true inputs.

Strictly speaking, we cannot make a single-stage CMOS exclusive-or (XOR) gate because XOR is a non-monotonic function. A positive transition on an input may cause either a positive or negative transition on an output depending on the state of the other inputs. However, if we have inverted versions of the inputs, we can realize a two-input XOR function as shown in Figure 4.19(a), taking advantage of the switch network of Figure 4.7. A three-input XOR function can be realized as shown in Figure 4.19(b). The switch networks here are not series-parallel networks. If inverted inputs are not available, , it is

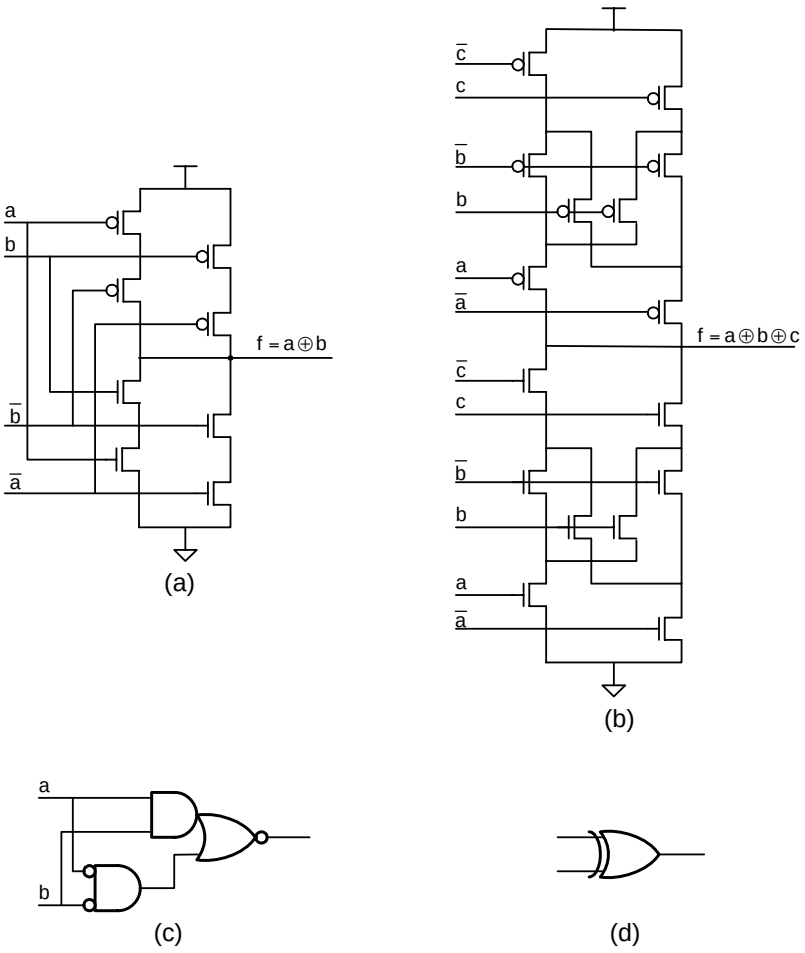


Figure 4.19: Exclusive-or (XOR) gates.

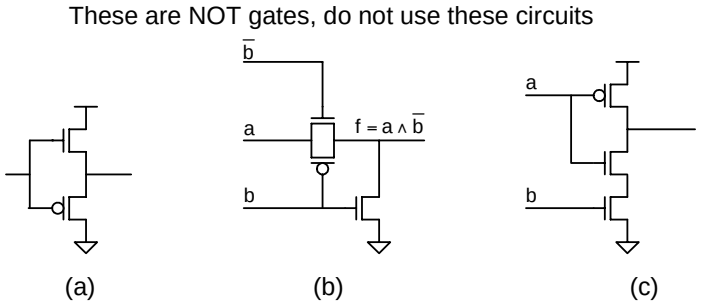


Figure 4.20: Three circuits that are *not* gates and shoudl not be used. (a) Attempts to pass a 1 through an NFET and a 0 through a PFET. (b) Does not restore high output values. (c) Does not drive the output when $a = 1$ and $b = 0$.

more efficient to realize a 2-input XOR gate using two CMOS gates in series as shown in Figure 4.19(c). We leave the transistor-level design of this circuit as an exercise. An XOR symbol is shown in Figure 4.19(d).

Before closing this chapter its worth examining a few circuits that aren't gates and don't work but represent common errors in CMOS circuit design. Figure 4.20 shows three representative mistakes. The would-be buffer in Figure 4.20(a) doesn't work because it attempts to pass a 1 through a PFET and a 0 through an NFET. The transistors cannot reliably pass those values and so the output signal is undefined — attenuated from the input swing at best. The AND-NOT circuit of Figure 4.20(b) does in fact implement the logical function $f = a \wedge \overline{b}$. However, it violates the digital abstraction in that it does not *restore* its output. If $b = 0$ an noise on input a is passed directly to the output.[10] Finally, the circuit of Figure 4.20(c) leaves its output disconnected when $a = 1$ and $b = 0$. Due to parasitic capacitance, the previous output value will be *stored* for a short period of time on the output node. However, after a period, the stored charge will leak off and the output node becomes an undefined value.

4.4 Bibliographic Notes

Kohavi gives a detailed treatment of switch networks. The switch model of the MOS transistor was first proposed by Bryant. A digital circuit design text like Rabaye is a good source of more detailed information on digital logic circuits.

[10]Such circuits can be used with care in isolated areas, but must be followed by a restoring stage before a long wire or another non-restoring gate. In most cases its better to steer clear of such *short-cut* gates.

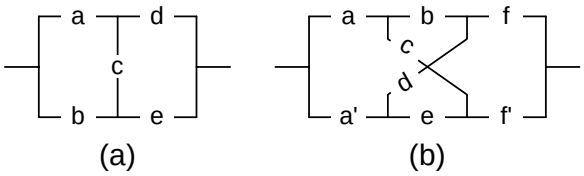


Figure 4.21: Switch network for Exercises 4–3 and 5.7.

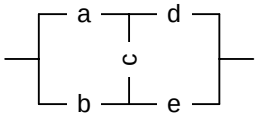


Figure 4.22: Switch network for Exercise 4–5.

4.5 Exercises

4–1 Analyze a simple switch circuit.

4–2 Synthesize a simple switch circuit.

4–3 Write down the logic function that describes the conditions under which the switch network of Figure 4.21(a) connects its two terminals. Note that this is not a series-parallel network.

4–4 Write down the logic function for the network of Figure 4.21(b).

4–5 Write down the logic function for the network of Figure 5.7.

4–6 Draw a schematic using NFETs and PFETs for a restoring logic gate that implements the function $f = a \wedge (b \vee c)$.

4–7 Write down the logic function implemented by the CMOS circuit of Figure ??.

4–8 Draw a transistor-level schematic for the XOR gate of Figure 4.19(c).

Chapter 5

Delay and Power of CMOS Circuits

The specification for a digital system typically includes not only its function, but also the delay and power (or energy) of the system. For example, a specification for an adder describes the function, that the output is to be the sum of the two inputs, as well as the delay, that the output must be valid within 2ns after the inputs are stable, and its energy, that each add consume no more than 5pJ. In this chapter we shall derive simple metohds to estimate the delay and power of CMOS logic circuits.

5.1 Delay of Static CMOS Gates

As illustrated in Figure 5.1 the delay of a logic gate, t_p, is the time from when the input of the gate crosses the 50% point between V_0 and V_1. Specifying delay in this manner allows us to compute the delay of a chain of logic gates by simply summing the delays of the individual gates. For example, in Figure 5.1 the delay from a to c is the sum of the delay of the two gates. The 50% point on the output of the first inverter is also the 50% point on the input of the second inverter.

Because the resistance of the PFET pull-up network may be different than that of the NFET pull-down network, a CMOS gate may have a rising delay that is different from its falling delay. When the two delays differ we denote the rising delay, the delay from a falling input to a rising output, as t_{pr} and the falling delay as t_{pf} as shown in Figure 5.1.

We can use the simple switch model derived in Section 4.2 to estimate t_{pr} and t_{pf} by calculating the RC time constant of the circuit formed by the output resistance of the driving gate and the input capacitance of its load(s).[1] Because

[1] In reality the driving gate has output capacitance roughly equal to its input capacitance. We ignore that capacitance here to simplify the model.

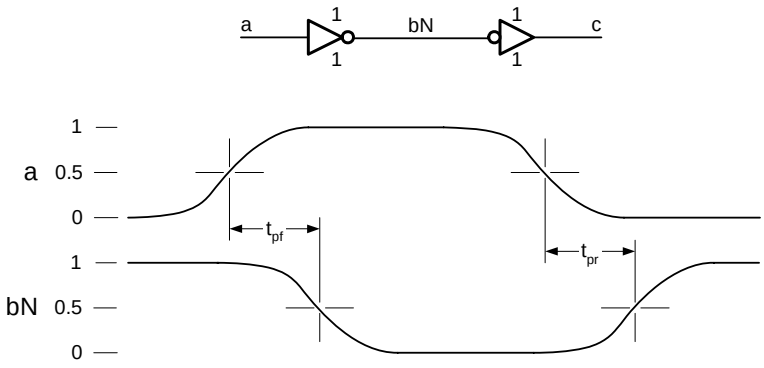


Figure 5.1: We measure delay from the 50% point of an input transition to the 50% point of an output transition. This figure shows the waveforms on input a and output bN with the falling and rising propagation delays, t_{pf} and t_{pr}, labeled

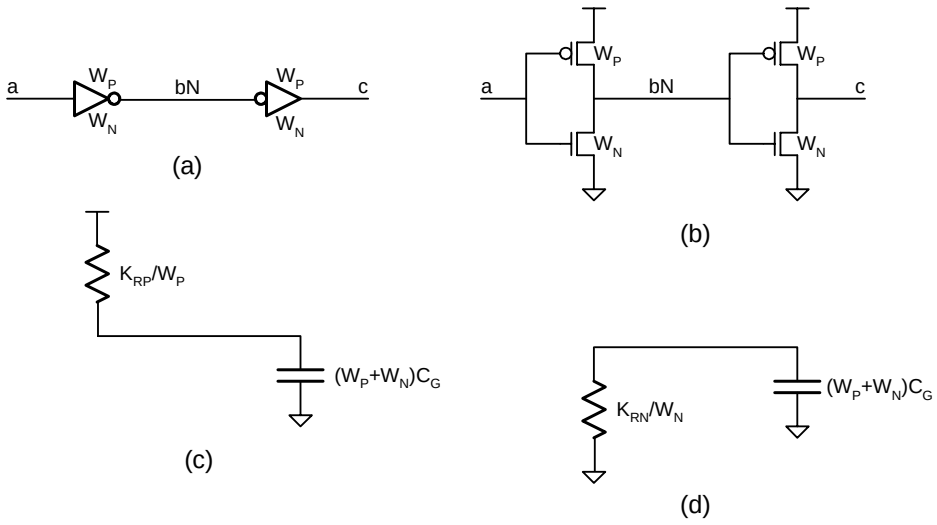


Figure 5.2: Delay of an inverter driving an identical inverter. (a) Logic diagram (all numbers are device widths), (b) Transistor-level circuit. (c) Switch-level model to compute rising delay, (d) Switch-level model for falling delay.

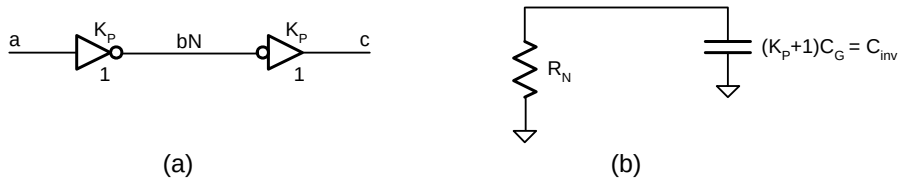


Figure 5.3: An inverter pair with equal rise-fall delays. (a) Logic diagram (sizings reflect parameters of Table 4.2). (b) Switch-level model of falling delay (rising delay is identical).

this time constant depends in equal parts on the driving and receiving gates, we cannot specify the delay of a gate by itself, but only as a function of output load.

Consider, for example, a CMOS inverter with a pullup of width W_P and a pulldown of width W_N driving an identical inverter, as shown in Figures 5.2(a) and (b).[2] For both rising and falling edges, the input capacitance of the second inverter is the sum of the capacitance of the PFET and NFET: $C_{\text{inv}} = (W_P + W_N)C_G$. When the output of the first inverter rises, the output resistance is that of the PFET with width W_P as shown in Figure 5.2(c): $R_P = K_{RP}/W_P = K_P K_{RN}/W_P$. Thus for a rising edge we have:

$$t_{pr} = R_P C_{\text{inv}} = \frac{K_P K_{RN}(W_P + W_N)C_G}{W_P}. \tag{5.1}$$

Similarly, for a falling edge, the output resistance is the resistance of the NFET pulldown as shown in Figure 5.2(d): $R_N = K_{RN}/W_N$. This gives a falling delay of:

$$t_{pf} = R_N C_{\text{inv}} = \frac{K_{RN}(W_P + W_N)C_G}{W_N}. \tag{5.2}$$

Most of the time we wish to size CMOS gates so that the rise and fall delays are equal; that is so $t_{pr} = t_{pf}$. For an inverter, this implies that $W_P = K_P W_N$, as show in Figure 5.3. We make the PFET K_P times wider than the NFET to account for the fact that its resistivity (per square) is K_P times larger. The PFET pull-up resistance becomes $R_P = K_{RP}/W_P = (K_P K_{RN})/(K_P W_N) = K_R N/W_N = R_N$. This gives equal resistance and hence equal delay. Equivalently, substituting for W_P in the formulae above gives.

$$t_{\text{inv}} = \frac{K_{RN}}{W_N}(K_P + 1)W_N C_G = (K_P + 1)K_{RN}C_G = (K_P + 1)\tau_N. \tag{5.3}$$

[2] W_P and W_N are in units of $W_{\min} = 8L_{\min}$. C_G here is the capacitance of a gate with width $8L_{\min}$, so $C_G = 1.6$fF.

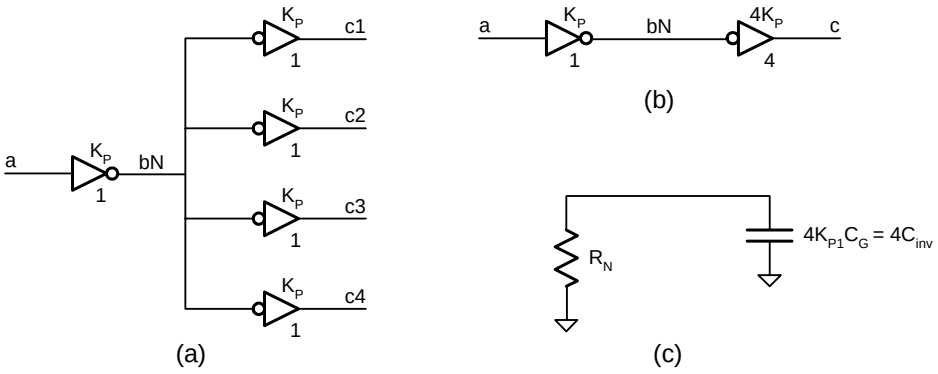


Figure 5.4: An inverter driving four-times its own load (a) Driving four other inverters. (b) Driving one large (4×) inverter. (c) Switch-level model of falling delay.

Note that the W_N term cancels out. The delay of an inverter driving an identical inverter, t_{inv} is independent of device width. As the devices are made wider R decreases and C increases leaving the total delay RC unchanged. For our model 0.13μm process with $K_P = 2.5$ this delay is $3.5\tau_N = 14$ps.[3]

Because the quantity $K_P + 1$ will appear frequently in our delay formulae, we will abbreviate this as $K_{P1} = K_P + 1$.

5.2 Fanout and Driving Large Loads

Consider the case where a single inverter of size 1 $W_N = W_{\min}$ sized for equal rise/fall delay ($W_P = K_P W_N$) drives four identical inverters as shown in Figure 5.4(a). The equivalent circuit for calculating the RC time constant is shown in Figure 5.4(c). Compared to the situation with identical inverters (fanout of one), this fanout-of-four situation has the same driving resistance, R_N, but four times the load capacitance, $4C_{\text{inv}}$. The result is that the delay for a fanout of four is four times the delay of the fanout of one circuit. In general, the delay for a fanout of F is F times the delay of a fanout of one circuit:

$$t_F = F t_{\text{inv}}. \tag{5.4}$$

The same situation occurs if the unit-sized inverter drives a single inverter that is sized four-times larger, as shown in Figure 5.4(b). The load capacitance on the first inverter is four times its input capacitance in both cases.

[3]For a minimum-sized $W_N = 8L_{\min}$ inverter, with equal rise/fall delay, $C_{\text{inv}} = 5.6fF$ in our model process.

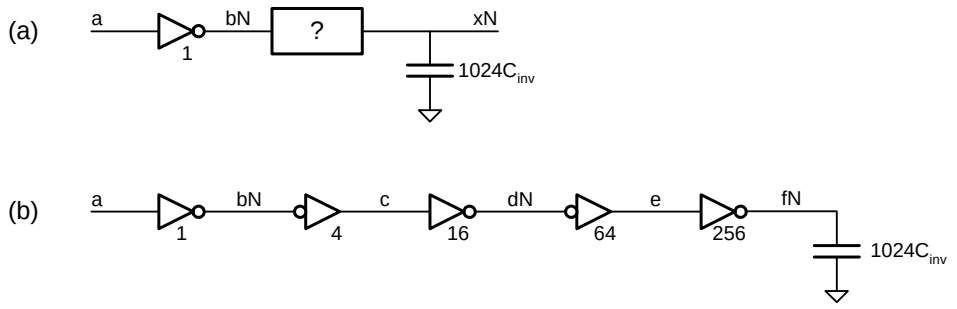


Figure 5.5: Driving a large capacitive load. (a) The output of a unit sized inverter needs to drive a fanout of 1024. We need a circuit to buffer up the signal bN to drive this large capacitance. (b) Minimum delay is achieved by using a chain of inverters that increases the drive by the same factor (in this case 4) at each stage.

When we have a very large fanout, it is advantageous to increase the drive of a signal in stages rather than all at once. This gives a delay that is logarithmic, rather than linear in the size of the fanout. Consider the situation shown in Figure 5.5(a). Signal bN, generated by a unit-sized inverter[4], must drive a load that is 1024 times larger than a unit-sized inverter (a fanout of $F = 1024$). If we simply connect bN to xN with a wire, the delay will be $1024t_{\text{inv}}$. If we increase the drive in stages, as shown in Figure 5.5(b), however, we have a circuit with five stages each with a fanout of four for a much smaller total delay of $20t_{\text{inv}}$.

In general, if we divide a fanout of F into n fanout of $\alpha = F^{1/n}$ stages, our delay will be

$$t_{Fn} = nF^{1/n}t_{\text{inv}} = \log_\alpha F\alpha t_{\text{inv}}. \tag{5.5}$$

We can solve for the minimum delay by taking taking the derivative of Equation (5.5) with respect to n (or α) and setting this derivative to zero. Solving shows that the minimum delay occurs for a fanout per stage of $\alpha = e$. In practice fanouts between 3 and 6 give good results. Fanouts much smaller than 3 result in too many stages while fanouts larger than 6 give too much delay per stage. A fanout of 4 is often used in practice. Overall, driving a large fanout, F, using multiple stages with a fanout of α reduces the delay from one that increases linearly with F to one that increases logarithmically with F — as $\log_\alpha F$.

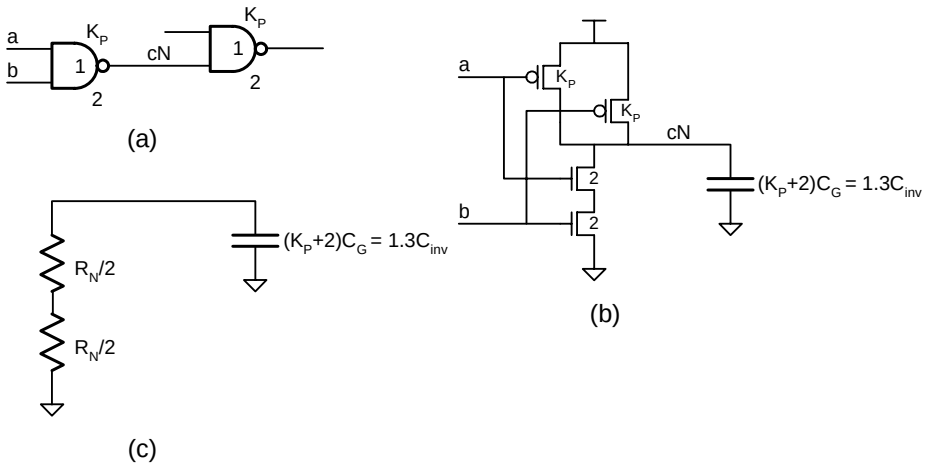


Figure 5.6: (a) A NAND gate driving an identical NAND gate. Both are sized for equal rise and fall delays. (b) Transistor-level schematic. (c) Switch-level model.

5.3 Fan-in and Logical Effort

Just as fan-out increases delay by increasing load capacitance, fan-in increases the delay of a gate by increasing output resistance — or equivalently input capacitance. To keep output drive constant, we size the transistors of a multi-input gate so that both the pull-up and pull-down the series resistance is equal to the resistance of an equal rise/fall inverter with the same relative size.

For example, consider a two-input NAND gate driving an identical NAND gate as shown in Figure 5.6(a). We size the devices of each NAND gate so each has the same worst-case up and down output resistance as a unit-drive equal rise/fall inverter as shown in Figure 5.6(b). Since in the worst-case only a single of the pullup PFETs is on, we size these PFETS $W_P = K_P$, just as in the inverter. We get no credit for the parallel combination of PFETs since both are on in only one of the three input states where the output is high (both inputs zero). To give a pull-down resistance equal to R_N each NFET in the series chain is sized at twice the minimum width. As shown in Figure 5.6(c) putting these two $R_N/2$ devices in series gives a total pull-down resistance of R_N. The capacitance of each input of this unit-drive NAND gate is the sum of the PFET and NFET capacitance: $(2 + K_P)C_G = \frac{2+K_P}{1+K_P}C_{\text{inv}}$.

We refer to this increase in input capacitance for the same output drive as the *logical effort* of the two input NAND gate. It represents the effort (in additional charge that must be moved compared to an inverter) to perform the

[4] From now on we may drop W_P from our diagrams whenever gates are sized for equal rise and fall.

2-input NAND logic function. The delay of a gate driving an identical gate (as in Figure 5.6(a)) is the product of its logical effort and $t_{\rm inv}$.

In general, for a NAND gate with fan-in F, we size the PFETs K_P and the NFETs F giving an input capacitance of:

$$C_{\rm NAND} = (F + K_P)C_G = \frac{F + K_P}{1 + K_P}C_{\rm inv}, \tag{5.6}$$

and hence a logical effort of:

$$LE_{\rm NAND} = \frac{F + K_P}{1 + K_P}, \tag{5.7}$$

and a delay of

$$t_{\rm NAND} = LE_{\rm NAND}t_{\rm inv} = \frac{F + K_P}{1 + K_P}t_{\rm inv}, \tag{5.8}$$

With a NOR gate the NFETs are in parallel, so a unit-drive NOR gate has NFETs pulldowns of size 1. In the NOR, the PFETs are in series, so a unit-drive NOR with a fan-in of F has PFET pullups of size FW_P. This gives a total input capacitance of:

$$C_{\rm NOR} = (1 + FK_P)C_G = \frac{1 + FK_P}{1 + K_P}C_{\rm inv}, \tag{5.9}$$

and hence a logical effort of:

$$LE_{\rm NOR} = \frac{1 + FK_P}{1 + K_P}. \tag{5.10}$$

For reference, Table 5.3 gives the logical effort as a function of fan-in, F, for NAND and NOR gates with 1 to 5 inputs both as functions of K_P and numerically for $K_P = 2.5$ (the value for our model process).

5.4 Delay Calculation

The delay of each stage i of a logic circuit is the product of its fanout or *electrical effort* from stage i to stage $i+1$ and the logical effort of stage $i+1$. The fanout is the ratio of the drive of stage i to stage $i+1$. The logical effort is the capacitance multiplier applied to the input of stage $i + 1$ to implement the logical function of that stage.

For example, consider the logic circuit shown in Figure 5.9. We calculate the delay from a to e one stage at a time as shown in Table 5.2. The first

Fan-in	Logical Effort			
F	$f(K_P)$		$K_P = 2.5$	
	NAND	NOR	NAND	NOR
1	1	1	1.00	1.00
2	$\frac{2+K_P}{1+K_P}$	$\frac{1+2K_P}{1+K_P}$	1.29	1.71
3	$\frac{3+K_P}{1+K_P}$	$\frac{1+3K_P}{1+K_P}$	1.57	2.43
4	$\frac{4+K_P}{1+K_P}$	$\frac{1+4K_P}{1+K_P}$	1.86	3.14
5	$\frac{5+K_P}{1+K_P}$	$\frac{1+5K_P}{1+K_P}$	2.14	3.86

Table 5.1: Logical effort as a function of fan-in for NAND and NOR gates (ignoring source/drain capacitance).

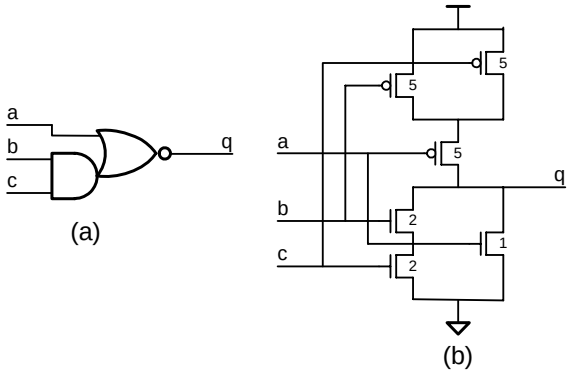


Figure 5.7: Logical effort of an AND-OR-Invert (AOI) gate. (a) Gate symbol. (b) Transistor-level schematic showing devices sized for equal rise/fall delays with unit drive.

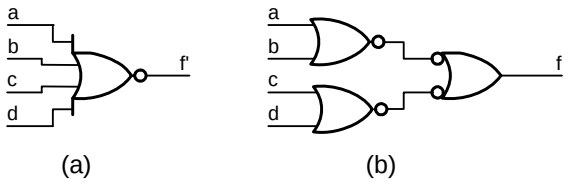


Figure 5.8: Choosing the number of stages for a logic function. Ignoring output polarity, we can implement a 4-input OR function as (a) a single 4-input NOR gate, or (b) two 2-input NOR gates feeding a 2-input NAND gate. The 4-input NOR gate has a logical effort of 3.14. The two-stage OR circuit has a logical effort of $1.71 \times 1.29 = 2.20$.

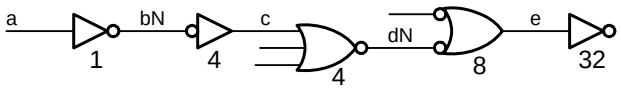


Figure 5.9: Logic circuit for example delay calculation. The number under each gate is its output drive (conductance) relative to a minimum sized inverter with equal rise/fall delays.

Driver	Signal	Fanout	Logical Effort	Delay
i	i to $i+1$	i to $i+1$	$i+1$	i to $i+1$
1	bN	4.00	1.00	4.00
2	c	1.00	2.43	2.43
3	dN	2.00	1.29	2.58
4	e	4.00	1	4.00
TOTAL				13.0

Table 5.2: Computing delay of a logic circuit. For each stage along the path we compute the fanout of the signal, and the logical effort of the gate receiving the signal. Multiplying the fanout by the logical effort gives the delay per stage. Summing over the stages gives the total delay.

stage, that drives signal bN for example has a fanout of 4 and the logical effort of the following stage (an inverter) is 1, so the total delay of this stage, is 4. The second stage, driving signal c, has a fanout of 1, both this stage and the next have a drive of 4. Signal c drives a 3-input NOR gate which has a logical effort of 2.43, so the total delay of this stage is 2.43. The third stage, driving signal dN, has both fanout and logical effort. The fanout of this stage is 2 (4 driving 8) and the logical effort is that of the two-input NAND, 1.29, for a total delay of $2 \times 1.29 = 2.58$. Finally the fourth stage, driving signal e has a fanout of 4 and logical effort of 1. We do not compute the delay of the final inverter (with drive 32). It is shown simply to provide the load on signal e. The total delay is determined by summing the delays of the four stages $t_{pae} = (4 + 2.43 + 2.58 + 4)t_{\text{inv}} = 13.0t_{\text{inv}} = 182\text{ps}$.

When we are computing the maximum delay of a circuit with fan-in, in addition to calculating the delay along a path (as shown in Table 5.2), we also need to determine the longest (or critical) path. For example, in Figure 5.10 suppose input signals a and p change at the same time, at time $t = 0$. The calculation is shown in Table 5.3. The delay from a to c is $6.53t_{\text{inv}}$ while the delay from p to qN is $1.57t_{\text{inv}}$. Thus, when calculating maximum delay, the critical path is from a to c to dN — a total delay of $14.53t_{\text{inv}}$. If we are concerned with the minimum delay of the circuit, then we use the path from p to qN to dN — with total delay $9.57t_{\text{inv}}$.

Some logic circuits include fanout to different gate types as shown for signal

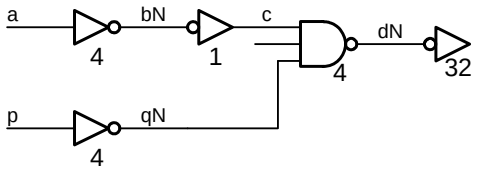


Figure 5.10: Logic circuit with fan-in. Inputs a and p change at the same time. The critical path for maximum delay is the path from a to c to dN.

Signal	Fanout	Logical Effort	Delay
i to $i+1$	i to $i+1$	$i+1$	i to $i+1$
bN	0.25	1	0.25
c	4	1.57	6.68
Subtotal a to c			6.53
qN	1	1.57	1.57
Subtotal p to qN			1.57
dN	8	1	8
TOTAL a to dN			14.53
TOTAL p to dN			9.57

Table 5.3: Delay calculation for both paths of Figure 5.10.

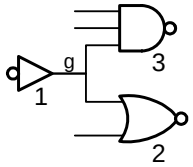


Figure 5.11: Logic circuit with fan-out to different gate types. The total effort of signal g is calculated by summing the product of fanout and logic effort across all receiving gates.

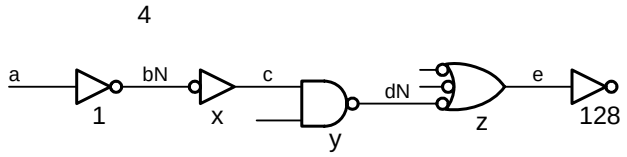


Figure 5.12: Unsized logic circuit. The sizes x, y, and z of the three middle stages must be chosen to minimize delay by equalizing the delay of each stage, and adding stages if needed.

Driver	Signal	Fanout	Logical Effort	Delay
i	i to $i+1$	i to $i+1$	$i+1$	i to $i+1$
1	bN	$x = 4.00$	1.00	$x = 4$
2	c	$y = 3.10$	1.29	$1.29y = 4$
3	dN	$z = 2.55$	1.57	$1.57z = 4$
4	e	$128/xyz$	1	$128/xyz = 4.04$
TOTAL				16.04

Table 5.4: Optimizing gate sizes to minimize delay. The total effort is determined and divided evenly across the stages.

g in Figure 5.11. In this case, we compute the fanout and logical effort for each fanout of signal g. The upper NAND gate has a fanout of 3 with a logical effort of 1.57 for a total effort of 4.71. The lower NOR gate has a fanout of 2 and a logical effort of 1.71 for a total effort of 3.42. Thus, the total delay (or effort) of signal g is $8.13t_{\text{inv}}$.

5.5 Optimizing Delay

To minimize the delay of a logic circuit we size the stages so that there is an equal amount of effort per stage. For a single n-stage path, a simple way to perform this optimization is to compute the total effort along the path, TE, and then divide this effort evenly across the stages by sizing each stage to have a total effort (product of fanout and logical effort) of $TE^{1/n}$.

Consider, for example, the circuit of Figure 5.12. The delay calculation for this circuit is shown in Table 5.4. The ratio of the first and last gates specify the total amount of fanout required, 128. We multiply this electrical effort with the logical effort of stages 3 and 4, 1.29 and 1.57 respectively, to give the total effort of 259. We then take $259^{1/4} \approx 4$ as the total effort (or delay) per stage. Thus, $x = 4$, $y = 4/1.29 = 3.10$, and $z = 2.55$. This gives a total delay of just over $16t_{\text{inv}}$.

Suppose the final inverter in Figure 5.12 was sized with a drive of 2,048 rather than 128. In that case the total effort is $TE = 2,048 \times 1.29 \times 1.57 \approx 4,148$. If we

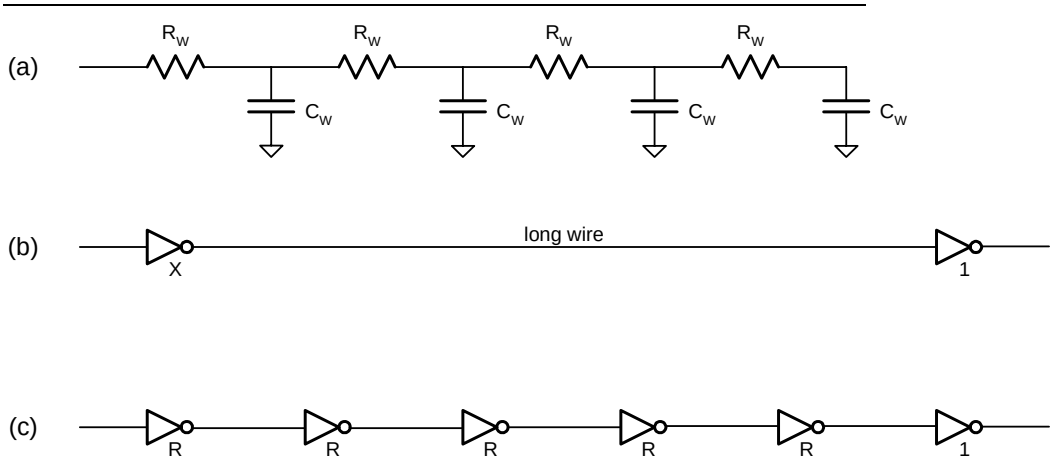


Figure 5.13: (a) A long on-chip wire has significant series resistance R_w and parallel capacitance C_w giving it a delay that grows quadratically with length. (b) Driving a long wire often gives unacceptable delay and rise time. Increasing the size X of the driver does not help due to the resistivity of the line. (c) The delay of the line can be made linear, rather than quadratic, with length by inserting repeaters at a fixed interval in the line.

attempt to divide this into four stages we would get a delay of $4,148^{1/4} = 8t_{\text{inv}}$ per stage, which is a bit high, giving a total delay of about $32t_{\text{inv}}$. In this case, we can reduce the delay by adding an even number of inverter stages, as in the example of Figure 5.5. The optimum number of stages is $\ln 4,148 \approx 8$. With 8 stages, each stage must have an effort of 2.83, giving a total delay of $22.6t_{\text{inv}}$. A compromise circuit is to aim for a delay of 4 per stage which requires $\log_4 4,148 \approx 6$ stages for a total delay of $24t_{\text{inv}}$.

If we are to add either 2 or 4 inverters to the circuit of Figure 5.12 we must decide where to add them. We could insert a pair of inverters at any stage of the circuit without changing its function. We could even insert individual inverters at arbitrary points if we are willing to convert the NANDs to NORs (which is generally a bad idea as it increases total effort.) However, it is usually best to place the extra stages *last* to avoid the extra power that would otherwise be consumed if the high logical effort stages were sized larger. However, if one of the signals has a large wire load, it may be advantageous to insert one or more of the extra stages before that point to ensure adequate drive for the wire.

5.6 Wire Delay

On modern integrated circuits a large fraction of delay and power is due to driving the wires that connect gates. An on-chip wire has both resistance and

Parameter	Value	Units	Description
R_w	0.25	Ω/square	Resistance per square
	1	Ω/μm	Resistance per μm
C_w	0.2	fF/μm	Capacitance per μm
τ_w	0.2	fs/μm^2	RC time constant

Table 5.5: Resistance and capacitance of wires in an 0.13μm process.

capacitance. Typical values for an 0.13μm process are shown in Table 5.5.

Wires that are short enough that their total resistance is small compared to the output resistance of the driving gate can be modeled as a lumped capacitance. For example, a minimum-sized ($W_N = 8L_{\min}$) inverter has an output resistance of 2.5kΩ. A wire of less than 500μm in length has a total resitance less than one fifth of this amount and can be considered a lumped capacitance. A wire of exactly 500μm, for example, can be modeled as a capacitance of 100fF, the equivalent of a fanout of 17 compared to the 5.6fF input capacitance of the minimum sized inverter.

For larger drivers, shorter wires have a resistance that is comparable to the driver output resistance. For a 16$\times$ minimum sized inverter with an output resistance of 156Ω, for example, a wire of length 156μm has a resitance equal to the output resistance of the driver and one must get down to a length of 31μm for the resistance to be less than one fifth of the driver resistance.

For wires that are long enough for their resistance to be significant compared to the resistance of their driver, the delay of the wire increases quadratically with wire length. As illustrated in Figure 5.13(a) as the wire gets longer both the resistance and the capacitance of the wire increase linearly causing the RC time constant to increase quadratically. Increasing the size of the driver as shown in Figure 5.13(b) does not improve the situation because the resistance is dominated by the wire resistance, so reducing the driver resistance does not substantially reduce the delay.

To make the delay of a long wire linear (rather than quadratic) with length, the wire can be divided into sections with each section driven by a *repeater* as shown in Figure 5.13(c). The optimum repeater spacing occurs when the delay due to the repeater equals the delay due to the wire segment between repeaters.

5.7 Power Dissipation in CMOS Circuits

In a CMOS chip, almost all of the power dissipation is due to charging and discharging the capacitance of gates and wires. The energy consumed charging the gate of an inverter from V_0 to V_1 and then discharging it again to V_0 is.

$$E_{\text{inv}} = C_{\text{inv}} V^2 \tag{5.11}$$

For our 0.13μm, with $C_{\text{inv}} = 5.6$fF and $V = V_1 - V_0 = 1.2$V, $E_{\text{inv}} = 8.1$fJ.

A remarkable property of CMOS circuits is that the energy E consumed by a function is proportional to L^3. This is because both capacitance and voltage scale linearly with L. Thus, as we halve the gate length L from 0.13μm to 65nm, we expect C_{inv} for a minimum-size inverter to halve from 5.6fF to 2.8fF, the voltage V to halve from 1.2V to 0.6V, and the switching energy E_{inv} to reduce by a factor of eight from 8.1fJ to about 1fJ.[5]

The power consumed charging and discharging this inverter depends on how often it transitions. For a circuit with capacitance C that operates at a frequency f and has α transistions each cycle, the power consumed is:

$$P = 0.5CV^2f\alpha \tag{5.12}$$

The factor of 0.5 is due to the fact that half of the energy is consumed on the charging transition and the other half on the discharge. For an inverter with activity factor $\alpha = 0.33$ and a clock rate of $f = 500$MHz, $P = 665$nW.

To reduce the power dissipated by a circuit we can reduce any of the terms of Equation (5.12). If we reduce voltage, power reduces quadratically. However the circuit also operates slower at a lower voltage. For this reason we often reduce V and f together, getting a factor of eight reduction in power each time we halve V and f. Reducing capacitance is typically accomplished by making our circuit as physically small as possible — so that wire length, and hence wire capacitance is as small as possible.

The activity factor, α, can be reduced through a number of measures. First, it is important that the circuit not make unnecessary transitions. For a combinational circuit, each transition of the inputs should result in at most one transition of each output. Glitches or hazards (see Section 6.10) should be eliminated as they result in unecessary power dissipation. Activity factor can also be reduced by gating the clock to unused portions of the circuit, so that these unused portions have no activity at all. For example, if an adder is not being used on a particular cycle, stopping the clock to the adder stops all activity in the adder saving considerable power.

Up to now we have focused on dynamic power — the power due to charging and discharging capacitors. As gate lengths and supply voltages shrink, however, static *leakage* power is becoming an increasingly important factor. Leakage current is the current that flows through a MOSFET when it is in the off state. This current is proportional to $\exp(-V_T)$. Thus, as threshold voltage decreases, leakage current increases exponentially. Today, leakage current is only a factor in circuits with very low activity factors. However, with continued scaling leakage current will ultimately become a dominant factor and will limit the ability to continue scaling supply voltage.

[5] This cubic scaling of energy and power with gate length cannot continue indefinitely because threshold voltage, and hence supply voltage must be maintained above a minimum level to prevent leakage current from dominating power dissipation.

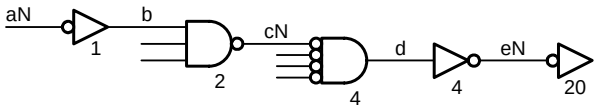


Figure 5.14: Circuit for Exercises 5–7 and 5–10.

5.8 Bibliographic Notes

Mead and Rem first described the exponential horn for driving large capacitive loads. Sutherland and Sproull introduced the notion of logical effort. Harris, Sutherland, and Sproull have written a monograph describing this concept and its application in detail.

5.9 Exercises

5–1 Compute delay of some complex CMOS gates.

5–2 *Sizing of CMOS gates.* Consider a 4-input static CMOS gate that implements the function $f = \overline{a \wedge (b \vee (c \wedge d))}$.
(a) Draw a schematic symbol for this gate - with the bubble on the output.
(b) Draw a transistor schematic for this gate and size the transistors for rise and fall delay equal to a minimum-sized inverter with equal rise/fall.
(c) Compute the logical effort of this gate.

5–3 *Sizing of CMOS gates.* Repeat Exercise 5–2 for a gate that implements the function $f = \overline{(a \wedge b) \vee (c \wedge d)}$.

5–4 Consider an inverter with output capacitance equal to η times its input capacitance.
(a) What is the delay of a fanout of one inverter considering this output capacitance?
(b) What is the delay of a fanout of F inverter considering this output capacitance.

5–5 Compute logical effort of some complex CMOS gates.

5–6 Choose size and number of inverters to drive a large load.

5–7 *Delay calculation.* Calculate the delay of the circuit in Figure 5.14.

5–8 *Delay calculation.* Calculate the delay of the circuit in Figure 5.15.

5–9 *Delay calculation.* Calculate the delay of the circuit in Figure 5.16.

5–10 *Delay optimizaiton.* Resize the gates in Figure 5.14 to give minimum delay. You may not change the size of input or output gates.

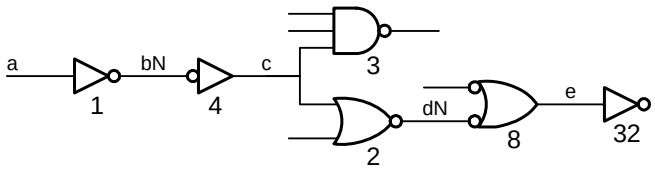


Figure 5.15: Circuit for Exercises 5–8 and 5–11.

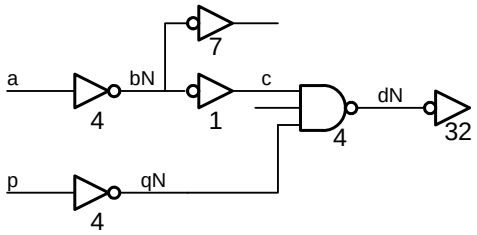


Figure 5.16: Circuit for Exercises 5–9 and 5–12.

5–11 *Delay optimizaiton.* Resize the gates in Figure 5.15 to give minimum delay. You may not change the size of input or output gates.

5–12 *Delay optimizaiton.* Resize the gates in Figure 5.16 to give minimum delay. You may not change the size of input or output gates.

5–13 Logical effort with output capacitance.

5–14 Compute switching energy of some logic functions.

5–15 Look at ways to reduce power in a circuit.

Chapter 6

Combinational Logic Design

Combinational logic circuits implement logical functions. Used for control, arithmetic, and data steering, combinational circuits are the heart of digital systems. Sequential logic circuits (see Chapter 14) use combinational circuits to generate their next state functions.

In this chapter we introduce combinational logic circuits and describe a procedure to design these circuits given a specification. At one time, before the mid 1980s, such manual synthesis of combinational circuits was a major part of digital design practice. Today, however, designers write the specification of logic circuits in a hardware description language (like Verilog) and the synthesis is performed automatically by a computer-aided design (CAD) program.

We describe the manual synthesis process here because every digital designer should understand how to generate a logic circuit from a specification. Understanding this process allows the designer to better use the CAD tools that perform this function in practice, and, on rare occasions, to manually generate critical pieces of logic by hand.

6.1 Combinational Logic

As illustrated in Figure 6.1, a combinational logic circuit generates a set of outputs whose state depends only on the *current* state of the inputs. Of course, when an input changes state, some time is required for an output to reflect this change. However, except for this *delay* the outputs do not reflect the *history* of the circuit. With a combinational circuit, a given input state will always produce the same output state regardless of the sequence of previous input states. A circuit where the output depends on previous input states is called a *sequential* circuit (see Chapter 14).

For example, a majority circuit, a logic circuit that accepts n inputs and outputs a 1 if at least $\lfloor n/2+1 \rfloor$ of the inputs are 1, is a combinational circuit. The output depends only on the number of 1s in the present input state. Previous input states do not effect the output.

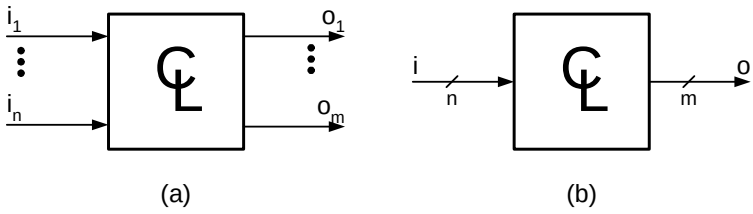


Figure 6.1: A combinational logic circuit produces a set of outputs $\{o_1, \ldots, o_m\}$ that depend only on the *current* state of a set of inputs $\{i_1, \ldots, i_n\}$. (a) Block CL is shown with n inputs and m outputs. (b) Equivalent block with n inputs and m outputs shown as buses.

On the other hand, a circuit that outputs a 1 if the number of 1s on the n inputs is greater than the previous input state is sequential (not combinational). A given input state, e.g., $i_k = 011$, can result in $o = 1$ if the previous input was $i_{k-1} = 010$, or it can result in $o = 0$ if the previous input was $i_{k-1} = 111$. Thus, the output depends not just on the present input, but also on the history (in this case very recent history) of previous inputs.

Combinational logic circuits are important because their static nature makes them easy to design and analyze. As we shall see, general sequential circuits are quite complex in comparison. In fact, to make sequential circuits tractable we usually restrict ourselves to *synchronous* sequential circuits which use combinational logic to generate a next state function (see Chapter 14).

Please note that logic circuits that depend only on their inputs are *combinational* and **not** *combinatorial*. While these two words sound similar, they mean different things. The word *combinatorial* refers to the mathematics of counting, not to logic circuits. To keep them straight, remember that combinational logic circuits *combine* their inputs to generate an output.

6.2 Closure

A valuable property of combinational logic circuits is that they are closed under *acyclic* composition. That is, if we connect together a number of combinational logic circuits — connecting the outputs of one to the inputs of another — and avoid creating any loops — that would be cyclic — the result will be a combinational logic circuit. Thus we can create large combinational logic circuits by connecting together small combinational logic circuits.

An example each of acyclic and of cyclic composition is shown in Figure 6.2. A combinational circuit realized by acyclically composing two smaller combinational circuits is shown in Figure 6.2(a). The circuit in Figure 6.2(b), on the other hand, is not combinational. The cycle created by feeding the output of the upper block into the input of the lower block creates state. The value of this

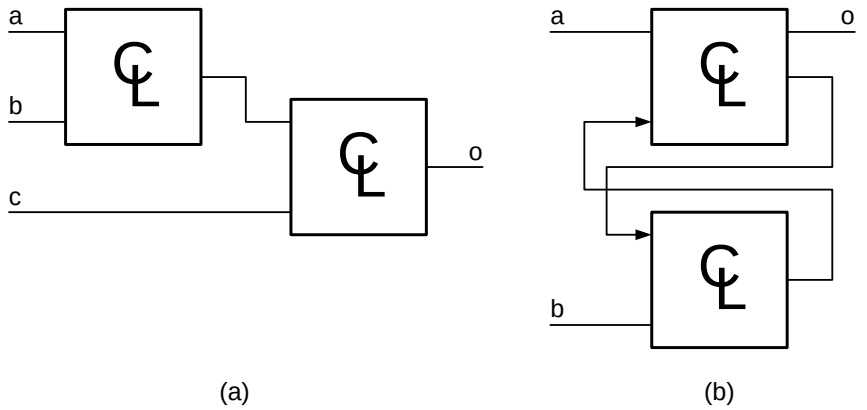


Figure 6.2: Combinational logic circuits are closed under *acyclic* composition. (a) This acyclic composition of two combinational logic circuits is itself a combinational logic circuit. (b) This cyclic composition of two combinational logic circuits is *not* combinational. The feedback of the cyclic composition creates internal state.

feedback variable can *remember* the history of the circuit. Hence the output of this circuit is not just a function of its inputs. In fact, we shall see that *flip-flops*, the building blocks of most sequential logic circuits are built using exactly the type of feedback shown in Figure 6.2(b).

It is easy to prove that acylic compositions of combinational circuits are themselves combinational by induction, starting at the input and working toward the output. Let a combinational block whose inputs are connected only to primary inputs (i.e., not to the outputs of other blocks) be a rank 1 block. Similarly, let a block whose inputs are connected only to primary inputs and/or to the outputs of blocks of ranks 1 through k be a rank $k+1$ block. By definition, all rank 1 blocks are combinational. Then, if we assume that all blocks of ranks 1 to k are combinational, then a rank $k+1$ block is also combinational. Since its outputs depend only on the current state of its inputs, and since all of its inputs depend only on the current state of the primary inputs, its outputs also depend only on the current state of the primary inputs.

6.3 Truth Tables, Minterms, and Normal Form

Suppose we want to build a combinational logic circuit that outputs a one when its four-bit input represents a prime number in binary. One way to represent the logic function realized by this circuit is with an English-language description — as we have just specified it. However, we generally prefer a more precise definition.

No.	in	out
0	0000	0
1	0001	1
2	0010	1
3	0011	1
4	0100	0
5	0101	1
6	0110	0
7	0111	1
8	1000	0
9	1001	0
10	1010	0
11	1011	1
12	1100	0
13	1101	1
14	1110	0
15	1111	0

Table 6.1: Truth table for a four-bit prime number circuit. The column *out* shows the output of the circuit for each of the 16 input combinations.

Often we start with a *truth table* that shows the output value for each input combination. Table 6.1 shows a truth table for the four-bit prime number function. For an n-input function, a truth table has 2^n rows (16 in this case), one for each input combination. Each row lists the output of the circuit for that input combination (0 or 1 for a one-bit output).

Of course, it is a bit redundant to show both the zero and one outputs in the table. It suffices to show just those input combinations for which the output is one. Such an abbreviated table for our prime number function is shown in Table 6.2.

The reduced table (Table 6.2) suggests one way to implement a logic circuit that realizes the prime function. For each row of the table, an AND gate is connected so that the output of the AND is true only for the input combination shown in that row. For example, for the first row of the table, we use an AND gate connected to realize the function $f_1 = \overline{d} \wedge \bar{c} \wedge \bar{b} \wedge a$ (where d, c, b, and a are the four bits of *in*). If we repeat this process for each row of the table, we get the complete function:

$$f = (\overline{d} \wedge \overline{c} \wedge \overline{b} \wedge a) \vee (\overline{d} \wedge \overline{c} \wedge b \wedge \overline{a}) \vee (\overline{d} \wedge \overline{c} \wedge b \wedge a) \vee (\overline{d} \wedge c \wedge \overline{b} \wedge a) \\ \vee (\overline{d} \wedge c \wedge b \wedge a) \vee (d \wedge \overline{c} \wedge b \wedge a) \vee (d \wedge c \wedge \overline{b} \wedge a). \quad (6.1)$$

Figure 6.3 shows a schematic logic diagram corresponding to Equation (6.1). The seven AND gates correspond to the seven product terms of Equation (6.1) which in turn correspond to the seven rows of Table 6.2. The output of each

No.	in	out
1	0001	1
2	0010	1
3	0011	1
5	0101	1
7	0111	1
11	1011	1
13	1101	1
	otherwise	0

Table 6.2: Abbreviated truth table for a four-bit prime number circuit. Only inputs for which the output is 1 are listed explicitly.

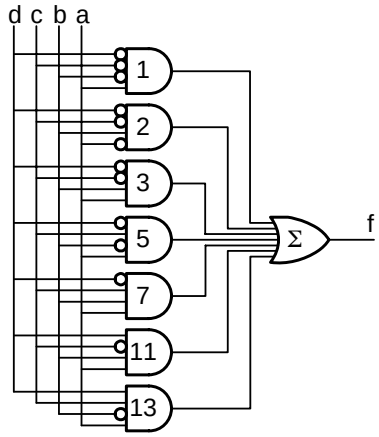


Figure 6.3: A four-bit prime-number circuit in conjunctive (sum-of-products) normal form. An AND gate generates the minterm associated with each row of the truth table that gives a true output. An OR gate combines the minterms giving an output that is true when the input matches any of these rows.

AND gate goes high when the inputs match the input values listed in the corresponding row of the truth table. For example, the output of the AND gate labeled 5 goes high when the inputs are 0101 (binary 5). The AND gates feed a 7-input OR gate which outputs high if any of the AND gates have a high output, that is if the input matches 1, 2, 3, 5, 7, 11, or 13 — which is the desired function.

Each product term in Equation (6.1) is called a *minterm*. A minterm is a product term that includes each input of a circuit or its complement. Each of the terms of Equation (6.1) includes all four inputs (or their complements). Thus they are minterms. The name minterm derives from the fact that these four-input product terms represent a minimal number of input states (rows of the truth table), just one. As we shall see in the next section, we can write product terms that represent multiple input states — in effect combining minterms.

We can write Equation (6.1) in shorthand as:

$$f = \sum_{\text{in}} m(1, 2, 3, 5, 7, 11, 13), \tag{6.2}$$

to indicate that the output is the sum (OR) of the minterms listed in the parentheses.

You will recall from Section 3.4 that expressing a logic function as a sum of minterms is a *normal form* that is unique for each logic function. While this form is unique, its not particularly efficient. We can do much better by combining minterms into simpler product terms that each represent multiple lines of our truth table.

6.4 Implicants and Cubes

An examination of Table 6.2 reveals several rows that differ in only one position. For example, the rows 0010 and 0011 differ only in the rightmost (least signficant) position. Thus, if we allow bits of *in* to be X (matches either 0 or 1), we can replace the two rows 0010 and 0011 by the single row 001X. This new row 001X corresponds to a product term that includes just three of the four inputs (or their complements):

$$f_{001\text{X}} = \overline{d} \wedge \overline{c} \wedge b = (\overline{d} \wedge \overline{c} \wedge b \wedge \overline{a}) \vee (\overline{d} \wedge \overline{c} \wedge b \wedge a). \tag{6.3}$$

The 001X product term subsumes the two minterms corresponding to 0010 and 0011 because it is true when at least one of them is true and nowhere else. Thus, in a logic function we can replace the two minterms for 0010 and 0011 with the simpler product term for 001X without changing the function.

A product term like 001X ($\overline{d} \wedge \overline{c} \wedge b$) that is only true when a function is true is called an *implicant* of the function. This is just a way of saying that the product term *implies* the function. A minterm may or may not be an implicant of a function. The minterm 0010 ($\overline{d} \wedge \overline{c} \wedge \overline{b} \wedge a$) is an implicant of the prime

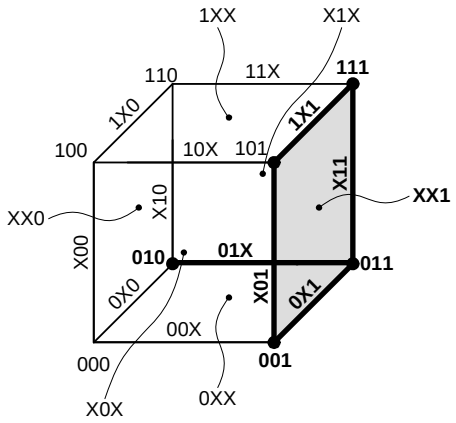


Figure 6.4: A cube visualization of the three-bit prime number function. Each vertex corresponds to a minterm, each edge to a product of two variables, and each face to a single variable. The bold vertices, edges and the shaded face show implicants of the three-bit prime number function.

function because it implies the function — when 0010 is true, the function is true. However, minterm 0100 ($\overline{d} \wedge c \wedge \overline{b} \wedge \overline{a}$) is a minterm, it is a product that includes each input or its complement, but it is not an implicant of the prime function. When 0100 is true, the prime function is false because 4 is not a prime. If we say that a product is a *minterm of a function* we are saying that it is both a minterm, and an implicant of the function.

It is often useful to visualize implicants on a *cube* as shown in Figure 6.4. This figure shows a three-bit prime number function mapped onto a three-dimensional cube. Each vertex of the cube represents a minterm. The cube makes it easy to see which minterms and implicants can be combined into larger implicants.[1] Minterms that differ in just one variable (e.g., 001 and 011) are adjacent to each other and the edge between two vertices (e.g., 01X) represents the product that includes the two minterms (the OR of the two adjacent minterms). Edges that differ in just one variable (e.g., 0X1 and 1X1) are adjacent on the cube and the face between the edges represents the product that includes the two edge products (e.g., XX1). In this figure, the three-bit prime number function is shown as five bold vertices (001, 010, 011, 101, and 111). Five bold edges connecting these vertices represent the five two variable implicants of the function (X01, 0X1, 0X1, X11, and 1X1). Finally, the shaded face (XX1) represents the single one variable implicant of the function.

A cube representation of the full four-bit prime number function is shown

[1] One implicant is larger than another if it contains more minterms. For example, implicant 001 has size 1 because it contains just one minterm. Implicant 01X has size 2 because it contains two minterms (001 and 011) and hence is larger.

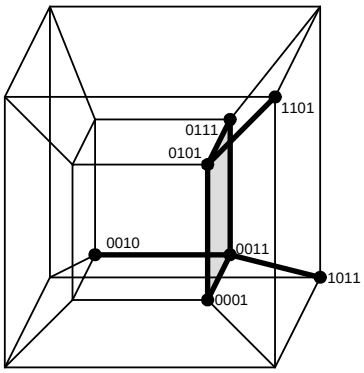


Figure 6.5: A cube visualization of the four-bit prime number function.

Number of variables			
4	3	2	1
0001	**001X**	**0XX1**	
0010	00X1		
0011	0X01		
0101	0X11		
0111	01X1		
1011	**X011**		
1101	**X101**		

Table 6.3: All implicants of the 4-bit prime number function. Prime implicants are shown in bold.

in Figure 6.5. To avoid clutter only the minterms of the function are labeled. To represent four variables, we draw a four-dimensional cube as two three-dimensional cubes, one within the other. As before, vertices represent minterms, edges represent products with one X, and faces represent products with two Xs. In four dimensions, however, we also have eight volumes that represent products with three Xs. For example, the outside cube represents 1XXX — all minterms where the leftmost (most significant) bit d is true. The four-bit prime number function has seven vertices (minterms). Connecting adjacent vertices gives seven edges (implicants with a single X). Finally, connecting adjacent edges gives a single face (implicant with two Xs). All of these implicants of the four-bit prime number function are shown in Table 6.3.

Computer programs that synthesize and optimize logic functions, such as we will be using in this class, use an internal representation of logic functions as a set of implicants where each implicant is represented as a vector with elements 0, 1, or X. To simplify a function, the first step is to generate all of the implicants of the function, such as that shown in Table 6.3. A systematic procedure to do

this is to start with all minterms of the function (the '4' column of Table 6.3). For each minterm, attempt to insert an X into each variable position. If the result is an implicant of the function, insert it in a list of single X implicants (the '3' column of Table 6.3). Then for each implicant with one X, attempt to insert an X into each of the remaining non X positions and if the result is an implicant, insert it in a list of two X implicants. The process is repeated for two X implicants and so on until no further implicants are generated. Such a procedure will, given a list of minterms, generate a list of implicants.

If an implicant x has the property that replacing any 0 or 1 digit of x with an X results in a product that is not an implicant, then we call x a *prime implicant.*[2] A prime implicant is an implicant that cannot be made any *larger* and still be an implicant. The prime implicants of the prime number function are shown in bold in Table 6.3.

If a prime implicant of a function x is the only prime implicant that contains a particular minterm of the function y, we say that x is an *essential prime implicant.* x is essential because no other prime implicant includes y. Without x a collection of prime implicants will not include minterm y. All four of the prime implicants of the four-bit prime number function are essential. Implicant 0XX1 is the only prime implicant that includes 0001 and 0111. Minterm 0010 is included only in prime implicant 001X, X101 is the only prime implicant that includes 1101, and 1011 is only included in prime implicant X011.

6.5 Karnaugh Maps

Because it is inconvenient to draw cubes (especially in 4 or more dimensions), we often use a version of a cube flattened into two two dimensions called a *Karnaugh map* (or K-map for short). Figure 6.6(a) shows how four variable minterms are arranged in a 4-variable K-map. Each square of a K-map corresponds to a minterm, and the squares of the K-map in Figure 6.6(a) are labeled with their minterm numbers. A pair of variables is assigned to each dimension and sequenced using a Gray code so that only one variable changes as we move from one square to another across a dimension — including the wrap-around from the end back to the beginning. In Figure 6.6(a) for example, we assign the rightmost two bits *ba* of the input *dcba* to the horizontal axis. As we move along this axis, these two bits (*ba*) take on the values 00, 01, 11, and 10 in turn. We map the leftmost bits *dc* to the vertical axis in a similar manner. Because only one variable changes from column to column and from row to row (including wrap arounds), two minterms that differ in only one variable are adjacent in the K-map, just as they are adjacent in the cube representation.

Figure 6.6(b) shows a K-map for the four-bit prime number function. The contents of each square is either a 1 which indicates that this minterm is an implicant of the function, or a 0 to indicate that it is not. Later we will allow squares to contain an X to indicate that the minterm may or may not be an implicant — i.e., it is a *don't care.*

[2]The use of the word '*prime*' here has nothing to do with the prime number function.

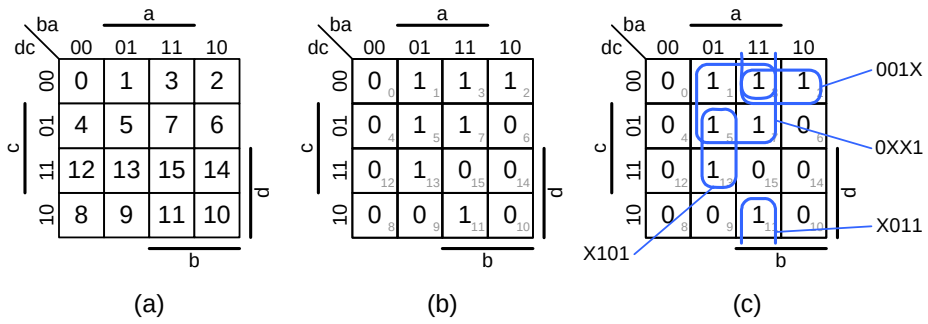


Figure 6.6: A Karnaugh map (K-map) for the four-bit prime number function. Inputs a and b change along the horizontal axis while inputs c and d change along the vertical axis. The map is arranged so that each square is adjacent (including wraparound) to all squares that correspond to changing exactly one input variable. (a) The arrangement of minterms in a 4-variable K-map. (b) The K-map for the 4-bit prime number function. (c) The same K-map with the four prime implicants of the function identified. Note that implicant X011 wraps around from top to bottom.

Figure 6.6(b) shows how the adjacency property of a K-map, just like the adjacency property of a cube, makes it easy to find larger implicants. The figure shows the prime implicants of the prime number function identified on the K-map. The three implicants of size two (single X) are pairs of adjacent 1s in the map. For example, implicant X011 is the pair of 1s in the $ab = 11$ column that wraps from top to bottom ($c = 0$). An implicant of size four contains four 1s and may be either a square, as is the case for 0XX1, or may be a full row or column, none in this function. For example, the product XX00 corresponds to the leftmost column of the K-map.

Figure 6.7 shows the arrangement of minterms for K-maps with 2, 3, and 5 variables. The 5-variable K-map consists of two four variable K-maps side by side. Corresponding squares of the two K-maps are considered to be *adjacent* in that their minterms differ only in the value of variable e. K-maps with up to 8-variables can be handled by creating a four-by-four array of 4-variable K-maps.

6.6 Covering a Function

Once we have a list of implicants for a function, the problem remains to select the least expensive set of implicants that *cover* the function. A set of implicants is a cover of a function if each minterm of the function is included in at least one implicant of the cover. We define the cost of an implicant as the number of variables in the product. Thus, for a four-variable function a minterm like 0011 has cost 4, a one X implicant like 001X has cost 3, a two X implicant like 0XX1

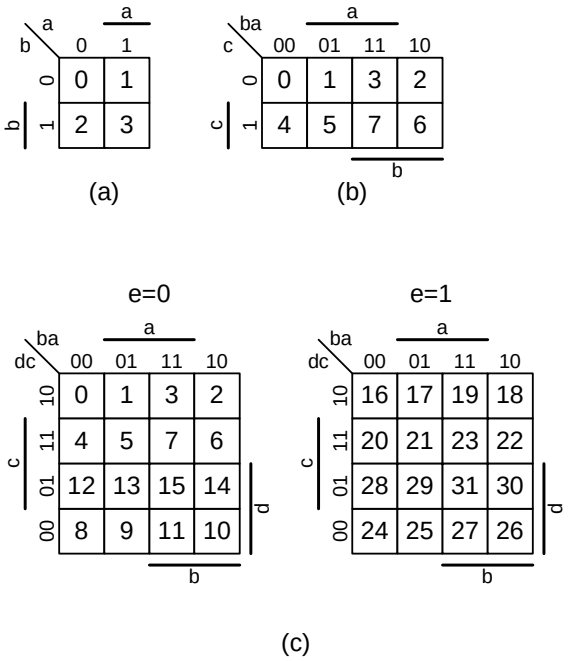


Figure 6.7: Position of minterms in K-maps of different sizes. (a) a two-variable K-map, (b) a 3-variable K-map, (c) a 5-variable K-map.

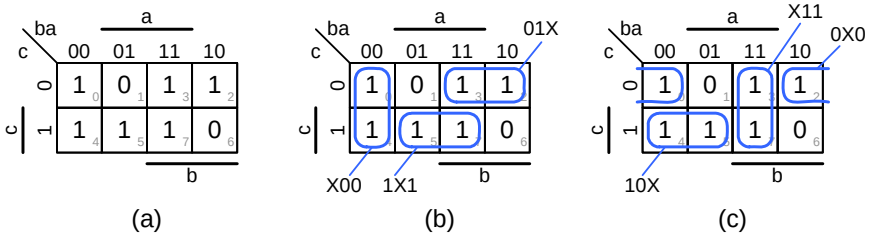


Figure 6.8: A function with a non-unique minimum cover and no essential prime implicants. (a) K-map of the function. (b) One cover contains X00, 1X1, and 01X. (c) A different cover contains 10X, X11, and 0X0.

has cost 2, and so on.

A procedure to select an inexpensive set of implicants is as follows:

1. Start with an empty cover.
2. Add all essential prime implicants to the cover.
3. For each remaining uncovered minterm, add the largest implicant that covers that minterm to the cover.

This procedure will always result in a *good* cover. However, there is no guarantee that is will give the lowest-cost cover. Depending on the order in which minterms are covered in step 3, and the method used to select between equal cost implicants to cover each minterm different covers of possibly different cost may result.

For the four-bit prime-number function, the function is completely covered by the four essential prime implicants. Thus, the synthesis process is done after step 2 and the cover is both minimum and unique.

Consider, however the logic function shown in Figure 6.8(a). This function has no essential prime implicants so our process moves to step 3 with an empty cover. At step 3, suppose we select uncovered minterms in numerical order. Hence we start with minterm 000. We can cover 000 with either X00 or 0X0. Both are minterms of the function. If we choose X00 the cover shown in Figure 6.8(b) will result. If instead we choose 0X0 we get the cover shown in Figure 6.8(c). Both of these covers are minimal - even if they aren't unique.

It is also possible for this procedure to generate a non-minimal cover. In the K-map of Figure 6.8, suppose we initially select implicant X00 and then select implicant X11. This is possible since it is one of the largest (size 2) implicants that covers an uncovered minterm. However, if we make this choice, we can no longer cover the function in 3 minterms. It will take 4 minterms to complete the cover. In practice this doesn't matter. Logic gates are inexpensive and except in rare cases, no one cares if your cover is minimal or not.

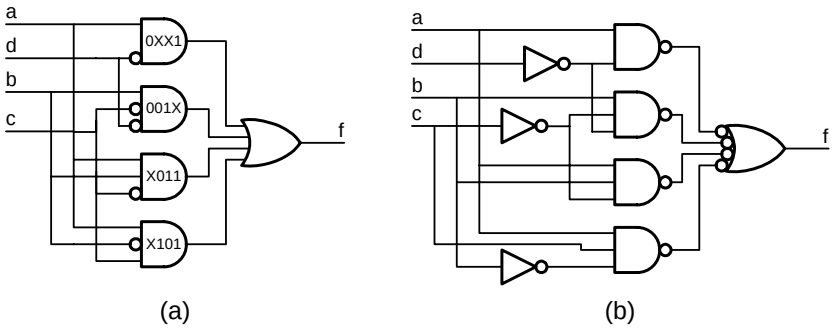


Figure 6.9: Logic circuit for the four-bit prime number function. (a) Logic circuit using AND and OR gates with arbitrary inversion bubbles on inputs. Each AND gate corresponds to a prime implicant in the cover of the function. (b) Logic circuit using CMOS NAND gates and inverters. NAND gates are used for both the AND and OR functions. Inverters complement inputs as required.

6.7 From a Cover to Gates

Once we have a minimum-cost cover of a logic function, the cover can be directly converted to gates by instantiating an AND gate for each implicant in the cover and using a single OR gate to sum the outputs of the AND gates. Such and AND-OR realization of the four-bit prime number function is shown in Figure 6.9(a).

With CMOS logic we are restricted to inverting gates, so we use NAND gates for both the AND and the OR functions as shown in Figure 6.9(b). Because CMOS gates have all inputs of the same polarity (all bubbles or no bubbles) we add inverters as needed to invert inputs. We could just have easily have designed the function using all NOR gates. NANDs are preferred, however, because they have lower logical effort for the same fan-in (see Section 5.3).

CMOS gates are also restricted in their fan-in (see Section 5.3). In typical cell libraries the maximum fan-in of a NAND or NOR gate is 4. If a larger fan-in is needed, a tree of gates (e.g., two NANDs into a NOR) is used to build a large AND or OR, adding inverters as needed to correct the polarity.

6.8 Incompletely Specified Functions (Dont' Cares)

Often our specification guarantees that a certain set of input states (or minterms) will never be used. Suppose, for example, we have been asked to design a one-digit decimal prime-number detecting circuit that need only accept inputs in the range from 0 to 9. That is for an input between 0 and 9 our circuit must output 1 if the number is a prime and 0 otherwise. However for inputs between 10 and 15 our circuit can output either 0 or 1 — the output is unspecified.

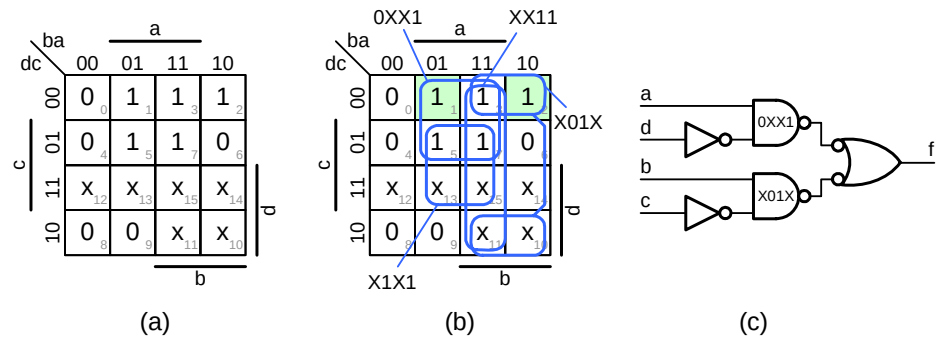


Figure 6.10: Design of a decimal prime-number circuit illustrates the use of don't cares in a K-map. (a) The K-map for the decimal prime-number circuit. Input states 10 through 15 labeled with X are don't care states. (b) The K-map with prime implicants shown. The circuit has four prime implicants 0XX1, X01X, XX11, and X1X1. The first two are essential as they are the only implicants that cover 0001 and 0010 respectively, the last two (XX11 and X1X1) are not essential and in fact is not needed. (c) A CMOS logic circuit derived from the K-map. The two NAND gates correspond to the two essential prime implicants.

We can simplify our logic by taking advantage of these don't care input states as shown in Figure 6.10. Figure 6.10(a) shows a K-map for the decimal prime number function. We place an X in each square of the K-map that corresponds to a don't care input state. In effect we are dividing the input states into three sets: f_1 - those input combinations for which the output must be 1, f_0 - those input combinations for which the output must be 0, and f_X - those input combinations where the output is not specified and may be either 0 or 1. In this case, f_1 is the set of five minterms labeled with 1 (1,2,3,5, and 7), f_0 containts the five minterms labeled 0 (0,4,6,8, and 9), and f_X contains the remaining minterms (10-15).

An implicant of an incompletely specified function is any product term that includes at least one minterm from f_1 and does not include any minterms in f_0. Thus we can expand our implicants by including minterms in f_X. Figure 6.10(b) shows the three prime implicants of the decimal prime number function. Note that implicant 001X of the original prime number function has been expanded to X01X to include two minterms from f_X. Also, two new prime implicants: X1X1 and XX11 have been added, each by combining two minterms from f_1 with two minterms from f_X. Note that products 11XX and 1X1X which are entirely in f_X are not implicants even though they contain no minterms of f_0. To be an implicant, a product must contain at least one minterm from f_1.

Using the notation of Equation 6.2 we can write a function with don't cares

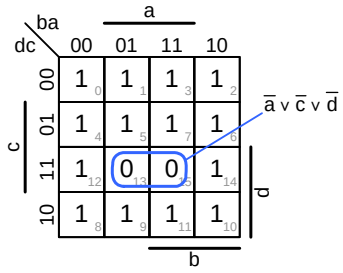


Figure 6.11: K-map for a function with two maxterms OR(0000) and OR(0010) that can be combined into a single sum, OR(00X0).

as:

$$f = \sum_{\text{in}} m(1,2,3,5,7) + D(10,11,12,13,14,15). \tag{6.4}$$

That is the function is the sum of five minterms plus six don't care terms.

We form a cover of a function with don't cares using the same procedure described in Section 6.6. In the example of Figure 6.10 there are two essential prime implicants: 0XX1 is the only prime implicant that includes 0001, and X01X is the only prime implicant that includes 0010. These two essential prime implicants cover all five of the minterms in f_1, so they form a cover of the function. The resulting CMOS gate circuit is shown in Figure 6.10(c).

6.9 Product-of-Sums Implementation

So far we have focused on the input states where the truth table is a 1 and have generated sum-of products logic circuits. By duality we can also realize product-of-sums logic circuits by focusing on the input states where the truth table is 0. With CMOS implementations we generally prefer the sum-of-products implementations because NAND gates have a lower logical effort than NOR gates with the same fan-in. However, there are some functions where the product-of-sums implementation is less expensive than the sum-of-products. Often both are generated and the better circuit selected.

A *maxterm* is a sum (OR) that includes every variable or its complement. Each zero in a truth table or K-map corresponds to a maxterm. For example, the logic function shown in the K-map of Figure 6.11 has two maxterms: $\overline{a} \vee \overline{b} \vee \overline{c} \vee \overline{d}$ and $\overline{a} \vee b \vee \overline{c} \vee \overline{d}$. For simplicity we refer to these as OR(0000) and OR(0010). Note that a maxterm corresponds to the complement of the input state in the K-map, so maxterm 0, OR(0000), corresponds to a 0 in square 15 of the K-map. We can combine adjacent 0s in the same way we combined adjacent 1s, so OR(0000) and OR(0010) can be combined into sum OR(00X0) = $\overline{a} \vee \overline{c} \vee \overline{d}$.

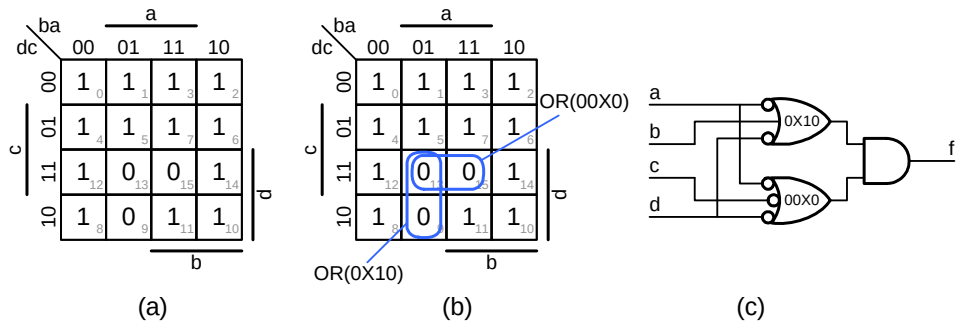


Figure 6.12: Product-of-sums synthesis. (a) K-map of a function with three maxterms. (b) Two prime sums. (c) Product of sums logic circuit.

The design process for a product-of-sums circuit is identical to sum-of-products design except that 0s in the K-maps are grouped instead of 1s. Figure 6.12 illustrates the process for a function with three maxterms. Figure 6.12(a) shows the K-map for the function. Two prime sums (OR terms that cannot be made any larger without including 1s) are identified in Figure 6.12(b): OR(00X0) and OR(0X10). Both of these sums are needed to cover all 0s in the K-map. Finally, Figure 6.12(c) shows the product-of-sums logic circuit that computes this function. The circuit consists of two OR gates, one for each of the prime sums, and an AND gate that combines the outputs of the OR gates so that the output of the function is 0 when the output of either OR gate is 0.

Once you have mastered sum-of-products design, the easiest way to generate a product-of-sums logic circuit is to find the sum-of-products circuit for the complement of the logic function (the function that results by swapping f_1 and f_0 leaving f_X unchanged.) Then, to complement the output of this circuit, apply Demorgan's theorem by changing all ANDs to ORs and complementing the inputs of the circuit.

For example, consider our decimal prime number function. The truth table for the complement of this function is shown in Figure 6.13(a). We identify three prime implicants of this function in Figure 6.13(b). A sum-of-products logic circuit that realizes the complement function of this K-map is shown in Figure 6.13(c). This circuit follows directly from the three prime implicants. Figure 6.13(d) shows the product-of-sums logic circuit that computes the decimal prime number function (the complement of the K-map in (a) and (b)). We derive this logic circuit by complementing the output of the circuit of Figure 6.13(c) and applying Demorgan's theorem to convert ANDs (ORs) to ORs (ANDs).

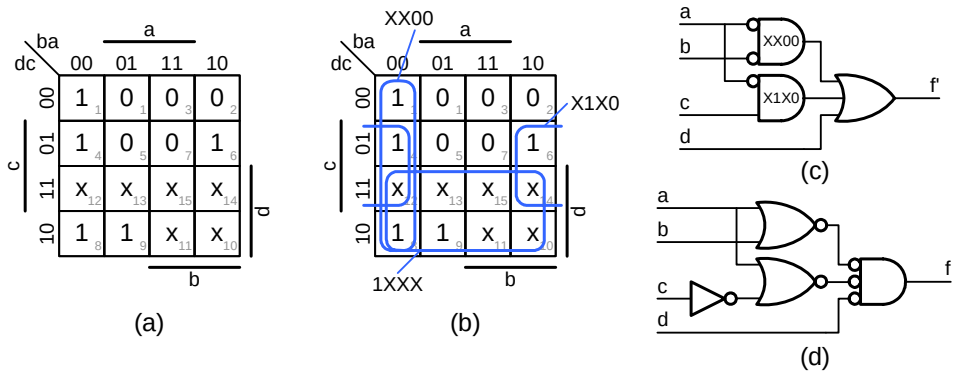


Figure 6.13: Implementation of the decimal prime number circuit in product-of-sums form using the complement method. (a) K-map for complement of the decimal prime number function (the decimal composite number function). (b) Prime implicants of this function (XX00, X1X0, and 1XXX). (c) Sum of products logic circuit that computes the complement decimal prime number function. (d) Logic circuit that generates the decimal prime number function. This is derived from (c) using Demorgan's theorem.

6.10 Hazards

On rare occasions we are concerned with whether or not our combinational circuits generate transient outputs in response to a single transition on a single input. Most of the time this is not an issue. For almost all combinational circuits we are concerned only that the steady-state output for a given input be correct — not how the output gets to its steady state. However, in certain applications of combinational circuits, e.g., in generating clocks or feeding an asynchronous circuit, it is critical that a single input transition produce at most one output transition.

Consider, for example, the two-input multiplexer circuit shown in Figure 6.14. This circuit sets the output f equal to input a when $c = 1$ and equal to input b when $c = 0$. The K-map for this circuit is shown in Figure 6.14(a). The K-map shows two essential prime implicants 1X1 ($a \wedge c$) and 01X ($b \wedge \overline{C}$) that together cover the function. A logic circuit that implements the function, using two AND gates for the two essential prime implicants, is shown in Figure 6.14(b). The number within each gate denotes the delay of the gate. The inverter on input c has a delay of 3, while the three other gates all have unit delay.

Figure 6.14(c) shows the transient response of this logic circuit when $a = b = 1$ and input c transitions from 1 to 0 at time 1. Three time units later, at time 4, the output of the inverter cN rises. In the meantime, the output of the upper AND gate d falls at time 2 causing output f to fall at time 3. At time 4,

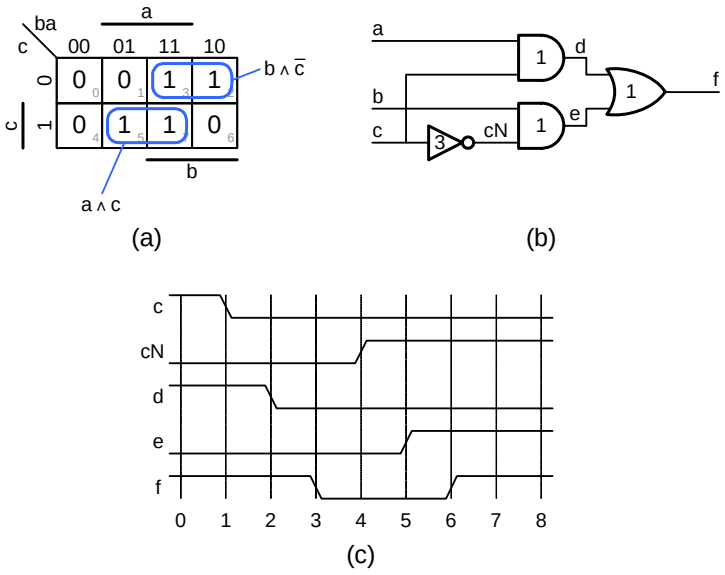


Figure 6.14: A two-input multiplexer circuit with a static 1 hazard. (a) K-map of the function showing two essential prime implicants. (b) Gate-level logic circuit for the multiplexer. The numbers denote the delay (in arbitrary units) of each gate. (c) Timing diagram showing the response of the logic circuit of (c) to a falling transition on input c when $a = b = 1$.

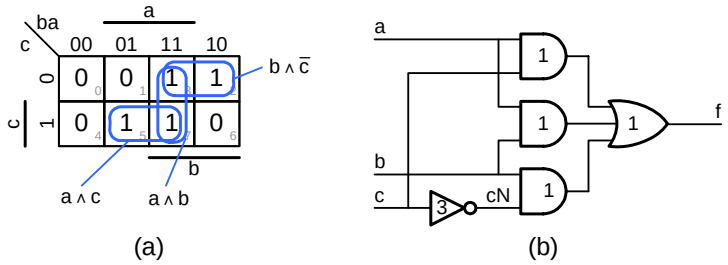


Figure 6.15: A two-input multiplexer circuit with no hazards. (a) K-map of the function showing three prime implicants. The implicant X11 is needed to cover the transition from 111 to 011 even though it is not essential. (b) Gate-level logic circuit for the hazard-free multiplexer.

the rising of signal cN causes signal e to rise, which in turn causes signal f to rise at time 6. Thus, a single transition on input c causes first a falling, then a rising transition on output f.

This transient 1-0-1 on output f is called a *static-1 hazard*. The output is normally expected to be a static 1, but has a transient hazard to 0. Similarly an output that undergoes a 0-1-0 response to a single input transition is said to have a *static-0 hazard*. More complex circuits, with more levels of logic, may also exhibit dynamic hazards. A dynamic-1 hazard is one in which an output goes through the states 0-1-0-1; starting at 0 and ending at 1 but with three transitions instead of 1. Similarly a dynamic-0 hazard is a three transition sequnce ending in the 0 state.

Intuitively the static-1 hazard of Figure 6.14 occurs because as the input transitions from 111 to 011, the gate associated with implicant 1X1 turns off before the gate associated with implicant 01X turns on. We can eliminate the hazard by covering the transition with an implicant of its own, X11, as shown in Figure 6.15. The third AND gate (the middle AND gate of Figure 6.15(b)), which corresponds to implicant X11, holds the output high while the other two gates switch. In general, we can make any circuit hazard free by adding redundant implicants to cover transitions in this manner.

6.11 Summary

After reading this chapter, you the reader now understand how to manually synthesize a combinational logic circuit. Given an English-language description of a circuit you can generate a gate-level implementation. You start by writing a *truth table* for the circuit to precisely define the behavior of the function. Writing the truth table in a *Karnaugh map* makes it easy to identify *implicants* of the function. Recall that *implicants* are products that include at least one minterm of f_1 and no minterms of f_0. They may or may not include minterms

of f_X. Once the implicants are identified, we generate a *cover* of the function by finding a minimal set of implicants that together contain every minterm in f_1. We start by identifying the *prime implicants*, that are included in no larger implicant, and the *essential prime implicants*, that cover a minterm of f_1 that is covered by no other prime implicant. We start our cover with the essential prime implicants of the function and then add prime implicants that include uncovered minterms of f_1 until all of f_1 is covered. From the cover it is straightforward to draw a CMOS logic circuit for the function. Each implicant in the cover becomes a NAND gate, their outputs are combined by a NAND gate (which performs the OR function), and inverters are added to the inputs as needed.

While it is useful to understand this process for manual logic synthesis, you will almost never use this procedure in practice. Modern logic design is almost always done using automatic logic synthesis in which a CAD program takes a high-level description of a logic function and automatically generates the logic circuit. Automatic synthesis programs relieve the logic designer from the drudgery of crunching K-maps, enabling her to work at a higher level and be more productive. Also, most automatic synthesis programs produce logic circuits that are better than the ones a typical designer could easily generate manually. The synthesis program considers multi-level circuits, considers implementations that make use of special cells in the library, and can try thousands of combinations before picking the best one. Its best to let the CAD programs do what they are good at — finding the optimal CMOS circuit to implement a given function — and have the designer focus on what humans are good at — coming up with a clever high-level organization for the system.

6.12 Bibliographic Notes

Not mentioning multiple output functions.

6.13 Exercises

6–1 *Combinational circuits.* Which of the circuits in Figure 6.16 are combinational? Each of the boxes is itself a combinational circuit.

6–2 *Fibonacci circuit.* Design a four-bit Fibonacci circuit. This circuit outputs a 1 iff its input is a Fibonacci number (i.e., 0,1,2,3,5,8, or 13). Go through the steps of:
(a) Write a truth table for the function.
(b) Draw a Karnaugh-map of the function.
(c) Identify the prime implicants of the function.
(d) Identify which of the prime implicants (if any) are essential.
(e) Find a cover of the function.
(f) Draw a CMOS gate circuit for the function.

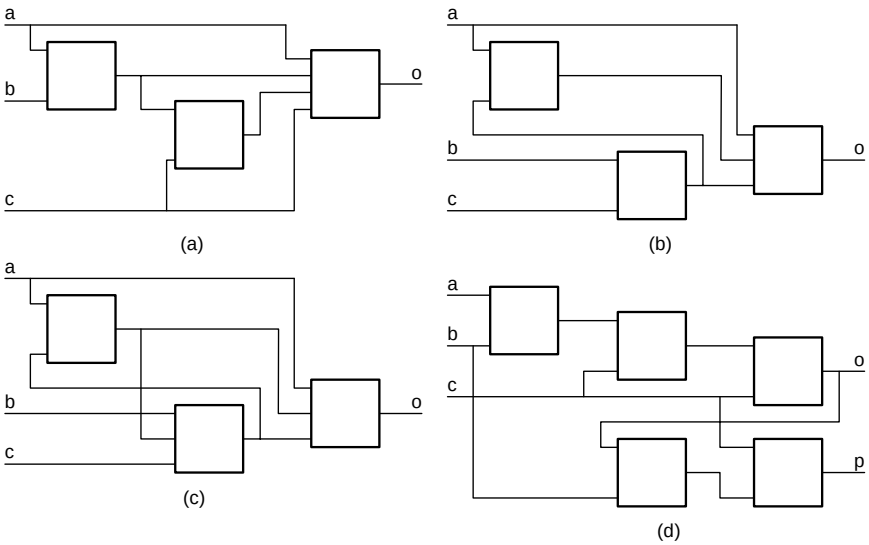


Figure 6.16: Circuits for Exercise 6–1. Each box is itself a combinational circuit.

6–3 *Decimal Fibonacci circuit.* Repeat Exercise 6–2, but for a decimal Fibonacci circuit. This circuit only need produce an output for inputs in the range of 0-9. The output is a don't care for the other six input states.

6–4 *Multiple-of three circuit.* Design a four-input multiple of three circuit. That is a circuit whose output is true if the input is 3,6,9,12, or 15.

6–5 *Combinational design.* Design a minimal CMOS circuit that implements the function $f = \sum m(3, 4, 5, 7, 9, 13, 14, 15)$.

6–6 *Five-input prime number circuit.* Design a five-input prime number circuit. The output is true if the input is a prime number between 1 and 31.

6–7 *Six-input prime number circuit.* Design a six-input prime number circuit. This circuit must also recognize the primes between 32 and 63 (neither of which is prime).

6–8 *Seven-segment decoder.* A seven segment decoder is a combinational circuit with a four-bit input a and a seven bit output q. Each bit of q corresponds to one of the seven segments of a display according to the following pattern:

```
 6666
1    5
```

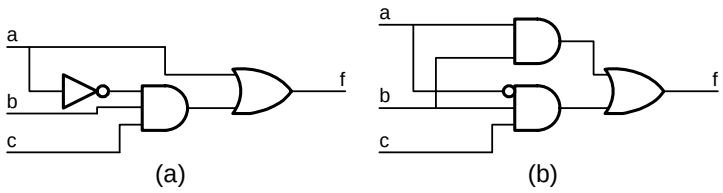


Figure 6.17: Circuits for Exercise 6–9.

```
1     5
 0000
2     4
2     4
 3333
```

That is, bit 0 (the LSB) of q controls the middle segment, bit 1 the upper left segment, and so on, with bit 6 (the MSB) controlling the top segment. Seven-segment decoders are described in more detail in Section 7.3. A full decoder decodes all 16 input combinations - approximating the letters A-F for combinations 10-15. A decimal decoder decodes only combinations 0-9, the remainder are don't cares.
(a)-(g) Design a sum-of-products circuit for one segment of the full decoder (for (a) do segment 0, for (b) do segment 1, and so on...).
(h)-(n) Design a product-of-sums circuit for one segment of the full decoder. (for (h) do segment 0, for (i) do segment 1, and so on...).
(o)-(u) Design a sum-of-products circuit for one segment of a decimal seven-segment decoder. (for (o) do segment 0, for (p) do segment 1, and so on...).
(v)-(z),(aa) Design a product-of-sums circuit for one segment of a decimal seven-segment decoder. (for (v) do segment 0, for (w) do segment 1, and so on...).
(ab) Design a sum-of-products circuit for the full decoder that generates the outputs for both segments 0 and 1. Share logic between the two outputs where possible.

6–9 *Hazards.* (a) Fix the hazard that may occur in Figure 6.17(a).
(b) Fix the hazard that may occur in Figure 6.17(b).

6–10 *Karnaugh maps.* A *half adder* is a circuit which takes in 1-bit binary numbers a and b and outputs a sum s and a carry out co. The concatenation of co and s `co,s` is the two-bit value that results from adding a and b (e.g., if a=1 and b=1, s=0 and co =1.)

A *full adder* is a circuit which takes in 1-bit binary numbers a, b, and ci (carry in), and outputs s and co. The concatenation of co and s {`co,s`}

is the two-bit value that results from adding a, b and ci (e.g., if a=1, b=0, and ci =1 then s=0 and co =1.)

Half and full adders are described in more detail in Chapter 10.

(a) Write out truth tables for the s and co outputs of a half adder.

(b) Draw Karnaugh maps for the s and co outputs of the half adder.

(c) Circle the prime implicants and write out the logic equations for the s and co outputs of the half adder

(d) Write out the truth tables for the s and co outputs for the full adder.

(e) Draw Karnaugh maps for the s and co outputs of the full adder

(f) Circle the prime implicants and write out the logic equations for the s and co outputs of the full adder

(g) How would the use of an XOR gate help in the half adder? In the full adder?

Chapter 7

Verilog Descriptions of Combinational Logic

In Chapter 6 we saw how to manually synthesize combinational logic circuits from a specification. In this chapter we show how to describe combinational circuits in the Verilog hardware description language, building on our discussion of Boolean expressions in Verilog (Section 3.6). Once the function is described in Verilog, it can be automatically synthesized, eliminating the need for manual synthesis.

Because all optimization is done by the synthsizer, the main goal in writing synthesizable Verilog is to make it easily readable and maintainable. For this reason, descriptions that are close to the function of a module (e.g., a truth-table specified with a `case` or `casex` statement) are preferable to those that are close to the implementation (e.g., equations using an assign statement, or a structural description using gates). Descriptions that specify just the function tend to be easier to read and maintain than those that reflect a manual implementation of the function.

To verify that a Verilog module is correct, we write a *test bench*. A test bench is a piece of Verilog code that is used during simulation to instantiate the module to be tested, generate input stimulus, and check the module's outputs. While modules must be coded in a strict synthesizable subset of Verilog, test benches, which are not synthesized, can use the full Verilog language, including looping constructs. In a typical modern digital design project at least as much effort goes into *design verification* (writing test benches) as goes into doing the design itself.

7.1 The Prime Number Circuit in Verilog

In describing combinational logic using Verilog we restrict our use of the language to constructs that can easily be synthesized into logic circuits. Specifically we restrict combinational circuits to be described using only `assign`, `case`, or

```
module <module_Name>(<port names>) ;
  <port declarations> ;
  <internal signal, wire and reg, declarations> ;
  <module body> ;
endmodule
```

Figure 7.1: A Verilog module declares a module, that is a block with inputs and outputs. It consists of a module declaration, input and output signal declarations, internal signal declarations, and a module body. The logic of the module is implemented in the body.

`casex` statements or by the structural composition of other combinational modules.[1]

In this section we shall look at four ways of implementing the prime number circuit we introduced in Chapter 6 as combinational verilog.

7.1.1 A Verilog Module

Before diving into our four implementations of the prime number module lets quickly review the structure of a Verilog module. A module is a block of logic with specified input and output ports. Logic within the module computes the outputs based on the inputs — on just the current state of the inputs for a combinational module. After declaring a module, we can instantiate one or more copies, or instances, of the module within a higher level module.

The basic form of a Verilog module is shown in Figure 7.1 and a module that implements the four-bit prime number function using a Verilog `case` statement is shown in Figure 7.2. All modules start with the keyword `module` and end with the keyword `endmodule`. From the word `module` to the first semicolon is the module declaration consisting of the module name (e.g., `prime` in Figure 7.2) followed by a list of port names enclosed in parentheses. For example the ports of the prime module are named `in` and `isprime`.

After the module declaration comes input and output declarations. Each of these statements starts with the keyword `input` or `output`, an optional width specification, and a list of ports with the specified direction and width. For example, the line `input [3:0] in ;` specifies that port `in` is an input of width 4 with the most-significant bit (MSB) of `in` being bit `in[3]`. Note that we could have declared it as `input [0:3] in ;` to have the MSB of `in` be `in[0]`.

Next comes the internal signal declarations. Here signals that will be assigned within the module are declared. Note that this may include output signals. If a signal is used to connect modules or assigned to with an `assign`

[1] It is possible to describe combinational modules using `if` statements. However, we discourage this practice because it is too easy to generate a sequential circuit by excluding an `else` clause, or by forgetting to assign to *every* output variable in *every* branch of the if statement.

```
//------------------------------------------------------------------------
// prime
//   in      - 4 bit binary number
//   isprime - true if "in" is a prime number 1,2,3,5,7,11, or 13
//------------------------------------------------------------------------
module prime(in, isprime) ;
  input [3:0] in ; // 4-bit input
  output      isprime ; // true if input is prime
  reg         isprime ;

  always @(in) begin
    case(in)
      1,2,3,5,7,11,13: isprime = 1'b1 ;
      default:         isprime = 1'b0 ;
    endcase
  end
endmodule
```

Figure 7.2: Verilog description of the four-bit prime-number function using a case statement to directly encode the truth table.

statement it is declared as a `wire` (see Figures 7.3 and 7.6). If a signal is assigned to in a `case` or `casex` statement it is declared as a `reg`, as with `isPrime` in Figure 7.2. Don't let this syntax confuse you, declaring a signal as `reg` does not create a register. We are still building combinational logic. Signal declarations may include a width field if the signal is wider than a single bit.

The module body statements perform the logic that computes the module outputs. In the subset of Verilog we will use here, the module body consists of one or more of module instantiations, `assign` statements, `case` statements, and `casex` statements. Examples of each of these are in the four implementations of the prime number circuit below.

7.1.2 The Case Statement

As shown in Figure 7.2 a Verilog case statement allows us to directly specify the truth-table of a logic function. The case statement allows us to specify the output value of a logic function for each input combination. In this example, to save space, we specify the input states where the output is 1 and make the 0 state a default.

Case statements must be contained within an `always @` block. This syntax specifies that the block will be evaluated each time the arguments specified after `@` change state. In this case, the block is evaluated each time the four-bit input variable `in` changes state. The output variable `isprime` is declared as a `reg` in this module. This is because it is assigned a value within an `always @`

```
module prime ( in, isprime );
input  [3:0] in;
output isprime;
    wire n1, n2, n3, n4;
    OAI13 U1 ( .A1(n2), .B1(n1), .B2(in[2]), .B3(in[3]), .Y(isprime) );
    INV   U2 ( .A(in[1]), .Y(n1) );
    INV   U3 ( .A(in[3]), .Y(n3) );
    XOR2  U4 ( .A(in[2]), .B(in[1]), .Y(n4) );
    OAI12 U5 ( .A1(in[0]), .B1(n3), .B2(n4), .Y(n2) );
endmodule
```

Figure 7.3: Result of synthesizing the Verilog description of Figure 7.2 with the Synopsys design compiler using a typical standard cell library. A schematic of this synthesized circuit is shown in Figure 7.4.

block. There is no register associated with this variable. The circuit is strictly combinational.

Whenever an `always @` block is used to describe a combinational circuit, it is critical that all inputs be included in the argument list after the `@`. If an input is omitted, the block will not be evaluated when this input changes state and the result will be sequential, not combinational, logic. Omitting signals from the list also results in odd behavior that can be difficult to debug.

The result of synthesizing the Verilog description of Figure 7.2 with the Synopsys design compiler using a typical CMOS standard cell library is shown in Figure 7.3. The synthesizer has converted the *behavioral* Verilog description of Figure 7.2, that specifies what is to be done (i.e., a truth table), to a *structural* Verilog description, that specifies how to do it (i.e., five gates and the connections between them). The structural verilog instantiates five gates: two OR-AND-Invert gates (OAI), two inverters (INV), and one exclusive-OR gate (XOR). The four wires connecting the gates are declared as `n1` through `n4`. For each gate, the design compiler output instantiates the gate by declaring the type of the gate (e.g., `OAI13`), giving this instance a name (e.g., `U1`), and then specifying which signal is connected to each gate input and output (e.g., `.A1(n2)` implies that signal `n2` is connected to gate input `A1`).

Note that a module can be instantiated with either this explicit notation for connecting signals to inputs and outputs or with a positional notation. For example, if the ports were declared in the order shown, we could instantiate the XOR gate with the simpler syntax:

```
    XOR2  U4 (in[2], in[1], n4) ;
```

The two forms are equivalent. For complex modules, the explicit connection syntax avoids getting the order wrong. For simple modules, the positional syntax is more compact and easier to read.

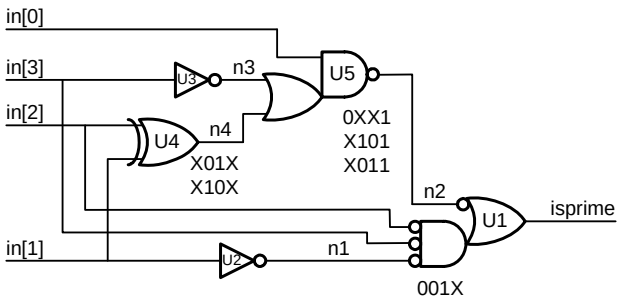


Figure 7.4: Schematic showing the circuit of Figure 7.3.

Figure 7.4 shows a schematic of the synthesized circuit to make it easier to see how the synthesizer has optimized the logic. Unlike the two-level synthesis method we employed in Chapter 6, the synthesizer has used four levels of logic (not counting inverters), and an exclusive-OR gate as well as ANDs and ORs. However, this circuit still implements the same four prime implicants (0XX1, 001X, X01X, and X10X). As shown in the Figure, the bottom part of gate U1 directly implements implicant 001X. Gate U5 implements the other three implicants - factoring `in[0]` out of the implicants so that this AND can be shared across all three. The top input to the OR of U5 (`n3`) ANDed with `in[0]` gives 0XX1. The output of the XOR gate gives products X01X and X10X which when ANDed with `in[0]` in U5 give the remaining two implicants X101 and X011.

This synthesis example illustrates the power of modern computer-aided design tools. A skilled designer would have to spend considerable effort to generate a circuit as compact as this one. Moreover, the synthesis tool can (via a constraint file) be asked to reoptimize this circuit for speed rather than area with minimum effort. With modern synthesis tools, the primary role of the logic designer has changed from one of optimization to one of specification. However, with this simplification of the low-level design task has come an increase in complexity at the high level as systems have gotten continuously larger.

7.1.3 The CaseX Statement

An alternative implementation of the prime-number function using the verilog `casex` statement to specify four prime implicants that cover the function is shown in Figure 7.5. This implementation is identical to the one in Figure 7.2 except that we use the `casex` statement in place of the `case` statement of Figure 7.2. The `casex` statement allows don't cares (Xs) in the cases. This allows us to put implicants, rather than just minterms, on the left side of each case. For example, the first case `4'b0xx1` corresponds to implicant 0XX1 and covers minterms 1, 3, 5, and 7.

```
module prime1(in, isprime) ;
  input [3:0] in ; // 4-bit input
  output      isprime ; // true if input is prime
  reg         isprime ;

  always @(in) begin
    casex(in)
      4'b0xx1: isprime = 1 ;
      4'b001x: isprime = 1 ;
      4'bx011: isprime = 1 ;
      4'bx101: isprime = 1 ;
      default: isprime = 0 ;
    endcase
  end
endmodule
```

Figure 7.5: Verilog description of the four-bit prime-number function using a casex statement to describe the implicants in a cover.

The `casex` statement is useful in describing combinational modules where one input often overrides the others. For example, when a disable input causes all outputs to go low regardless of the other inputs, or for a priority encoder (see Section 8.4). For the prime-number function, however, the implementation in Figure 7.2 is preferred because, even though its longer, it more clearly describes the function being implemented and is easier to maintain. There is no need to manually reduce the function to implicants. The synthesis tools do this.

7.1.4 The Assign Statement

Figure 7.6 shows a third Verilog description of the prime number circuit. This version uses an `assign` statement to describe the logic function using an equation. The word `assign` does not actually appear in this description because the `assign` statement has been combined with the `wire` statement declaring `isprime`. The `wire isprime = ...` statement is equivalent to

```
wire isprime ;
assign isprime = ...
```

As with the description using `casex`, there is little advantage to describing the prime number circuit with an equation. The truth table description is easier to write, easier to read, and easier to maintain. The synthesizer is perfectly capable of reducing the truth table to an equation. The designer doesn't need to do this.

```
module prime2(in, isprime) ;
  input [3:0] in ; // 4-bit input
  output      isprime ; // true if input is prime

  wire isprime = (in[0] & ~in[3]) |
                 (in[1] & ~in[2] & ~in[3]) |
                 (in[0] & ~in[1] & in[2]) |
                 (in[0] & in[1] & ~in[2]) ;
endmodule
```

Figure 7.6: Verilog description of the four-bit prime-number function using an assign statement. (In this case `assign` is combined with `wire`.

```
module prime3(in, isprime) ;
  input [3:0] in ; // 4-bit input
  output      isprime ; // true if input is prime

  and(a1,in[0],~in[3]) ;
  and(a2,in[1],~in[2],~in[3]) ;
  and(a3,in[0],~in[1],in[2]) ;
  and(a4,in[0],in[1],~in[2]) ;
  or(isprime,a1,a2,a3,a4) ;
endmodule
```

Figure 7.7: Verilog description of the four-bit prime-number function using explicit gates.

7.1.5 Structural Description

Our fourth and final description of the prime number function, shown in Figure 7.7, is a structural description that, much like the output of the synthesizer, describes the function by instantiating five gates and describing the connections between them. Unlike the synthesizer output (Figure 7.3), however, this description does not instantiate modules like OAI13. Instead it uses Verilog's built in `and` and `or` gate functions.

As with the previous two descriptions, we show this structural description of the prime number circuit to illustrate the range of the Verilog language. This is not the right way to describe the prime number function. As above, the designer should let the synthesizer do the synthesis and optimization.

```
module prime_dec(in, isprime) ;
  input [3:0] in ; // 4-bit input
  output      isprime ; // true if input is prime
  reg         isprime ;

  always @(in) begin
    casex(in)
      0: isprime = 0 ;
      1: isprime = 1 ;
      2: isprime = 1 ;
      3: isprime = 1 ;
      4: isprime = 0 ;
      5: isprime = 1 ;
      6: isprime = 0 ;
      7: isprime = 1 ;
      8: isprime = 0 ;
      9: isprime = 0 ;
      default: isprime = 1'bx ;
    endcase
  end
endmodule
```

Figure 7.8: Verilog description of the four-bit decimal prime-number function using a case statement with don't care on the default output.

```
module prime_dec ( in, isprime );
input  [3:0] in;
output isprime;
    wire n3, n4;
    NOR2  U3 ( .A(in[3]), .B(n3), .Y(isprime) );
    AOI12 U4 ( .A1(in[0]), .B1(in[1]), .B2(n4), .Y(n3) );
    INV   U5 ( .A(in[2]), .Y(n4) );
endmodule
```

Figure 7.9: Results of synthesizing the Verilog description of Figure 7.8 using Synopsys design compiler. A schematic diagram of this synthesized circuit is shown in Figure 7.10. The resulting circuit is considerably simpler than the fully specified circuit of Figure 7.4.

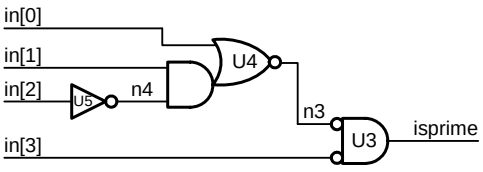


Figure 7.10: Schematic showing the circuit of Figure 7.9.

7.1.6 The Decimal Prime Number Function

Figure 7.8 illustrates how don't care input states can be specified in a Verilog description of a logic function. Here we again use the Verilog `casex` statement to specify a truth table with don't cares. In this case, however, we specify, using the `default` case, that input states 10 to 15 have a don't care output (`isprime = 1'bx`). Because we can have only a single default statement, and here we choose to use it to specify don't cares, we must explicitly include the five input states for which the output is zero.

The result of synthsizing the Verilog description of Figure 7.8 using Synopsys design compiler is shown in Figure 7.9, and a schematic diagram of the synthesized circuit is shown in Figure 7.10. With the don't cares specified, the logic is reduced to one 2-input gate, one 3-input gate, and one inverter as compared to one 4-input gate, one 3-input gate, an XOR, and two inverters for the fully-specified circuit.

7.2 A Testbench for the Prime Circuit

To test via simulation that the Verilog description of a module is correct, we write a Verilog *test bench* that exercises the module. A test bench is itself a

```
module test_prime ;
  reg [3:0] in ;
  wire isprime ;

  // instantiate module under test
  prime  p0(in, isprime) ;

  initial begin
    // apply all 16 possible input combinations to module
    in = 0 ;
    repeat (16) begin
      #100
      $display("in = %2d isprime = %1b",in,isprime) ;
      in = in+1 ;
    end
  end
endmodule
```

Figure 7.11: Verilog test bench for prime number module

Verilog module. However it is a module that is never synthesized to hardware. The test bench module is used only to facilitate testing of the module under test. The test bench module instantiates the module under test, generates the input signals to exercise the module, and checks the output signals of the module for correctness. The test bench is analogous to the instrumentation you would use on a lab bench to generate input signals and observe output signals from a circuit.

Figure 7.11 shows a simple test bench for the prime number circuit. The test bench is a Verilog module itself, but one with no inputs and outputs. Local variables are used as the inputs and outputs of the module under test, in this case `prime`. The test bench declares the input of the prime module, `in`,as a `reg` variable so it can be assigned values inside an `initial` block. The test bench instantiates an instance of module `prime` with inputs and outpus appropriately connected.

The actual test code for the test bench is contained within an `initial` block. An `initial` block is like an `always @` block except that instead of being executed every time a signal changes, it is executed exactly once at the beginning of the simulaiton. The initial block sets `in` to zero and then enters a loop. The `repeat (16)` statement repeats the loop body 16 times. During each iteration of the loop body, the simulator waits for 100 units of time `#100` for the output of the module to settle, displays the input and output, and then increments the input variable for the next loop iteration. After 16 iterations, the loop completes and the simulation terminates.

The test bench does not describe a piece of our design, but rather is just a

```
# in =  0 isprime = 0
# in =  1 isprime = 1
# in =  2 isprime = 1
# in =  3 isprime = 1
# in =  4 isprime = 0
# in =  5 isprime = 1
# in =  6 isprime = 0
# in =  7 isprime = 1
# in =  8 isprime = 0
# in =  9 isprime = 0
# in = 10 isprime = 0
# in = 11 isprime = 1
# in = 12 isprime = 0
# in = 13 isprime = 1
# in = 14 isprime = 0
# in = 15 isprime = 0
```

Figure 7.12: Output from test bench of Figure 7.11 on module described in Figure 7.2.

source of input stimulus and a monitor of output results. Because the test bench module doesn't have to be synthesized, it can use Verilog constructs that are not permitted in synthesizable designs. For example, the `initial` and `repeat` statements in Figure 7.11 are not allowed in synthesizable Verilog modules, but are quite useful in test benches. When writing Verilog, it is important to keep in mind whether one is writing synthesizable code or a test bench. Very different styles are used for each.

The output of a Verilog simulation of the test bench of Figure 7.11 and the prime number module of Figure 7.2 is shown in Figure 7.12. Each iteration of the loop, the `$display` statement in the test bench generates one line of output. By examining this output we can see that the prime number module is operating correctly.

Checking a Verilog module by manually examining its output works fine for small modules that need to be checked just once. However, for larger modules, or repeated testing[2] manual checking is at best tedious and at worst error prone. In such cases, the test bench must check results in addition to generating inputs.

One approach to a self-checking test bench is to instantiate two separate implementations of the module and compare their outputs as shown in Figure 7.13. (Another approach is to use an inverse function as shown below in Section 7.3.) In Figure 7.13, the test bench creates one instance of module `prime` (Figure 7.2)

[2]It is common practice to rerun a large test suite on an entire design on a periodic basis (e.g., every night). This *regression* testing catches many errors that result from the unintended consequences of making a change to one part of the design on a different, and often distant part.

```
module test_prime1 ;
  reg [3:0] in ;
  reg check ;  // set to 1 on mismatch
  wire isprime0, isprime1 ;

  // instantiate both implementations
  prime  p0(in, isprime0) ;
  prime1 p1(in, isprime1) ;

  initial begin
    in = 0 ; check = 0 ;
    repeat (16) begin
      #100
      if(isprime0 != isprime1) check = 1 ;
      in = in+1 ;
    end
    if(check != 1) $display("PASS") ; else $display("FAIL") ;
  end
endmodule
```

Figure 7.13: Go/no-go test bench that checks results using a second implementation of the prime-number module.

and one input of module `prime1` (Figure 7.5).[3] All 16 input patterns are then applied to both modules. If the outputs of the modules don't match for any pattern, the variable `check` is set equal to one. After all cases have been tried, a `PASS` or `FAIL` is indicated based on the value of `check`.

7.3 Example, A Seven-Segment Decoder

In this section we examine the design of a seven-segment decoder to introduce the concepts of constant definitions, signal concatenation, and checking with inverse functions.

A seven-segment display depicts a single decimal digit by illuminating a subset of seven light-emitting segments. The segments are arranged in the form of the numeral "8" as shown in the top part of Figure 7.14, numbered from 0 to 6 as shown. A seven segment decoder is a module that accepts a four-bit binary-coded input signal, `bin[3:0]`, and generates a seven-bit output signal, `segs[6:0]` that indicates which *segments* of a seven-segment display should be illuminated to display the number encoded by `bin`. For example, if the binary code for "4", 0100, is input to a seven-segment decoder, the output is 0110011 which indicates that segments 0, 1, 4, and 5 are illuminated to display a "4".

The first order of business in describing our seven-segment decoder is to define ten constants that each describe which segments are illuminated to display a particular numeral. Figure 7.14 shows the definition of ten constants `SS_0` through `SS_9` that serve this purpose. The constants are defined using the Verilog `` `define `` construct. Each `` `define `` statement maps a constant name to a constant value. For example, the constant named `SS_4` is defined to have the 7-bit string 0110011 as its value.

We define constants for two reasons. First, using constant names, rather than values, in our code makes our code more readable and easier to maintain. Second, defining a constant allows us to change all uses of the constant by changing a single value. For example, suppose we decide to drop the bottom segment on the "9". To do this, we would simply change the definition of `SS_9` to be 1110011 rather than 1111011 and this change would propagate automatically to every use of `SS_9`. Without the definition, we would have to manually edit every use of the constant — and would be likely to miss at least one.

The constant definitions give an example of the syntax used to describe numbers in Verilog. The general form of a number in Verilog is `<size><base><value>`. Here ¡size¿ is a decimal number that describes the width of the number in bits. In each constant definition, the size of the number is 7, specifying that each constant is seven bits wide. Note that `3'b0` and `7'b0` are different numbers, both have the value 0, but the first is 3 bits wide while the second is seven bits wide. The `<base>` portion of a number is `'b` for binary, `'d` for decimal, `'o` for octal (base 8), or `'h` for hexadecimal (base 16). In the constant definitions

[3] In this example, there is little advantage to comparing these two implementations since they are of roughly the same complexity. In other situations, however, there is often a very simple non-synthesizable description that can be used for comparison.

```
//-------------------------------------------------------------------------
// define segment codes
// seven bit code - one bit per segment, segment is illuminated when
// bit is high.  Bits 6543210 correspond to:
//
//         6666
//        1    5
//        1    5
//         0000
//        2    4
//        2    4
//         3333
//
//-------------------------------------------------------------------------
`define SS_0 7'b1111110
`define SS_1 7'b0110000
`define SS_2 7'b1101101
`define SS_3 7'b1111001
`define SS_4 7'b0110011
`define SS_5 7'b1011011
`define SS_6 7'b1011111
`define SS_7 7'b1110000
`define SS_8 7'b1111111
`define SS_9 7'b1111011
```

Figure 7.14: Defining the constants for the seven-segment decoder.

```
//-----------------------------------------------------------------------
// sseg - converts a 4-bit binary number to seven segment code
//
// bin  - 4-bit binary input
// segs - 7-bit output, defined above
//-----------------------------------------------------------------------
module sseg(bin, segs) ;
  input  [3:0] bin ;            // four-bit binary input
  output [6:0] segs ;           // seven segments
  reg    [6:0] segs ;

  always@(bin) begin
    case(bin)
      0: segs = `SS_0 ;
      1: segs = `SS_1 ;
      2: segs = `SS_2 ;
      3: segs = `SS_3 ;
      4: segs = `SS_4 ;
      5: segs = `SS_5 ;
      6: segs = `SS_6 ;
      7: segs = `SS_7 ;
      8: segs = `SS_8 ;
      9: segs = `SS_9 ;
      default: segs = 7'b0000000 ;
    endcase
  end
endmodule
```

Figure 7.15: A seven-segment decoder implemented with a case statement.

of Figure 7.14 all numbers are in binary. The inverse seven-segment module in Figure 7.16 uses hexadecimal numbers. Finally, the `<value>` portion of the number is the value in the specified base.

Now that we have the constants defined writing the verilog code for the seven-segment decoder module `sseg` is straightforward. As shown in Figure 7.15, we use a `case` statement to describe the truth table of the module, just as we did for the prime-number function in Section 7.1.2. The output values are defined using our defined constants. A defined constant is used by placing a backquote before its name. For example, the output when `bin` is 4 is `` `SS_4 `` which we have defined to be 0110011. Its much easier to read this code with the mnemonic constant names than if the right side of the case statement were all bit strings. When an input value is not in the range of 0-9, the `sseg` module outputs all zeros — a blank display.

To aid in testing our seven-segment decoder, we will also define an inverse

```
//-------------------------------------------------------------------------
// invsseg - converts seven segment code to binary - signals if valid
//
// segs  - seven segment code in
// bin   - binary code out
// valid - true if input is a valid seven segment code
//
//      segs = legal code (0-9) ==> valid = 1, bin = binary
//      segs = zero ==> valid = 0, bin = 0
//      segs = any other code ==> valid = 0, bin = 1
//-------------------------------------------------------------------------
module invsseg(segs, bin, valid) ;
  input  [6:0] segs ;           // seven segment code in
  output [3:0] bin ;            // four-bit binary output
  output       valid ;          // true if input code is valid
  reg    [3:0] bin ;
  reg          valid ;

  always@(segs) begin
    case(segs)
      `SS_0: {valid,bin} = 5'h10 ;
      `SS_1: {valid,bin} = 5'h11 ;
      `SS_2: {valid,bin} = 5'h12 ;
      `SS_3: {valid,bin} = 5'h13 ;
      `SS_4: {valid,bin} = 5'h14 ;
      `SS_5: {valid,bin} = 5'h15 ;
      `SS_6: {valid,bin} = 5'h16 ;
      `SS_7: {valid,bin} = 5'h17 ;
      `SS_8: {valid,bin} = 5'h18 ;
      `SS_9: {valid,bin} = 5'h19 ;
      0:     {valid,bin} = 5'h00 ;
      default: {valid,bin} = 5'h01 ;
    endcase
  end
endmodule
```

Figure 7.16: A Verilog description on an *inverse* seven-segment decoder, used to check the output of the seven-segment decoder.

segmen-segment decoder module as shown in Figure 7.16. Module `invsseg` accepts a seven-bit input string `segs`. If the input is one of the ten codes defined in Figure 7.14, the circuit outputs the corresponding binary code on output `bin` and a "1" on output `valid`. If the input is all zeros (corresponding to the output of the decoder when the input is out of range) the output is `valid = 0`, `bin = 0`. If the input is any other code, the output is `valid = 0`, `bin = 1`.

Again, our inverse seven-segment decoder uses a case statement to describe a truth-table. For each case, to assign both `valid` and `bin` in a single assignment, we concatenate the two signals and assign to the five-bit concatenated value. Placing two or more signals separated by commas, ",", in curly brackets, "{" and "}", concatenates those signals into a single signal with length equal to the sum of the lengths of its constituents. Thus, the expression {`valid, bin`} is a five-bit signal with `valid` as bit 4 and `bin` as bits 3-0. This five-bit composite signal can be used on either the left or right side of an expression. For example, the statement

```
{valid,bin} = 5'h14 ;
```

Is equivalent to

```
begin
  valid = 1'b1 ;
  bin   = 4'h4 ;
end
```

It assigns a logic 1 to `valid` (bit 4 of the composite signal), and a hex 4 to `bin` (the low four bits (3-0)of the composite signal). Assigning to a composite signal rather than separately assigning the two signals produces code that is more compact and more readable than assigning them separately.

Now that we have defined the seven-segment decoder module `sseg` and its inverse module `invsseg` we can write a test bench that uses the inverse module to check the functionality of the decoder itself. Figure 7.17 shows the testbench. The module instantiates the decoder and its inverse. The decoder accepts input `bin_in` and generates output `segs`. The inverse circuit accepts `segs` and generates outputs `valid` and `bin_out`.

After instantiating and connecting the modules, the test bench contains an `initial` block that loops through the 16 possible inputs. For inputs in range (between 0 and 9) it checks that `bin_in = bin_out` and that `valid` is 1. If these two conditions don't hold, an error is flagged. Similarly, for inputs out of range it checks that `bin_out` and `valid` are both zero. Note that we could have encoded the condition being checked as:

```
{valid, bin_out} != 0
```

Using an inverse module to check the functionality of a combinational module is a common technique in writing test benches. It is particularly useful in

```
//-----------------------------------------------------------------------
// test seven segment decoder - using inverse decoder for a check
//      note that both coders use the same set of defines so an
//      error in the defines will not be caught.
//-----------------------------------------------------------------------
module test_sseg ;
  reg  [3:0] bin_in ;           // binary code in
  wire [6:0] segs ;             // segment code
  wire [3:0] bin_out ;          // binary code out of inverse coder
  wire       valid ;            // valid out of inverse coder
  reg        error ;

  // instantiate decoder and checker
  sseg    ss(bin_in, segs) ;
  invsseg iss(segs, bin_out, valid) ;

  // walk through all 16 inputs
  initial begin
    bin_in = 0 ; error = 0 ;
    repeat (16) begin
      #100
      // uncomment the following line to display each case
      // $display("%h %b %h %b",bin_in,segs, bin_out, valid) ;
      if(bin_in < 10) begin
        if((bin_in != bin_out)||(valid != 1)) begin
          $display("ERROR: %h %b %h %b",bin_in,segs, bin_out, valid) ;
          error = 1 ;
        end
      end
      else begin
        if((bin_out != 0) || (valid != 0)) begin
          $display("ERROR: %h %b %h %b",bin_in,segs, bin_out, valid) ;
          error = 1 ;
        end
      end
      bin_in = bin_in+1 ;
    end
    if(error == 0) $display("TEST PASSED") ;
  end
endmodule
```

Figure 7.17: Test bench for the seven-segment decoder using the inverse function to test the output.

checking arithmetic circuits (see Chapter 10). For example, in writing a test bench for a square-root unit, we can square the result (a much simpler operation) and check that we get the original value.

The use of an inverse module in a test bench is also an example of the more general technique of using *checking modules*. Checking modules in test benches are like *assertions* in software. They are redundant logic that is inserted to check *invarients*, conditions that we know should always be true (e.g., two modules should not drive the bus at the same time). Because the checking modules are in the test bench, they cost us nothing. They are not included in the synthesized logic and consume zero chip area. However they are invaluable in detecting bugs during simulation.

7.4 Bibliographic Notes

7.5 Exercises

7–1 *Fibonacci circuit.* Write a Verilog description for a circuit that accepts a 4-bit input and outputs true if the input is a Fibonacci number (0,1,2,3,5,8, or 13). Describe why the approach you chose (case, casex, assign, structural) is the right approach.

7–2 *Decimal Fibonacci circuit.* Write a Verilog description for a circuit that accepts a 4-bit input that is guaranteed to be in the range of 0 to 9 and outputs true if the input is a Fibonacci number (0,1,2,3,5, or 8). The output is a don't care for input states 10 to 15. Describe why the approach you chose (case, casex, assign, structural) is the right approach.

7–3 *Logic synthesis.* Use a synthesis tool to synthesize the prime number circuit of Figure 7.2. Show the results of your synthesis.

7–4 *FPGA implementation.* Use an FPGA mapping tool (such as Xilinx Foundation) to map the seven-segment decoder of Figure 7.15 to an FPGA. Use the floorplanning tools to view the layout of the FPGA. How many CLBs did the synthesis use?

7–5 *Seven segment decoder.* Modify the seven segment decoder to output the characters 'A' through 'F' for input states 10 to 15 respectively.

7–6 *Test bench.* Modify the test bench of Figure 7.11 to check the output and indicate only pass or fail for the test.

7–7 *Test bench.* Write a Verilog test bench for the Fibonacci circuit of Exercise 7–2.

Chapter 8

Combinational Building Blocks

A relatively small number of modules: decoders, multiplexers, encoders, etc... are used repeatedly in digital designs. These building blocks are the idioms of modern digital design. Often, we design a module by composing a number of these building blocks to realize the desired function, rather than writing its truth table and directly synthesizing a logical implementation.

In the 1970s and 1980s most digital systems were built from small integrated circuits that each contained one of these building block functions. The popular 7400 series of TTL logic, for example, contained many multiplexers and decoders. During that period the art of digital design largely consisted of selecting the right building blocks from the TTL databook and assembling them into modules. Today, with most logic implemented as ASICs or FPGAs we are not constrained by what building blocks are available in the TTL databook. However, the basic building blocks are still quite useful elements from which to build a system.

8.1 Decoders

In general, a *decoder* converts symbols from one code to another. We have already seen an example of a binary to seven-segment decoder in Section 7.3. When used by itself, however, the term decoder means a binary to *one-hot* decoder. That converts a symbol from a binary code (each bit pattern represents a symbol) to a one-hot code (at most one bit can be high at a time and each bit represents a symbol). In Section 8.3 we will discuss *encoders* that reverse this process. That is, they are one-hot to binary decoders.

The schematic symbol for a $n \rightarrow m$ decoder is shown in Figure 8.1. Input signal a is an n-bit binary signal and output signal b is a m-bit ($m \leq 2^n$) one-hot signal. A truth table for a $3 \rightarrow 8$ decoder is shown in Table 8.1. If we think of both the input and output as binary numbers, if the input has value i, the

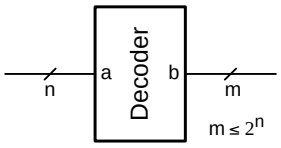


Figure 8.1: Schematic symbol for an $n \rightarrow m$ decoder.

bin	ohout
000	00000001
001	00000010
010	00000100
011	00001000
100	00010000
101	00100000
110	01000000
111	10000000

Table 8.1: Truth table for a $3 \rightarrow 8$ decoder. The decoder converts a 3-bit binary input, bin, to an eight-bit one-hot output, ohout.

output has value 2^i.

A verilog description of a $n \rightarrow m$ decoder is shown in Figure 8.2. This module introduces the use of Verilog *parameters*. The module uses parameters n and m to allow this single module type to be used to instantiate decoders of arbitrary input and output width. In the module description, the statement `parameter n=2 ;` declares that `n` (the input signal width) is a parameter with a default value of 2. Similarly `m` (the output signal width) is a parameter with a default value of 4.

If we instantiate the module as usual, the module will be created with the default values for all parameters. For example, the following code creates a $2 \rightarrow 4$ decoder since the default values are `n=2` and `m=4`.

```
Dec dec24(a, b) ;
```

We can override the default parameter values when we instantiate a module. The general form for such a parameterized module instantiation is:

```
<module name> #(<parameter list>) <instance name>(<port list>) ;
```

For example, to instantiate a $3 \rightarrow 8$ decoder, the appropriate Verilog code is:

```
Dec #(3,8) dec38(a, b) ;
```

Here the parameter list of `#(3,8)` sets `n=3` and `m=8` for this instance of the `Dec` module with instance name `dec38`. Similarly, a $4 \rightarrow 10$ decoder is created with:

```
//-----------------------------------------------------------------------
// n -> m  Decoder
// a - binary input   (n bits wide)
// b - one hot output (m bits wide)
//-----------------------------------------------------------------------
module Dec(a, b) ;
  parameter n=2 ;
  parameter m=4 ;

  input  [n-1:0] a ;
  output [m-1:0] b ;

  wire [m-1:0] b = 1<<a ;
endmodule
```

Figure 8.2: Verilog description of an n to m decoder.

```
Dec #(4,10) dec410(a, b) ;
```

Note that the output width m need not be equal to 2^n for input width n. In many cases (where not all input states occur) it is useful to instantiate decoders that have less than full width outputs. The module of Figure 8.2 uses the left shift operator "`<<`" to shift a 1 over to the position specified by binary input `a` to create one-hot output `b`.

A small decoder is constructed using an AND gate to generate each output as shown for a $2 \rightarrow 4$ decoder in Figure 8.3. Each input is complemented by an inverter. An AND gate for each output then selects the true or complement for each input to form the product corresponding to that output. Output $b1$ for example is generated by an AND gate with inputs $a0$ and $\overline{a1}$, so $b0 = a0 \wedge \overline{a1}$.

Large decoders can be constructed from small decoders as shown for a $6 \rightarrow 64$ decoder in Figure 8.4. The six-bit input $a[5:0]$ is divided into three two-bit fields and each is decoded by a 2:4 decoder generating three four-bit signals x, y, and z. In effect this *predecoding* stage converts the six-digit binary input into a three-digit quaternary (base 4) input. Each of the four-bit signals x, y, and z represents one quaternary number. Each bit of the one-hot representation of the quaternary number corresponds to a particular value — 0,1,2, or 3. The 64 outputs are generated by three input AND gates that combine one bit from each of the quaternary digits. The AND gate for output $b[i]$ selects the bits that correspond to the quaternary representation of the output number, i. For example output $b[27]$ (not shown) combines $x[1]$, $y[2]$, and $z[3]$ because $27_{10} = 123_4$.

Building a large decoder using predecoders as shown in Figure 8.4 reduces logical effort by factoring a large AND gate into two stages of smaller AND gates. For the 6:64 decoder of the figure, a 6-input AND gate would be re-

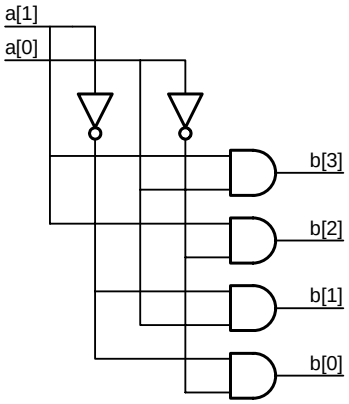


Figure 8.3: Schematic diagram of a $2 \rightarrow 4$ decoder. An array of inverters create the complements of the inputs and and array of AND gates generate the outputs.

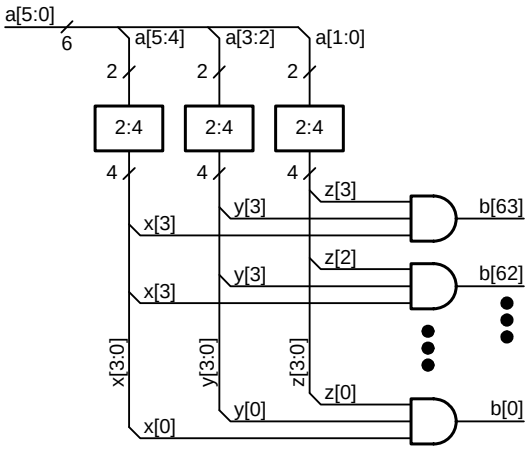


Figure 8.4: Schematic diagram of a $6 \rightarrow 64$ decoder. Three $2 \rightarrow 4$ decoders predecode each pair of input bits into three 4-bit one-hot signals, x, y, and z. An array of 64 3-input AND gates generates the outputs using these predecoded signals.

quired to realize the decoder in a single stage. The implementation with 2:4 predecoders replaces the 6-input AND gate with three 2-input AND gates (one in each predecoder) followed by a 3-input AND gate. Efficiency is also gained by sharing the 2-input AND gates across several outputs. In the figure, each 2-input AND gate (each predecoder output) is shared across 16 outputs (one for each combination of the other two quaternary digits).

The design of a large decoder is a compromise between wiring density and logical efficiency. A single level $n \rightarrow 2^n$ decoder requires $2n$ wiring tracks to run the inputs and their complements to all 2^n AND gates. Using 2:4 predecoders requires exactly the same number of wires since the four wires required to carry 2 binary bits and their complements are replaced by a four-wire one-hot quaternary digit. This is clearly a win because the fan-in of the output gates are halved without any increase in wire tracks. Moreover power is reduced since at most two of the bits of the quaternary signal change state (one up, one down) each time the input is changed, while all four wires of the true/complement binary signal may change state.

Going from 2:4 to 3:8 predecoders is more of a tradeoff. The number of wiring tracks increases by 33% (from $2n$ to $8n/3$) in exchange for reducing the fan-in of the output gates by 33% (from $n/2$ to $n/3$). This is still usually a good trade. However larger pre-decoders (e.g., 4:16) are rarely used because of the excessive number of wiring tracks required. An i-input predecoder requires $2^i n/i$ wiring tracks and n/i-input AND gates.

For very large decoders, the upper-digits of the pre-decoder are often distributed to eliminate the need to run all output wires across the entire AND-gate array. In Figure 8.4 (which is not really a *very* large decoder), for example, we could distribute the four AND gates of the predecoder that generates $x[3:0]$ so that the AND gate that generates $x[0]$ is next to the output AND gates that generate $b[15:0]$, the AND gate that generates $x[1]$ is next to the AND gates that generate $b[31:16]$ and so on. For wide input decoders, distributing decoders in this manner reduces wiring tracks. When the second most significant decoder is distributed, it is typically repeated for each output of the most significant decoder — negating some of the gate sharing advantage of pre-decoding to reduce wiring complexity.

A $n \rightarrow 2^n$ decoder can be used to build an arbitrary n-input logic function. The decoder generates all 2^n minterms of n inputs. An OR gate can be used to combine the minterms that are implicants of the function to be implemented. For example, Figure 8.5 shows how a 3-bit prime-number function can be realized with a 3:8 decoder. The decoder generates all 8 minterms $b[7:0]$. An OR gate combines the minterms $b[1], b[2], b[3], b[5]$, and $b[7]$ that are implicants of the function.

A Verilog module that describes a 3-bit prime number function using a decoder in this manner is shown in Figure 8.6. While it would be inefficient to actually implement the prime number function in this manner, it is a very compact and readable way to describe the function — very close to the notation $f = \sum m(1, 2, 3, 5, 7)$, and a good synthesizer will reduce this description to efficient logic.

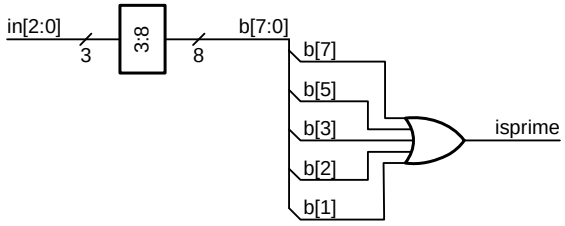


Figure 8.5: A 3-bit prime-number function implemented with a 3:8 decoder.

```
module Primed(in, isprime) ;
  input [2:0] in ;
  output      isprime ;
  wire  [7:0] b ;

  // compute the output as the OR of the required minterms
  wire        isprime = b[1] | b[2] | b[3] | b[5] | b[7] ;

  // instantiate a 3->8 decoder
  Dec #(3,8) d(in,b) ;
endmodule
```

Figure 8.6: A Verilog module that implements the 3-bit prime number function using a 3:8 decoder.

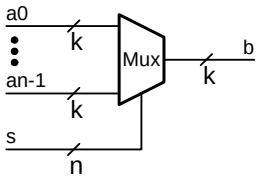


Figure 8.7: Schematic symbol for a k-bit $n \rightarrow 1$ multiplexer.

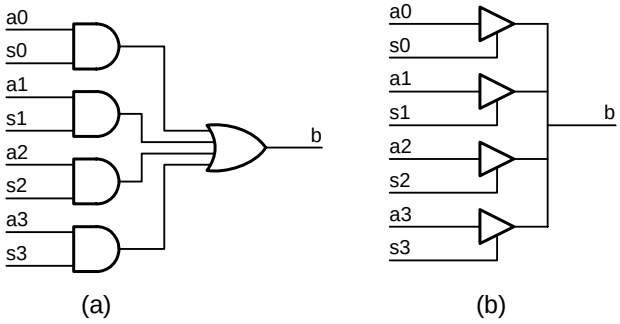


Figure 8.8: Schematic diagram of a $4 \rightarrow 1$ multiplexer. (a) Using AND and OR gates. (b) Using tri-state buffers.

8.2 Multiplexers

Figure 8.7 shows the schematic symbol for a k-bit $n \rightarrow 1$ multiplexer. This circuit accepts n k-bit wide input data signals $a_0, \ldots a_{n-1}$ and an n-bit one-hot select signal s. The circuit selects the input signal a_i that corresponds to the high-bit of s and outputs this value of a_i on the single k-bit wide output signal, b. In effect the multiplexer acts as a k-pole n-throw switch to select one of the n input data signals under control of the select signal.

Multiplexers are commonly used in digital systems as *data selectors*. For example a multiplexer on the input of an ALU selects the source of data to feed the ALU, and a multiplexer on the address lines of a RAM selects the data source to provide a memory address each cycle.

Figure 8.8 shows two implementations of a 1-bit 4:1 multiplexer. The implementation shown in Figure 8.8(a) uses AND and OR gates. Each data input a_i is ANDed with its corresponding select bit s_i and the outputs of the ANDs are ORed together. Because the select signal is one-hot, only the select bit s_i corresponding to the selected input is true, the output of this AND gate will be a_i, and the output of all other AND gates will be zero. Thus the output of the OR will be the selected input, a_i. An alternative design using tri-state buffers

```
// three input mux with one-hot select (arbitrary width)
module Mux3(a2, a1, a0, s, b) ;
  parameter k = 1 ;
  input [k-1:0] a0, a1, a2 ;  // inputs
  input [2:0]   s ; // one-hot select
  output[k-1:0] b ;
  wire [k-1:0] b = ({k{s[0]}} & a0) |
                   ({k{s[1]}} & a1) |
                   ({k{s[2]}} & a2) ;
endmodule
```

Figure 8.9: Verilog description of an arbitrary width $3 \rightarrow 1$ multiplexer.

is shown in Figure 8.8(b). A tri-state buffer is a logic gate for which its output is equal to its data input (left input) if its control input (bottom input) is high, and disconnected (open circuit) if the control input is low. The high bit of the select input s_i enables one of the tri-state buffers to transmit a_i to the output, all other tri-state buffers are disabled — effectively disconnected from the output. The advantage of the tri-state implementation is that it can be distributed, with each buffer placed near its corresponding data source, and only a single output line connecting the tri-state buffers. The AND/OR implementation on the other hand is more difficult to distribute because of the wiring required to connect to the final OR.

A Verilog description of an arbitrary-width 3:1 multiplexer is shown in Figure 8.9. This module takes 3 k-bit data inputs `a0`,`a1`, and `a2`, a 3-bit one-hot select input `s`, and generates a k-bit output `b`. The implementation uses an assign statement that matches the gate implementation of Figure 8.8(a) with two differences. First, since this is a three input multiplexer, there are three ANDs rather than four. Second, and more importantly, since this multiplexer is k-bits wide, each AND is k-bits wide — that is, k copies of a 2-input AND.

To feed each bit of the select signal, e.g., `s[0]`, into a k-bit wide AND gate, it must first be replicated to make a k-bit wide signal, each bit of which is `s[0]`. This is accomplished using the signal replication notation. In Verilog writing `{k{x}}` makes `k` copies of signal `x` concatenated end-to-end. Thus, in this module we make k copies of select bit 0, `s[0]`, by writing `{k{s[0]}}`.

Most standard-cell libraries provide multiplexers with one-hot select signals, and in most cases this is what we want — because our select signal is already in one-hot form. However in some cases, it is desirable to have a multiplexer with a binary select signal. This may be because our select signal is in binary rather than one-hot form, or because we have to transmit our select signal over a long distance (or through a narrow pin interface) and want to economize on wiring.

Figure 8.10(a) shows the symbol for a binary-select multiplexer. This circuit takes an $m = \lceil \log_2 n \rceil$ bit binary select signal, sb and selects one of the n input signals according to the binary value of sb — i.e., if $sb = i$ then a_i is selected.

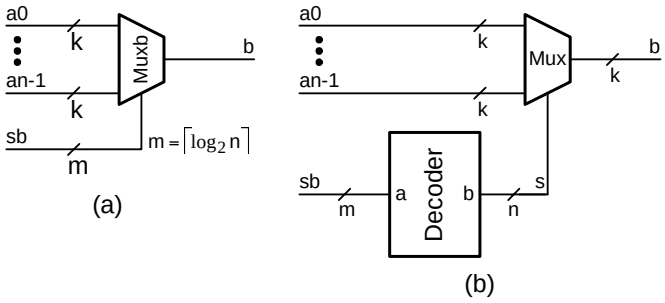


Figure 8.10: Binary select multiplexer. (a) A k-bit wide $n \rightarrow 1$ binary-select multiplexer selects input a_i where i is the value of m-bit wide binary select signal *sb*. (b) We can implement the binary-select multiplexer using a decoder and a normal (one-hot select) multiplexer.

We can implement a binary select multiplexer using two blocks we have already designed as shown in Figure 8.10(b). We use a $m \rightarrow n$ decoder to decode binary select signal *sb* into a one-hot select signal s and then use a normal multiplexer (with one-hot select) to select the desired input.

The Verilog description of a k-bit wide 3:1 binary-select multiplexer is shown in Figure 8.11. The description exactly matches that shown in Figure 8.10(b). A 2:3 decoder is instantiated to convert 2-bit binary select *sb* to 3-bit one-hot select s. The one-hot select is then used in a normal (one-hot select) 3:1 multiplexer to select the desired input.

A schematic diagram for a 4:1 binary-select multiplexer is shown in Figure 8.12. Each 2-input AND gate at the decoder output has been combined with the 2-input AND gate at the multiplexer input into a single 3-input AND gate. This fused implementation is more efficient than literally combining the two modules. However, a Verilog description that combines two modules, as in Figure 8.11, does not result in an inefficient implementation. A good synthesis program will generate a very efficient gate-level implementation from such a description. Again, the goal of the Verilog description is to be readable, maintainable, and synthesizable — optimization should be left to the synthesis tools.

A common design error is to use a binary-select multiplexer in a situation where the select signal is originally in one-hot form (e.g., the output of an arbiter that determines which bidder gets access to a shared resource). All too often a designer takes such a one-hot select signal, and encodes it into binary (see Section 8.3) only to decode it back to one-hot in the multiplexer. Such gratuitious encoding and decoding wastes chip area, burns power, and complicates the Verilog description. (Unfortunately most synthesis programs can't undo such a convoluted design.) Many designers over use binary-select multiplexers because they equate *multiplexer* with *binary-select multiplexer*. Don't do this. A basic multiplexer has a one-hot select input. If you have a one-hot select

```
// 3:1 multiplexer with binary select (arbitrary width)
module Muxb3(a2, a1, a0, sb, b) ;
  parameter k = 1 ;
  input [k-1:0] a0, a1, a2 ;  // inputs
  input [1:0]   sb ;          // binary select
  output[k-1:0] b ;
  wire  [3:0]   s ;

  Dec #(2,3) d(sb,s) ;              // decoder converts binary to one-hot
  Mux3 #(k)  m(a2, a1, a0, s, b) ;  // multiplexer selects input
endmodule
```

Figure 8.11: Verilog description of a binary-select 3:1 multiplexer using a decoder and a normal multiplexer.

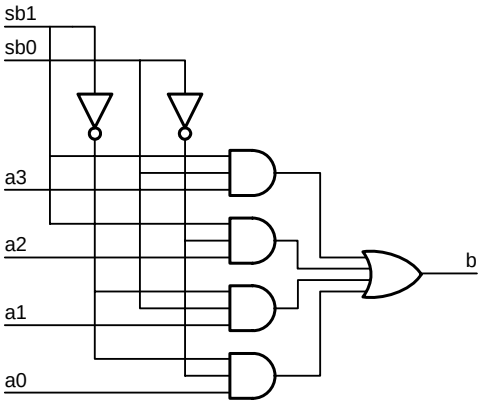


Figure 8.12: Schematic diagram of a $4 \rightarrow 1$ binary-select multiplexer.

```
module Mux6a(a5, a4, a3, a2, a1, a0, s, b) ;
  parameter k = 1 ;
  input [k-1:0] a0, a1, a2, a3, a4, a5 ;  // inputs
  input [5:0]   s ;                       // one-hot select
  output[k-1:0] b ;
  wire  [k-1:0] ba, bb ;
  wire  [k-1:0] b = ba | bb ;

  Mux3 #(k) ma(a2, a1, a0, s[2:0], ba) ;
  Mux3 #(k) mb(a5, a4, a3, s[5:3], bb) ;
endmodule
```

Figure 8.13: A six-input multiplexer created by OR-ing the output of two three-input multiplexers.

signal, leave it that way.

We can combine several small one-hot select multiplexers to build a larger multiplexer by just OR-ing their outputs together. The large one-hot select vector is then divided over the small multiplexers. The select signal for most of the multiplexers will be all zeros giving a zero output. Only the small multiplexer with the selected input gets a one-hot select signal, which enables the selected input to propagate all the way to the output. For example, Figure 8.13 shows how a 6:1 multiplexer can be constructed from two 3:1 multiplexers. Note for this or-ing of small multiplexers to work, each small multiplexer must output a zero when its select input is all zeros. This may not be the case for tri-state multiplexers (Figure 8.8(b)).

A large binary multiplexer is constructed as a tree of multiplexers as shown in Figure 8.14. The figure shows a 16:1 multiplexer constructed from four 4:1 multiplexers. A single 4:1 multiplexer at the output uses the most-significant bits (MSBs) of the select signal `s[3:2]` to select between the four 4-bit input groups. Within each input group the least-significant bits (LSBs) of the select signal `s[1:0]` select one of the four inputs in that group. For example, to select input `a11`, the select input is set to 1011, the binary code for 11. The MSBs are 10, which selects the `x2` input on the output multiplexer and hence the signal group from `a8` to `a11`. The LSBs are 11, which selects `a11` from within this group. One downside of this tree multiplexer structure, as compared to the one-hot approach of Figure 8.13, is that if the LSBs of the select signal switch, all four of the `x[3:0]` signals switch — dissipating energy, even though only one is needed.

An arbitrary n-input combinational logic function can be implemented using an $n \rightarrow 1$ multiplexer by placing the truth-table of the function on the inputs of the multiplexer. The binary select input of the multiplexer acts as the input to the logic function and selects the appropriate entry of the truth table. This is shown in Figure 8.15 for the 3-bit prime-number function. We can actually

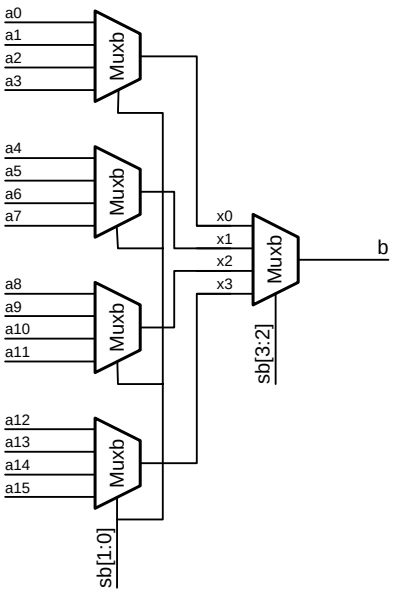


Figure 8.14: A large binary-select multiplexer is constructed from a tree of smaller binary-select multiplexers.

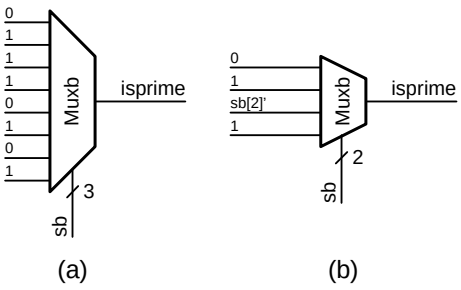


Figure 8.15: Combinational logic functions can be directly implemented with binary-select multiplexers. (a) The 3-bit prime-number function is implemented using an 8:1 binary select multiplexer. (b) The same function is realized with a 4:1 binary-select multiplexer.

```
module Primem(in, isprime) ;
  input [2:0] in ;
  output      isprime ;

  Muxb8 #(1) m(1, 0, 1, 0, 1, 1, 1, 0, in, isprime) ;
endmodule
```

Figure 8.16: Verilog description of the 3-bit prime-number function implemneted using an 8:1 binary-select multiplexer with its data inputs set to the prime-number truth table.

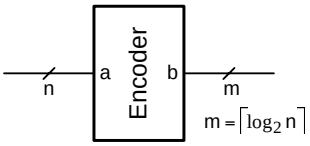


Figure 8.17: Schematic symbol for an $n \rightarrow m$ encoder that converts and n-bit one-hot signal to an $m = \lceil \log_2 n \rceil$ bit binary signal.

realize any n-input logic function using only an $n-1$ input multiplexer by factoring the function about the last input. This is shown for the 3-bit prime-number function in Figure 8.15(b). In effect we split the truth table into two pieces — for `sb[2]=0` and `sb[2]=1`. For each combination of the remaining inputs `sb[1:0]` we compare the two truth table halves. If the truth table is 0 (1) in both halves, we put a 0 (1) on the corresponding multiplexer input. However, if the function is 0 (1) when `sb[2]=0` and 1 (0) when `sb[2]=1`, we put `sb[2]` ($\overline{\texttt{sb[2]}}$) on the corresponding multiplexer input.

A Verilog module that implements the 3-bit prime-number function using a multiplexer is shown in Figure 8.16. The n-input multiplexer implementation is chosen because it is easier to write, read, and maintain. Again, we're leaving the low-level optimization to the synthesis tools.

8.3 Encoders

The schematic symbol for an encoder, a logic module that converts a one-hot input signal into a binary-encoded output signal, is shown in Figure 8.17. An encoder is the inverse function of a decoder. It accepts an n-bit one-hot signal and generates an $m = \lceil \log_2 n \rceil$ bit binary output signal. The encoder will only work properly if its input signal is one-hot.

An encoder is implemented with an OR gate for each output as shown in Figure 8.18 for a 4:2 encoder. Each OR gate has as inputs all of those one-hot input bits that correspond to binary numbers with that output set. In

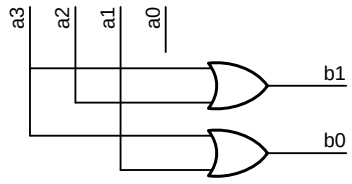


Figure 8.18: Schematic diagram of a 4 → 2 encoder.

```
// 4:2 encoder
module Enc42(a, b) ;
  input  [3:0] a ;
  output [1:0] b ;
  wire   [1:0] b = {a[3] | a[2], a[3] | a[1]} ;
endmodule
```

Figure 8.19: Verilog description of a 4:2 encoder.

Figure 8.18, for example, output $b1$ is true for binary codes 2 and 3. Hence it combines one-hot input bits $a2$ and $a3$.

The Verilog module for a 4:2 encoder is shown in Figure 8.19. This description follows the schematic shown in Figure 8.18. The two-bit output signal `b` is assigned in a single statement using signal concatenation to specify a separate logic equation for each bit of `b`.

A large encoder can be constructed from a tree of smaller encoders as shown in Figure 8.20. The figure shows a 16:4 encoder constructed from 4:2 encoders. To allow this composition, a summary output must be added to each 4:2 encoder. This output is true if any input of the encoder is true. The LSBs of the output are generated by OR-ing together the outputs of the 4:2 encoders connected directly to the inputs. The MSBs of the outputs are generated by an additional 4:2 encoder that encodes the summary outputs of the input encoders. The summary output of this final encoder (not shown) could be used as a summary output of the entire 16:4 encoder. Still larger encoders can be constructed by building additional levels to the tree.

To understand how the tree encoder works, consider the case where input `a[10]` is true and all other inputs are false. The third (from the bottom) input encoder has an input of 0100 and hence outputs 10 and sets its summary output true. All other input encoders have zero inputs and hence generate zero outputs. Since all other outputs are zero, the output of the third encoder pass through the two OR gates to directly generate the LSB of the output, `b[1:0] = 10`. The MSB encoder has an input of 0100 since only the third summary output is true. Hence it generates an output of `b[3:2] = 10`, giving an overall output of `b[3:0] = 1010` which is 10 in decimal.

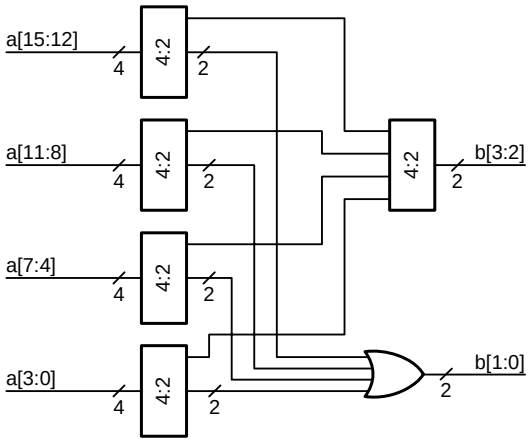


Figure 8.20: Large encoders can be constructed from a tree of smaller encoders. Each small encoder requires an additional output that indicates if any of its inputs are true.

Verilog code for the tree encoder is shown in Figure 8.21. In the code for the 4:2 encoder with summary output `Enc42a`, the statement that generates summary output `c` uses the Verilog *reduction* construct. The expression `|a` returns the result of OR-ing all of the bits of signal `a` together. (Similarly `&a` ANDs all of the bits of `a` together.)

8.4 Arbiters and Priority Encoders

Figure 8.22 shows the schematic symbol for an *arbiter* which is sometimes called a *find-first-one* (FF1) unit. This circuit accepts an arbitrary input signal and outputs a one-hot signal that has its sole one in the position of the least significant one in the input signal. For example, if the input to an 8-bit arbiter were 01011100, the output would be 00000100 since the least significant 1 in the input is in bit 2. In some applications we reverse the arbiter and look for the most significant one. For the remainder of this section, however, we'll focus on arbiters that look for the least-significant one in the input signal.

Arbiters are used in digital systems to arbitrate requests for shared resources. For example if n units share a bus that only one can use at a time, an n-input arbiter is used to determine which unit gets access to the bus during a given cycle.[1] Another use of an arbiter is in arithmetic circuits where to *normalize* numbers we need to find the position of the most significant one. In this

[1] In this application we would typically use an arbiter with rotating priority so that the resource is shared fairly. With a fixed priority arbiter, the unit connected to the least significant input gets an unfair advantage.

```
//-------------------------------------------------------------------------
// 4 to 2 encoder - with summary output
module Enc42a(a, b, c) ;
  input  [3:0] a ;
  output [1:0] b ;
  output c ;
  wire   [1:0] b = {a[3] | a[2], a[3] | a[1]} ;
  wire   c  = |a ;
endmodule
//-------------------------------------------------------------------------
// factored encoder
module Enc164(a, b) ;
  input [15:0] a ;
  output[3:0]  b ;
  wire [3:0] b ;
  wire [7:0] c ; // intermediate result of first stage
  wire [3:0] d ; // if any set in group of four

  // four LSB encoders each include 4-bits of the input
  Enc42a e0(a[3:0],  c[1:0],d[0]) ;
  Enc42a e1(a[7:4],  c[3:2],d[1]) ;
  Enc42a e2(a[11:8], c[5:4],d[2]) ;
  Enc42a e3(a[15:12],c[7:6],d[3]) ;

  // MSB encoder takes summaries and gives msb of output
  Enc42 e4(d[3:0], b[3:2]) ;

  // two OR gates combine output of LSB encoders
  assign b[1] = c[1]|c[3]|c[5]|c[7] ;
  assign b[0] = c[0]|c[2]|c[4]|c[6] ;
endmodule
//-------------------------------------------------------------------------
```

Figure 8.21: Verilog code for a 16:4 encoder built as a tree of 4:2 encoders with a summary output (Module `Enc42a`). The summary output is true if any input of the module is true.

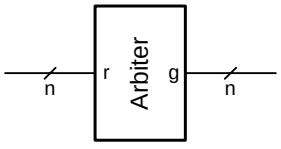


Figure 8.22: Schematic symbol for an arbiter.

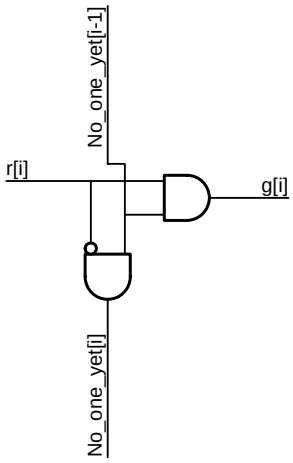


Figure 8.23: Logic diagram for one bit of an arbiter. The output is true only if no ones have been found so far, and the input is one. If a one has been previously found, or if the input to this stage is one, an output signal informs other stages that a one has been found.

application, they are called find-first-one units, because there is no arbitration going on, and they are reversed (compared to what we discuss here) to find the most-significant one.

An arbiter can be constructed as an *iterative circuit*. That is, we can design the logic for one bit of the arbiter and repeat (or iterate) it. Figure 8.23 shows the logic for one bit (bit i) of the arbiter. One AND gate generates the grant output for this bit `g[i]`. The grant is set high if the request is high `r[i]` and no one has been found so far, as signaled by a one on the top input. The second AND gate signals downstream bits if no one has been found by this stage or any previous stage.

We will see many examples of iterative circuits in our study of digital design. They are widely used, for example, in arithmetic circuits (Chapter 10).

To build a four-bit arbiter, for example, we would connect four copies of the bit cell of Figure 8.23 and connect the top input of the first cell to "1". The resulting circuit is shown in Figure 8.24(a). The vertical chain of AND gates scans over the request inputs until the first 1 is found and disables all outputs below this input. The output AND gates pass the first input 1 to the output and force zeros on all outputs below this first one.

The linear chain of AND gates in this circuit gives a delay that increases linearly with the number of inputs. In some applications this delay can be prohibitive. We can shorten the delay somewhat by flattening the logic as shown in Figure 8.24(b). This technique is often called *look ahead* since the gate that generates output `g3` is looking ahead at input `r0` rather than waiting for

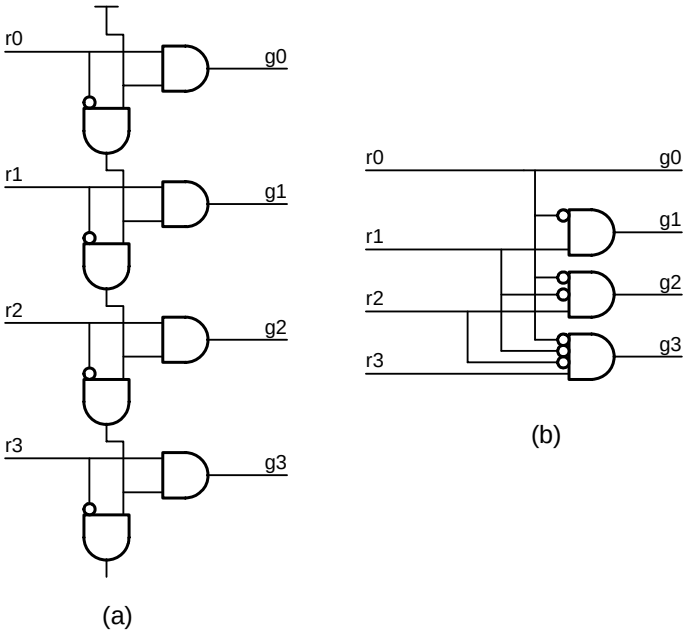


Figure 8.24: Two implementations of a four-bit arbiter. (a) using the bit-cell of Figure 8.23. (b) using *look-ahead*.

```
//------------------------------------------------------------------------
// arbiter (arbitrary width) - LSB is highest priority
module Arb(r, g) ;
  parameter n=8 ;
  input  [n-1:0] r ;
  output [n-1:0] g ;
  wire   [n-1:0] c = {(~r[n-2:0] & c[n-2:0]),1'b1} ;
  wire   [n-1:0] g = r & c ;
endmodule
//------------------------------------------------------------------------
// arbiter (arbitrary width) - MSB is highest priority
module RArb(r, g) ;
  parameter n=8 ;
  input  [n-1:0] r ;
  output [n-1:0] g ;
  wire   [n-1:0] c = {1'b1,(~r[n-1:1] & c[n-1:1])} ;
  wire   [n-1:0] g = r & c ;
endmodule
//------------------------------------------------------------------------
```

Figure 8.25: Verilog description of two arbitrary width fixed-priority arbiters. `Arb` finds the least-significant one and `RArb` finds the most-significant one.

the affect of `r0` to propagate through a chain of gates. We will look at more scalable approaches to look ahead for iterative circuits in Section 12.1.

Verilog descriptions of two arbiters are shown in Figure 8.25. One arbiter, `Arb`, finds the least-signficant one and the other, `RArb`, finds the most significant one. The implementation of `Arb` directly follows Figure 8.24(a). Signal `g` is generated by ANDing the 'no ones so far' signal `c` with the request input `r`. The 'no ones so far' signal, `c`, is generated by using a concatenation to set its LSB to 1 (always enabling output `g0`) and to set the remaining bits to `c[i] = r[i-1]&c[i-1]`. At first this definition looks circular because it appears that we are defining `c` in terms of itself. However, on closer examination we realize that each bit of `c` only depends on less signficant bits of `c`. Thus the definition is not circular.

One use of an arbiter is to build a *priority encoder* as shown in Figure 8.26. A priority encoder takes an n-bit input signal `a` and outputs an $m = \lceil \log_2 n \rceil$ bit binary signal `b` that indicates the position of the first one bit in `a`. The priority encoder operates in two steps as shown in Figure 8.26(a). First, an arbiter finds the first one bit in input `a` and outputs a one-hot signal, `g`, with only this single bit set. Then, one-hot signal `g` is converted to a binary signal `b` by an encoder. A Verilog description of the priority encoder is shown in Figure 8.27.

When input `a` is zero, the arbiter will output `g` = 0 which is not a one-hot signal. In this case the encoder, if it is constructed from OR gates as described

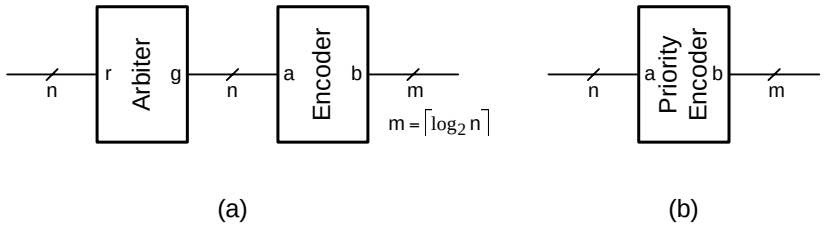


Figure 8.26: (a) A priority encoder realized by connecting an arbiter to an encoder. (b) Schematic symbol for a priority encoder.

```
// 8:3 priority encoder
module PriorityEncoder83(r, b) ;
  input  [7:0] r ;
  output [2:0] b ;
  wire   [7:0] g ;
  Arb #(8) a(r, g) ;
  Enc83    e(g, b) ;
endmodule
```

Figure 8.27: Verilog description of a priority encoder.

above (Section 8.3) will also output `b` = 0 which is often acceptable. In some applications, however, this all-zero state must be detected and a special code output to distinguish the case when the input is zero from the case where the first bit set is bit zero. This is easily accomplished by making the arbiter one bit wider than otherwise needed and applying a constant one signal to the last input bit.

8.5 Comparators

Figure 8.28 shows the schematic symbol for an *equality comparator*. This module accepts two n-bit binary inputs `a` and `b` and outputs a one-bit signal that indicates if `a` = `b` — i.e., that each bit of `a` is equal to the corresponding bit of `b`.

Figure 8.29 shows a logic digram for a four-bit equality comparator. An array of exclusive-NOR (XNOR) gates compares individual bits of the input signals. The output of each XNOR gate is high if its two inputs are equal, so signal eq_i is true if $a_i = b_i$. An AND gate combines the eq_i signals and outputs true only if all bits are equal. Alternately, We can design the equality comparator as an iterative circuit by linearly scanning the bits to determine that all are equal.

A verilog description of an equality comparator is shown in Figure 8.30.

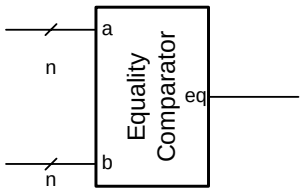


Figure 8.28: Schematic symbol for an equality comparator. The `eq` output is true if `a` = `b`.

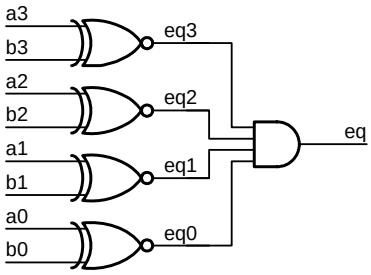


Figure 8.29: Logic diagram of a four-bit equality comparator. Exclusive-NOR (XNOR) gates are used to compare individual bits of inputs `a` and `b`. An AND gate combines the bit-by-bit comparisons and signals true if all bits are equal.

```
// 8:3 priority encoder
module EqComp(a, b, eq) ;
  parameter k=8 ;
  input  [k-1:0] a, b ;
  output eq ;
  wire   eq = (a == b) ;
endmodule
```

Figure 8.30: Verilog description of an equality comparator using the Verilog "==" operator .

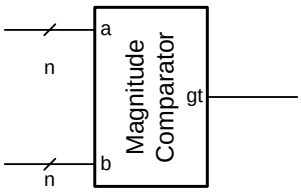


Figure 8.31: Schematic symbol for an magnitude comparator. The `gt` output is true if `a` > `b`.

Here we use the Verilog "`==`" operator to directly generate equality output `eq`. An alternate implementation would use the XNOR operator "`~^`" to perform a bitwise comparison as:

```
wire eq = &(a ~^ b) ;
```

Here we use the AND reduction operator to combine the bits of the XNOR without need to declare an intermediate signal.

A *magnitude comparator* is a module that compares the relative magnitude of two binary numbers. Strictly speaking this is an arithmetic circuit, since it treats its inputs as numbers, and thus we should defer its treatment until Chapter 10, we present it here because it is an excellent example of an iterative circuit.

Figure **??** shows a schematic symbol for a magnitude comparator. The single bit output `gt` is true if n-bit input `a` is greater than n-bit input `b`. One binary signal is greater than another if it has a 1 in the most significant position in which the two numbers are not equal.

We can structure two different iterative circuits for the magnitude comparator as shown in Figure 8.32. In Figure 8.32(a) we scan from LSB to MSB to find the most significant bit in which the two numbers disagree. In this design we propagate a signal `gtb` (greater-than below). Signal $\texttt{gtb}_i$ if set indicates that from the LSB through bit $i-1$ signal `a` is greater than `b` — i.e., `gtb[i]` implies `a[i-1:0] > b[i-1:0]`. Bit $\texttt{gtb}_{i+1}$ is set if `a[i] > b[i]` or if `a[i] == b[i]` and `a[i-1:0] > b[i-1:0]`. The `gtb[n]` signal out of the most significant bit gives the required answer since it indicates that `a[n-1:0] > b[n-1:0]`. A Verilog description of this LSB-first magnitude comparator is shown in Figure 8.33.

An alternate iterative implementation of a magnitude comparator that operates MSB first is shown in Figure 8.32(b). Here we have to propagate two signals between each bit position. Signal `gta[i]` (greater than above) indicates that `a > b` just bits more significant than the current bit, i.e., `a[n-1:i+1] > b[n-1:i+1]`. Similarly `eqa[i]` (equal above) indicates that `a[n-1:i+1] == b[n-1:i+1]`. These two signals scan the bits from MSB to LSB. As soon as a difference is found, we know the answer. If the first difference is a bit where `a > b` we set `gta[i-i]` and clear `eqa[i-i]` out of this bit position, and these values propagate all the way to the output. On the other hand, if `b > a` in the

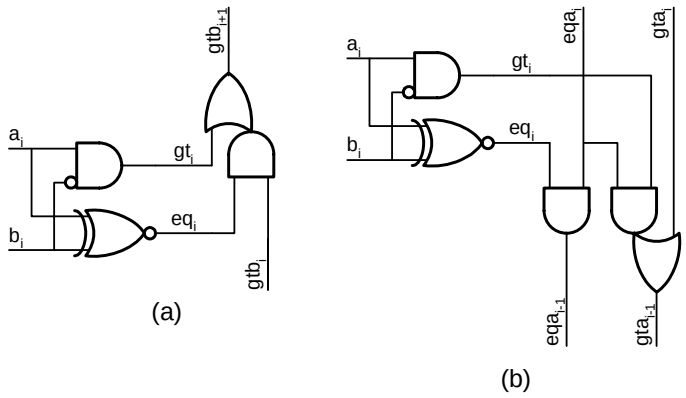


Figure 8.32: Two iterative implementations of the magnitude comparator. (a) LSB first, a greater-than below `gtb` signal is propagated upward. upward. (b) MSB first, two signals: greater-than above `gta` and equal above `eqa` are propagated downward.

```
module MagComp(a, b, gt) ;
  parameter k=8 ;
  input  [k-1:0] a, b ;
  output gt ;
  wire   [k-1:0] eqi = a ~^ b ;
  wire   [k-1:0] gti = a & ~b ;
  wire   [k:0] gtb = {((eqi[k-1:0]&gtb[k-1:0])|gti[k-1:0]),1'b0} ;
  wire   gt = gtb[k] ;
endmodule
```

Figure 8.33: Verilog description of an LSB-first magnitude comparator.

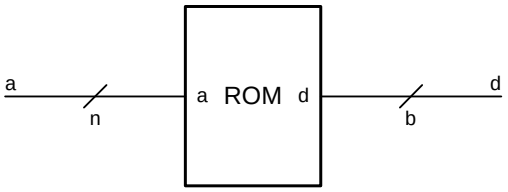


Figure 8.34: Schematic symbol for a ROM. The n-bit address a selects a location in a table. The value stored in that location is output on the b-bit data output d.

first bit that differs, we clear `eqa[i-1]` but leave `gta[i-1]` low. These signals also propagate all the way to the output. The output is the signal gta[-1].

8.6 Read-Only Memories (ROMs)

A *read-only memory* or ROM is a module that implements a look-up table. It accepts an address as input and outputs the value stored in the table at that address. The ROM is *read-only* because the values stored in the table are predetermined - hard-wired at the time the ROM is manufactured and cannot be changed. Later we will examine *read-write memories* where the table entries can be changed.

The schematic symbol for a ROM is shown in Figure 8.34. For an N-word $\times$ b-bit ROM, an $n = \lceil \log_2 N \rceil$ bit address signal a selects a word of the table. The b-bit value stored in that word is output on data output d.

A ROM can implement an arbitrary logic function by storing the truth table of that function in the ROM. For example, we can implement a seven-segment decoder with a 10-word $\times$ 7-bit ROM. The value 1111110, the segment pattern for 0, is placed in the first location (location 0), the value 0110000, the segment pattern for 1, is placed in the second location (location 1) and so on.

A simple implementation of a ROM using a decoder and tri-state buffers is shown in Figure 8.35. A $n \rightarrow N$ decoder decodes the n-bit binary address a into an N-bit one-hot word select signal, w. Each bit of this word select signal is connected to a tri-state gate. When an address $a = i$ is applied to the ROM, word select signal w_i goes high and enables the corresponding tri-state buffer to drive table entry d_i onto the output.

For large ROMs, the one-dimensional ROM structure of Figure 8.35 becomes unweildy and inefficient. The decoder becomes very large — requring N AND gates. Above a certain size, it is more efficient to construct a ROM as a two-dimensional array of cells as shown in Figure 8.36. Here the eight-bit address $a_{7:0}$ is divided into a six-bit *row address* $a_{7:2}$ and a two-bit *column address* $a_{1:0}$. The row address is input to a decoder and use to select a row via a 64-bit one hot select signal w. The column address is input to a binary-select multiplexer

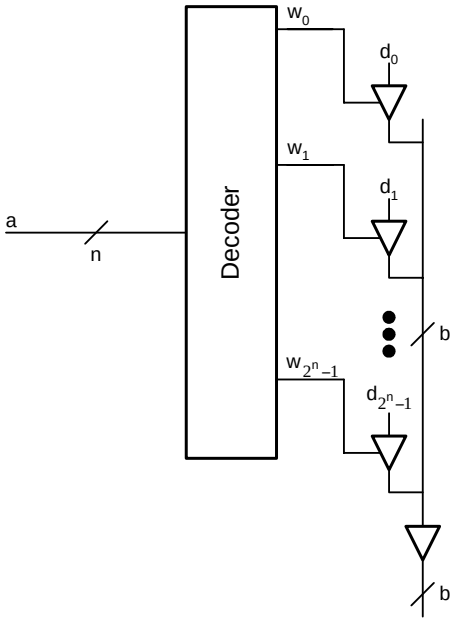


Figure 8.35: A ROM can be implemented with a decoder and a set of tri-state gates with constants connected to their inputs. The address is decoded to select one of the tri-state gates. That gate drives its corresponding value onto the output.

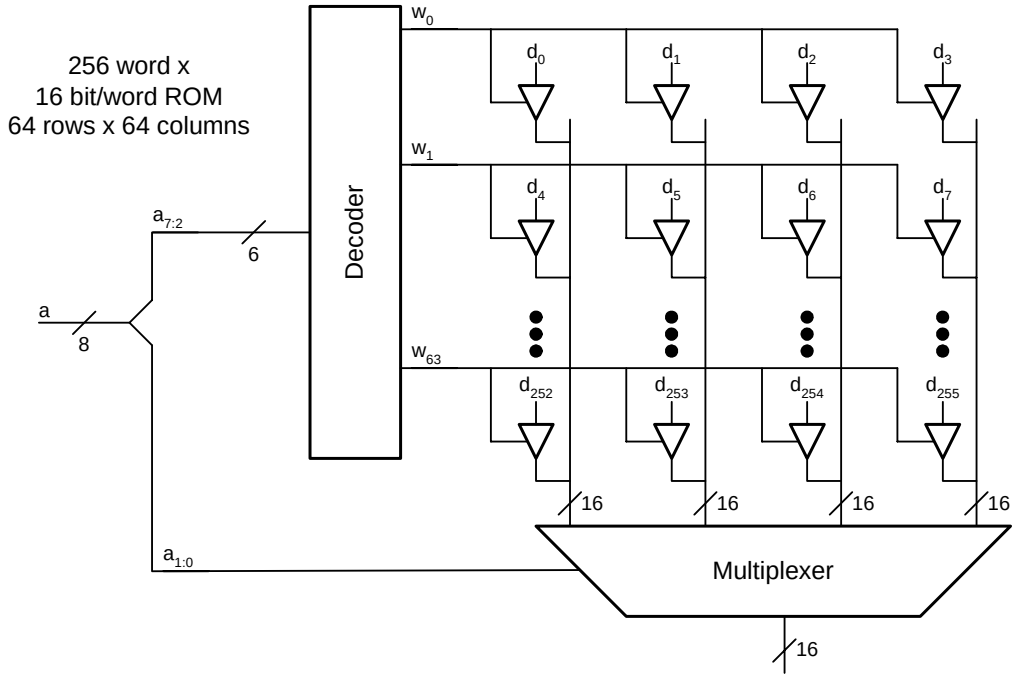


Figure 8.36: A ROM can be implemented more efficiently as a two-dimensional structure. A decoder selects a *row* of words. A multiplexer selects the desired *column* from the selected row.

that selects the appropriate word from the row. For example, if the address is $a = 49 = 110001$, then the row address is $1100 = 12$, and the column address is 01. Thus select line w_{12} goes high to select the row containing d_{48} through d_{51} and the multiplexer selects the second word of this line, d_{49}.

While the address bits are not split evenly, the ROM of Figure 8.36 is realized using a *square* array of bits. There are 64 rows and each row contains 64 bits — 4 words of 16 bits each. Square arrays tend to give the most efficient memory layouts because they minimize the peripheral (decoder and multiplexer) overhead.

In practice, ROMs employ highly optimized circuits. For clarity, we illustrate a ROM here with conventional logic symbols — a tri-state buffer at each location. In fact, most ROMs use circuits that require only a single transistor (or its absence) to store each bit.

We could write a Verilog case statement with the contents of a ROM and synthesize a logic block that implements the ROM. However using a regular ROM structure has several advantages. First, with optimized circuit design and layout, a ROM is typically much smaller, and often much faster, than an equivalent logic circuit. Second, the regular structure allows us to change the contents of the ROM without changing its layout. Thus, we can design a chip with a particular sized ROM and then change the contents of the ROM without changing the global layout of the chip — only small changes to the internals of the ROM are required. Some ROMs are programmable by changing only a single metal layer making changing ROM contents relatively inexpensive.

The contents of most ROMs is determined at the time the ROM is manufactured — by the presence or absence of a transistor. Programmable ROMs, or PROMs are manufactured without a pattern and are programmed electrically later — by blowing a fuse or placing charge on a floating gate. Using PROMs makes low-volume applications more economical by removing the tooling costs otherwise required to configure a ROM. Some PROMs are one-time programmable - once programmed they cannot be changed. Eraseable programmable ROMs or EPROMs can be erased and reprogrammed multiple times. Some EPROMs are erased by exposure to UV light (UV-EPROMs) while others are electrically eraseable (EEPROMs).

8.7 Read-Write Memories (RAMs)

A *read-write memory* or RWM is like a ROM but also allows the contents of the table to be changed or *written*. For historical reasons read-write memories are commonly called RAMs.[2] The use of the term RAM to refer to a RWM is almost universal, so we will adopt it as well. Strictly speaking, a RAM is a sequential logic device — it has state and hence its outputs depend on its input

[2] The acronym RAM stands for *random-access memory*. Both ROMs and RAMs as we have defined them allow random access — i.e., any location can be accessed in any order. In contrast a tape is a sequential access memory — words stored on the tape must be accessed in sequential order.

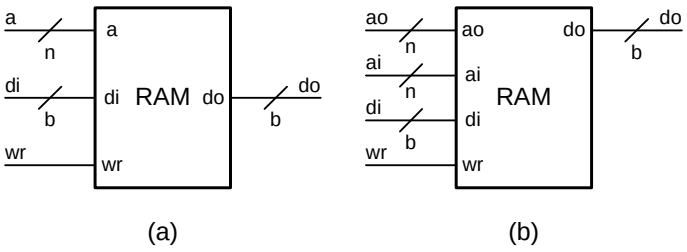


Figure 8.37: Schematic symbols for RAMs. The data input *di* is written to the selected location if the write line *wr* is true. (a) A single port RAM shares address lines for both read and write. (b) A dual-port RAM has separate address lines: *ao* for read and *ai* for write.

history – and so we should defer discussing RAMs until Chapter 14. However, we will go ahead and discuss the basics of RAMs here.

A schematic symbol for a single-port RAM is shown in Figure 8.37(a). If the write signal *wr* is low, the RAM performs just like the ROM. Applying an address to input *a* results in the contents of the corresponding location being output on data output *do*. When signal *wr* goes high, a write takes place. The value on data input *di* is written to the location specified by *a*. We can use this RAM to store a value in a location — by addressing the location and setting *wr* high. Then at a later time we can read out the stored value — by addressing the location again with *wr* low.

In the single-port RAM of Figure 8.37(a), only one location, specified by the single address *a* can be addressed at a time. If we are writing the location specified by *a* we cannot read a different location at the same time. A dual-port RAM overcomes this limitation. The schematic symbol for a dual-port RAM is shown in Figure 8.37(b). With the dual-port RAM, the read port (signals *ao* and *do*) is independent from the write port (signals *ai*, *di*, and *wr*). A location specified by *ao* can be read onto data output *do* at the same time data input *di* is being written to a different location specified by *ai*. Dual-port RAMs are often used to interface two subsystems with one subsystem writing the RAM and the other subsystem reading it.

A simple implementation of a dual-port RAM using two decoders, latches, and tri-state buffers is shown in Figure 8.38. The read decoder and tri-state buffers form a structure identical to the ROM of Figure 8.35. The read address *ao* is decoded to N read word select lines $wo_0, \ldots, wo_{N-1}$. Each read word select line enables the corresponding location onto the read output *do*.

The difference between the read port and the ROM is that the data stored at each location is obtained from a latch in the RAM (rather than being a constant in the ROM). A latch is a simple storage element that copies its input D to its output Q when its enable G input is high. When G is low, the output Q holds its previous value — giving us a simple one-bit memory. Latches are described

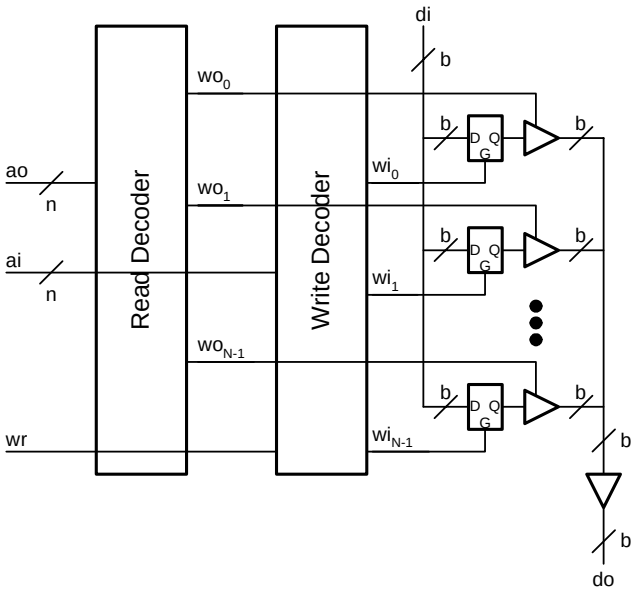


Figure 8.38: A dual-port RAM can be implemented with two decoders, latches to store the data, and an array of tri-state buffers for reading. Actual RAMs use a 2-D structure and more efficient circuit design for the storage element.

in more detail in Section 23.1.

The write port of the RAM uses the write decoder to decode the write address ai to write word select lines $wi_0, \ldots, wi_{N-1}$ when wr is asserted. When wr is not asserted, all write word select lines remain low. When location i is written, write word select line wi_i goes high storing the input data on di into the i^{th} latch. When the address changes or wr goes low, wi_i goes low and the latch retains the stored data.

As with the ROM, actual RAM implementations are much more efficient than the simple one we illustrate here. Most RAMs employ a 2-D structure (like the one shown for a ROM in Figure 8.36. The column "multiplexing" for writes is a bit more involved and will not be discussed further here. Most practical RAMs also use a much more more efficient bit cell than the latch plus tri-state buffer shown here. Most static RAMs (SRAMs) use a six-transistor storage cell while modern dynamic RAMs (DRAMSs) use a cell composed of a single transistor and a storage capacitor. The circuit details of these RAM cells is beyond the scope of this book.

8.8 Programmable Logic Arrays

A programmable logic array or PLA is a regular structure than can be configured to realize an arbitrary set of sum-of-products logic functions. As shown in Figure 8.39, a PLA consists of an AND-plane and an OR-plane. The AND-plane is a 2-D structure with literals (inputs and their complements) running vertically and product terms running horizontally. In each row, an arbitrary set of literals is selected as inputs to an AND gate to realize an arbitrary product term. The literals connected to each AND gate are denoted by squares in the figure. For example, the top AND gate takes as input three literals: a_0, $\overline{a_1}$, and $\overline{a_2}$.

The AND plane is much like the decoder in a ROM, except that the product term for each row is arbitrary while the product term for each row of a decoder is the minterm corresponding to the address of that row. Several rows of a PLA may go high at the same time, while only one row of a ROM's decoder will be activated at a time.

The OR-plane is another 2-D structure: the product terms run horizontally and the outputs (sums) run vertically. In each column, an arbitrary set of product terms is OR-ed together to form an output. In the figure, the products combined by each OR are denoted by squares. For example, the rightmost column ORs together the bottom three products.

In practice PLAs usually use the same structure for both the AND and OR planes — either a NAND or a NOR gate. By Demorgan's Law, A NAND-NAND PLA is equivalent to the AND-OR PLA shown in Figure 8.39. A NOR-NOR PLA realizes the complement of a function and can be followed by an inverter to realize a function directly. A highly optimized circuit structure is often used in which each crosspoint in a plane requires only a single transistor (or its absence).

Most PLAs are hard-wired at manufacturing time. The literals in each

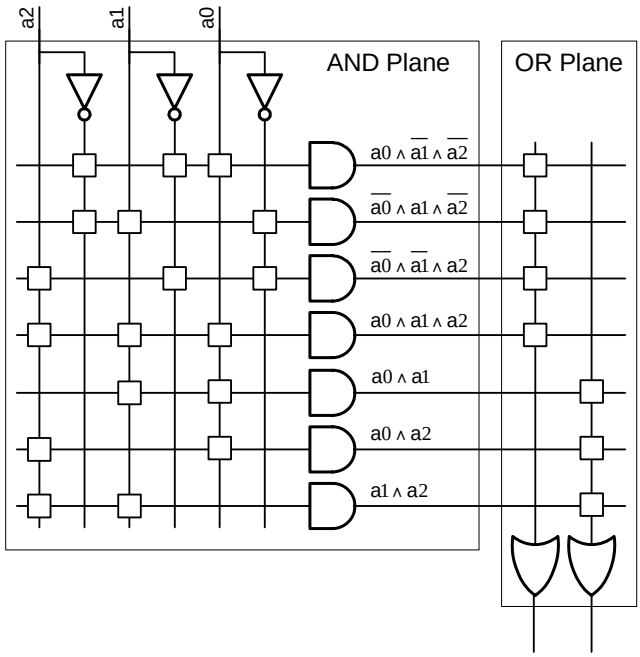


Figure 8.39: A programmable-logic array (PLA) consists of an AND-plane and an OR-plane. By programming the connections an arbitrary set of product terms can be realized by the AND plane. These product terms are then combined into sum-of-products realizations of logic functions by the OR plane. This figure shows a full adder implemented as a PLA with seven product terms in the AND plane and two sums (for sum and carry) in the OR plane.

product and the products in each sum are selected by the presence or absence of a transistor. Some PLAs are configurable with a bit of storage controlling whether a literal is included in a product or a product in a sum. Such configurable PLAs are many times larger than an equivalent hard-wired PLA.

8.9 Data Sheets

We often will use a building block, or an entire subsystem, in a larger design without understanding its implementation. When we use a building block in this manner we rely on the specification of the block. This specification, often called a *data sheet* gives enough information to use the block, but omits internal details of how the block is constructed. A data sheet typically contains:

1. A functional description of the block - what the block does. This should be in sufficient detail as to completely specify the block's behavior. For a combinational block, a truth table or equation is often used to specify the block's function.
2. A detailed description of the inputs and outputs of the block. A signal by signal description that gives the signal name, width, direction, and a brief description of the signal.
3. A description of all block parameters, if any.
4. A description of all of the visible state and registers in the block (for sequential blocks).
5. The synchronous timing of the block. The cycle-level timing of the block.
6. The detailed timing. The timing of input and output signals within a single cycle.
7. Electrical properties of the block: power requirements, power consumed, input and output signal levels, input loads and output drive levels.

We defer discussion of numbers 5 and 6 until after we have discussed sequential circuits and timing.

An example data sheet for a hypothetical $4 \rightarrow 16$ decoder is shown in Figure 8.40. This data sheet describes the behavior of a module without describing its implementation. The function of the module here is specifed by a formula (`b = 1<<a`). We could just as easily have used a sixteen row truth table. The timing section specifies the propagation and contamination delay (see Chapter 15) of the module in picoseconds (ps). Finally, the electrical section gives the input load in femptoFarads (fF) and the output resistance (drive) in kiloOhms (kOhms).

A building block that is a physical chip has the actual values of its electrical and timing parameters in the data sheet. For a Verilog block that has yet to be synthesized, these parameters are not yet known. The capacitive load of

```
Name: decode_4_16

Description: 4 to 16 decoder

Inputs:
  Name  Width  Direction  Description
  a     4      in         binary input
  b     16     out        one-hot output

Function:
  b = 1<<a

Timing:
  Parameter Min    Max    Units Description
  t_dab             300    ps    Delay from a to b - no load on b
  t_cab     100             ps    Contamination delay from a to b - no load on b

Electrical
  Parameter Min    Max    Units Description
  c_a               20     fF    Capacitance of each bit of a
  r_b       5               kOhms Effective output resistance of each bit of b
```

Figure 8.40: Example data sheet for a 4 to 16 decoder

```
set_max_delay 0.2 -from {a} -to {b}
set_driving_cell -lib_cell INV4 {a}
set_load -pin_load 5 {b}
```

Figure 8.41: Example constraint file for 4 to 16 decoder

each input, for example, is not known until after the block is synthesized and physical design of the block is complete.

A *constraint file* is used to specify targets for timing and electrical parameters. These targets (or constraints) are then used to direct the synthesis and physical design tools. A very simple constraint file for our $4 \rightarrow 16$ decoder is shown in Figure 8.41. This file is in a form suitable for use by the Synopsys Design Compiler. The file specifies that the delay of the decoder, from a to b must not exceed 0.2 ns. Rather than specify input load, the file specifies that input a is driven by a cell equivalent in drive to a `INV4` cell. If the synthesizer makes the input capacitance too large, the delay of this cell driving a, which is included in the total delay, will make it hard to meet the timing constraint. Finally, the file specifies that the output load, on each bit of b, is 5 (capacitive units). The synthesizer must size the output driver for the decoder large enough to drive this load without excessive delay.

8.10 Intellectual Property (IP)

A design team builds a chip by combining modules they design themselves with modules that they obtain from other sources. The modules that are obtained elsewhere are often called IP, for *intellectual property*.[3]

IP blocks are available both from vendors, some of which specialize in particular types of IP — e.g., MIPS and ARM specialize in selling microprocessors as IP. The open source movement that has revolutionized software has its parallel in the hardware world. Many useful pieces of Verilog IP are available for free under an open source license at `http://www.opencores.org`. Modules available include processors, interfaces (e.g., ethernet, PCI, USB, etc...), encryption/decryption, compression/decompression and others. While these building blocks are more complex than the ones described in this chapter, the concepts involved in using them are the same. The design team builds a system by combining a number of blocks. The blocks themselves are described by data sheets (and constraint files) that specify their function, interfaces, and parameters.

[3]The term *intellectual property (IP)* is much broader than its use here. IP encompasses anything of value which is independent of a physical object. That is, the value has been created by intellectual effort — rather than by manufacturing. For example, all software, books, movies, music, designs — including Verilog designs — is IP.

8.11 Bibliographic Notes

The TTL Data Book [**?**], first published in the 1970s describes the building block functions available as separate chips in the classic 7400-series TTL logic family. The chips include simple gates, multiplexers, decoders, seven-segment decoders, arithmetic functions, registers, counters (see Chapter **??**), and many others. The TTL Data Book also gives many good examples of data sheets. The function, interfaces, electrical parameters, and timing parameters are listed for each part.

The circuit design of RAMs, ROMs and PLAs are described in more detail in ???

IP

8.12 Exercises

8–1 *Decoder.* Write a Verilog description of a 3:8 decoder.

8–2 *Decoder logic.* Implement a seven-segment decoder using a 4 to 16 decoder and OR gates. (Optional - implement only a subset of the segments.)

8–3 *Two-Hot Decoder* Consider the alphabet of two-hot signals, that is binary signals with only two bits equal to one. There are $\frac{n(n-1)}{2}$ n-bit two-hot symbols. Assume these symbols are ordered by their binary value: i.e., for $n = 5$ the order is 00011, 00101, 00110, ..., 11000. Design a $4 \rightarrow 5$ binary to two-hot decoder.

8–4 *Distributed multiplexer.* You need to implement a large (32-input) multiplexer in which each multiplexer input and its associated select signal is in a different part of a large chip. The 32 inputs and selects are located along a line spaced 0.4mm apart. Show how this can be implemented using static CMOS gates (e.g., NANDs, NORs, and inverters - no tri-states) with only a single wire running along the line between adjacent input locations.

8–5 *Multiplexer logic.* Implement a 4-bit Fibonacci circuit (output true if input is a Fibonacci number) using a 8:1 binary-select multiplexer.

8–6 *Decoder test bench.* Write a test bench for a 4:16 decoder using an encoder as a checker.

8–7 *Two-Hot Encoder* Design a $5 \rightarrow 4$ two-hot encoder using the conventions of Exercise 8–3.

8–8 *Programmable priority encoder.* Design an arbiter with programmable priority - an input (one hot) selects which bit is highest priority. The priority rotates from that bit position.

8–9 *Comparator* Write a verilog module for an arbitrary-width magnitude comparator that propapagates information downward from MSB to LSB - as shown in Figure 8.32

8–10 *ROM logic - prime number function* Implement the 4-bit prime number function using a ROM. How large a ROM is needed (what are N and b)? What is stored in each location?

8–11 *ROM logic - seven-segment decoder* Implement a seven-segment decoder with a ROM. How large a ROM is needed (what are N and b)? What is stored in each location?

8–12 *PLA - prime number function* Implement the 4-bit prime number function using a PLA. How many product terms and sum terms are needed? What are the connections for each term?

8–13 *PLA - seven-segment decoder* Implement a seven-segment decoder with a PLA. How many product terms and sum terms are needed? What are the connections for each term?

Chapter 9

Combinational Examples

In this chapter we work several examples of combinational circuits to reinforce the concepts in the preceding chapters. A multiple-of-3 circuit is another example of an iterative circuit. The *tomorrow* circuit from Section 1.5 is an example of a counter circuit with subcircuits for modularity. A priority arbiter is an example of a building-block circuit - built using modules described in preceding chapters. Finally, a circuit to play tic-tac-toe gives a complex example combining many concepts.

9.1 Multiple-of-3 Circuit

In this section we develop a circuit that determines if an input number is a multiple of 3. We implement this function using an iterative circuit (like the magnitude comparator of Section 8.5). A block diagram of an iterative multiple-of-3 circuit is shown in Figure 9.1. This circuit checks our input number one bit at a time starting at the MSB. At each bit, we compute the remainder so far (0, 1, or 2). At the LSB we check if the overall remainder is 0. Each bit cell takes the remainder so far to its left and one bit of the input and computes the remainder so far to its right.

The verilog module for the bit cell of our iterative Multiple-of-3 circuit is shown in Figure 9.2. The remainder in `remin` represents the remainder from the neighboring bit to the left, and hence has a weight of 2 relative to the current bit position In our neighboring bit this signal represented a remainder of 0,1, or 2. However in the present bit, this value is shifted to the left one and it represents a value of 0, 2, or 4. Hence we can concatenate `remin` with the current bit of the input, `in`, to form a 3-bit binary number and then take the remainder (mod 3) of this number. A case statement is used to compute the new remainder.

The top-level multiple-of-3 module is shown in Figure 9.3. This module instantiates eight copies of the bit cell of Figure 9.2. The cells are connected together by passing 2-bits of remainder from one cell to the next via the 16-bit signal `rem`. Finally, the output is generated by comparing the remainder out to

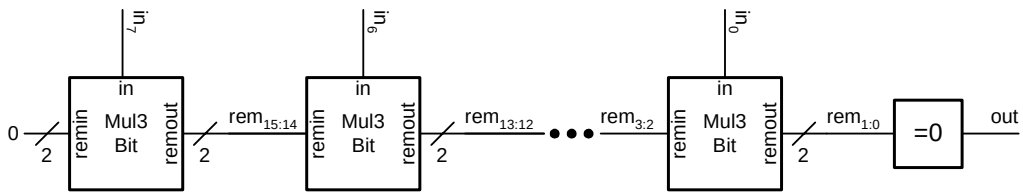


Figure 9.1: Block diagram of a multiple-of-3 circuit. The circuit computes the remainder, mod 3, of the input one bit at a time working from the left (MSB) to the right (LSB). Each bit cell computes the remainder of a 3-bit number formed by concatenating the remainder in (`remin`) with the current bit of the input (`in`). If the remainder out of the low bit is zero, the number is a multiple of 3.

zero.

While this module accepts an 8-bit input, it is straightforward to build a multiple-of-3 circuit of any length by instantiating and linking up the appropriate number of bit cells.

A test bench for the multiple-of-3 circuit is shown in Figure 9.4. The test bench checks the result of the circuit under test by checking the remainder (mod 3) using the verilog modulo operator `%`. Note that we do not want to use the % operator in our circuit itself because use of this operator will cause the synthesis program to instantiate a prohibitively expensive divider. However, use of the % operator in a test bench, which is not synthesized, causes no problems.

The test bench declares input, and output signals, instantiates the multiple-of-3 module, and then walks through all possible input states. In each input state the output of the module under test is compared to an output computed using the `%` operator. If there is a mismatch, an error is flagged. If all states are tested with no mismatch, the test passes.

9.2 Tomorrow Circuit

In Section 1.5 we introduced a calendar circuit. The key module of this circuit was a *tomorrow* circuit that given today's date in month, day-of-month, day-of-week format computes tomorrow's date in the same format. In this section we present the Verilog implementation of this tomorrow circuit.

A key step in designing a digital circuit is dividing a large problem into simpler subproblems. We can then design simple modules to solve these subproblems and compose these modules to solve our larger problem. For the tomorrow circuit we can define two subproblems:

1. Increment the day of the week. This is completely independent of the month or day of the month.

2. Determine the number of days in the current month.

```
//------------------------------------------------------------------------------
// Multiple_of_3_bit
// Cell for iterative multiple of 3 circuit.
// Determines the remainder (mod 3) of the number from this bit to the MSB.
// Input:
//    in - the current bit of the number being checked
//    remin - the remainder after the last bit checked (2 bits)
// Output:
//    remout - the remainder after checking this bit (2 bits).
//
// remin has weight 2 since its from the bit to the left, thus {remin, in}
// forms a 3 bit number.  We divide this number by 3 and produce the remainder
// on remout.
//------------------------------------------------------------------------------
module Multiple_of_3_bit(in, remin, remout) ;
  input in ;
  input [1:0] remin ;
  output [1:0] remout ;
  reg [1:0] remout ;

  always @(in, remin) begin
    case({remin, in})
      3'd0: remout = 0 ;
      3'd1: remout = 1 ;
      3'd2: remout = 2 ;
      3'd3: remout = 0 ;
      3'd4: remout = 1 ;
      3'd5: remout = 2 ;
      3'd6: remout = 0 ;
      3'd7: remout = 1 ;
    endcase
  end
endmodule
```

Figure 9.2: Verilog description of bit cell for Multiple-of-3 circuit.

```
//------------------------------------------------------------------------------
// Multiple_of_3
// Determines if input is a multiple of 3
// Input:
//    in - an 8-bit binary number
// Output:
//    out - true if in is a multiple of 3
//------------------------------------------------------------------------------
module Multiple_of_3(in, out) ;
  input [7:0] in ;
  output out ;

  wire [15:0] rem ; // two bits of remainder per cell

  // instantiate 8 copies of the bit cell
  Multiple_of_3_bit b7(in[7],2'b0,rem[15:14]) ;
  Multiple_of_3_bit b6(in[6],rem[15:14],rem[13:12]) ;
  Multiple_of_3_bit b5(in[5],rem[13:12],rem[11:10]) ;
  Multiple_of_3_bit b4(in[4],rem[11:10],rem[9:8]) ;
  Multiple_of_3_bit b3(in[3],rem[9:8],rem[7:6]) ;
  Multiple_of_3_bit b2(in[2],rem[7:6],rem[5:4]) ;
  Multiple_of_3_bit b1(in[1],rem[5:4],rem[3:2]) ;
  Multiple_of_3_bit b0(in[0],rem[3:2],rem[1:0]) ;

  // output is true if remainder out is zero
  wire out = (rem[1:0] == 2'b0) ;
endmodule
```

Figure 9.3: Verilog description of bit cell for Multiple-of-3 circuit.

```
module testMul3 ;
  reg [7:0] in ;
  reg error ;
  wire out ;

  Multiple_of_3 dut(in, out) ;

  initial begin
    in = 0 ; error = 0 ;
    repeat(256) begin
      #100
      // $display("%d %b",in,out) ;
      if(out != ((in %3) == 0)) begin
        $display("ERROR %d -> %b",in,out) ;
        error = 1 ;
      end
      in = in + 1 ;
    end
    if(error == 0) $display("PASS") ;
  end
endmodule
```

Figure 9.4: Verilog description of the test bench for the multiple-of-3 circuit.

```
module NextDayOfWeek(today, tomorrow) ;
  input [2:0] today ;
  output [2:0] tomorrow ;

  wire [2:0] tomorrow = (today == `SATURDAY) ? `SUNDAY : today + 3'd1 ;
endmodule
```

Figure 9.5: Verilog description of the `NextDayOfWeek` module which increments the day of the week.

Figure 9.5 shows a Verilog module that increments the day of the week. If the current day is `` `SATURDAY `` (which is defined to be 7) this module sets tomorrow's day to be `` `SUNDAY `` (which is defined to be 1). If the current day is other than `` `SATURDAY `` the module just increments today to get tomorrow.

In Section 16.1 we will see that the `NextDayOfWeek` module is the combinational part of a *counter*, a circuit that increments its state. The variation here is that when the counter reaches `` `SATURDAY `` it resets back to `` `SUNDAY ``.

This module is coded using definitions for `` `SATURDAY `` and `` `SUNDAY ``. However it will only work if the days are represented by consecutive three-bit integers starting with `` `SUNDAY `` and ending with `` `SATURDAY ``. In Exercise 9–2 we explore writing a more general version of this module that will work with arbitrary representations for the days of the week.

Figure 9.6 shows a Verilog module that computes the number of days in a given month. This is just a simple case statement that uses a `default` case to handle the common case of a month with 31 days. This module would be a bit easier to read if we defined constants for the month names. However, we use numbers for months often enough in our daily lives that little is lost by using the numbers here.

The astute reader will have observed that this `DaysInMonth` module is not quite right. We haven't considered leap years — when February has 29 days. We leave it as an exercise for the reader (Exercise 9–3) to remedy this situation.

With our two submodules defined, we can now develop the full `Tomorrow` module. Figure 9.7 shows the code for our full tomorrow module. After the module, input/output, and signal declarations it starts by instantiating our two submodules. The `NextDayOfWeek` module directly generates the `tomorrowDoW` output. This is also the only code to use the `todayDoW` input. The day-of-week function is completely independent of the month and day-of-month function.

Next, the circuit instantiates the `DaysInMonth` submodule. This submodule generates an internal signal `daysInMonth` that encodes the last day of the current month. The tomorrow module then generates two other internal signals: `lastDay` is true if today is the last day of the month and `lastMonth` is true if the current month is December. Using these two internal signals, the module then computes `tomorrowMonth` and `tomorrowDoM` using assign statements that use `? :` statements.

```
module DaysInMonth(month, days) ;
  input [3:0] month ; // month of the year 1 = Jan, 12 = Dec
  output [4:0] days ; // number of days in month

  reg [4:0] days ;

  always @(month) begin
    case(month)
      4,6,9,11: days = 5'd30 ; // thirty days have September...
      default: days = 5'd31 ;  // all the rest have 31
      2: days = 5'd28 ;        // except for February which has 28
    endcase
  end
endmodule
```

Figure 9.6: Verilog description of a module that computes the number of days in a month in a non-leap year.

Verifying the tomorrow circuit efficiently is a bit of a challenge. A brute force enumeration of all states requires simulating the circuit for 7 years worth of input dates — 2,555 inputs. We can reduce this to 365 inputs by observing that the day-of-week function is completely independent and can be verified independently. We can further collapse the set of tests by simulating only the beginning and end of each month.

9.3 Priority Arbiter

Our second example is a 4-input *priority arbiter*, a circuit which accepts four inputs and outputs the index of the input with the highest value. In the event of a tie, the circuit outputs the lowest index that has the highest value. As an example of the arbiter's function, suppose the four inputs are: 28, 32, 47, and 19. The arbiter will output 2 because input 2 has the highest value, 19. If the four inputs are: 17, 23, 19, 23, the arbiter will output 1 because of the two inputs (1 and 3) with the high value of 23, input 1 has the lowest index.

This circuit is used, for example, in networking equipment where the next packet to send is selected according to a quality-of-service (QoS) policy that gives each packet a score. The packet with the highest score is sent first. In this application, the score for each packet is computed an input to the priority arbiter the arbiter then selects the packet to transmit.

Our implementation of the priority arbiter is shown in Figure 9.8 and a Verilog implementation is given in Figure 9.9. The implementation performs a tournament to select the winning input. In the first round, inputs 0 and 1 and inputs 2 and 3 are compared. The second round compares the winners of the

```
module Tomorrow(todayMonth, todayDoM, todayDoW,
                tomorrowMonth, tomorrowDoM, tomorrowDoW) ;
  input [3:0] todayMonth ;  // today
  input [4:0] todayDoM ;
  input [2:0] todayDoW ;
  output [3:0] tomorrowMonth ;  // tomorrow
  output [4:0] tomorrowDoM ;
  output [2:0] tomorrowDoW ;
  wire [3:0] tomorrowMonth ;
  wire [4:0] tomorrowDoM ;

  // compute next day of week
  NextDayOfWeek ndow(todayDoW, tomorrowDoW) ;

  // compute days in current month
  wire [4:0] daysInMonth ;
  DaysInMonth dim(todayMonth, daysInMonth) ;

  // compute month and day of month
  wire lastDay = (todayDoM == daysInMonth) ;
  wire lastMonth = (todayMonth == `DECEMBER) ;
  assign tomorrowMonth = lastDay ? (lastMonth ? `JANUARY : todayMonth + 4'd1)
                                 : todayMonth ;
  assign tomorrowDoM = lastDay ? 5'd1 : todayDoM + 5'd1 ;
endmodule
```

Figure 9.7: Verilog description of a tomorrow circuit. The circuit accepts today's date in month, day-of-month, day-of-week format and outputs tomorrow's date in the same format.

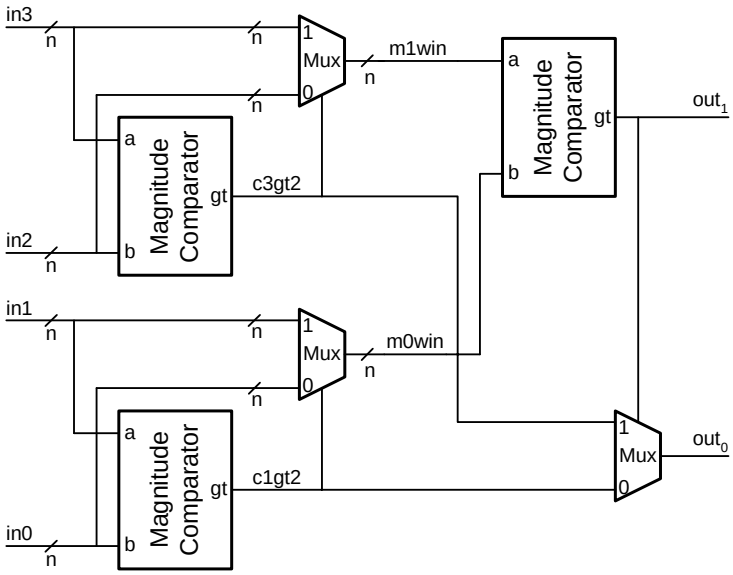


Figure 9.8: A four-input priority arbiter. This circuit accepts four inputs and outputs the index of the input with the largest value. It operates by performing a *tournament* on the inputs to find the highest value and then selecting match results to compute the index.

first round.

Each match in the tournament is performed using a magnitude comparator (Section 8.5). To break ties in favor of the lower number input, the magnitude comparator computes a signal `c1gt0` that is true if `in1` ¿ `in0`. If they are tied, this signal is false, indicating that `in0` has won the match. A similar comparison is made between `in3` and `in2`.

To select the competitors for the second round, two 2:1 multiplexers (Section 8.2) are used. Each multiplexer selects the winner of a first round match using the comparator output as the select signal.

A third magnitude comparator performs the second round match - comparing the two winners output by the multiplexers. The output of this second round comparator is the MSB of the priority arbiter. If this signal is true, the winner is `in2` or `in3`; if its false, the winner is `in0` or `in1`.

To get the LSB of the priority arbiter output, we select the output of the winning first round comparator. This is accomplished with a single-bit-wide 2:1 multiplexer controlled by the output of the final comparator.

9.4 Tic-Tac-Toe

In this section we develop a combinational circuit that plays the game of tic-tac-toe. Given a starting board position, it selects the square on which to play its next move. Being a combinational circuit, it can only play one move. However, it can easily be transformed into a sequential circuit (Chapter 14) that plays an entire game.

Our first task is to decide how to represent the playing board. We represent the input board position as two nine-bit vectors: one `xin` encodes the position of the X's and the other `oin` encodes the position of the O's. We map each nine-bit vector to the board as shown in Figure 9.10(a). The upper left corner is the LSB and the bottom right corner is the MSB. For example, the board shown in Figure 9.10(b) is represented by `xin` = 100000001 and `oin` = 00011000. For a legal board position, `xin` and `oin` must be orthogonal. That is `xin` $\wedge$ `oin` = 0.

Strictly speaking, a legal board should also have $N_O \geq N_X \geq N_O - 1$, where N_O is the number of bits set in `oin` and N_X is the number of bits set in `xin`. If X goes first, the input should always have equal numbers of bits set in the two inputs. If O goes first, the input will always have one more bit set in `oin` than in `xin`.

Our output will also be a nine-bit vector `xout` that indicates which position our circuit will be playing. A legal move must be orthogonal to both input vectors. On the next turn `xin` will be replaced by the OR of the old `xin` and `xout` and the opponent will have added a bit to `oin`.

Now that we have represented the board, our next step is to structure our circuit. A useful structure is as a set of ordered strategy modules that each apply a strategy to generate the next move. The highest priority module that is able to generate a move is selected. For example, a good set of strategy modules is:

```
//------------------------------------------------------------------------------
// 4-input Priority Arbiter
// Outputs the index of the input with the highest value
// Inputs:
//   in0, in1, in2, in3 - n-bit binary input values
// Out:
//   out - 2-bit index of the input with the highest value
//
// We pick the "winning" output via a tournament.
// In the first round we compare in0 against in1 and in2 against in3
// The second round compares the winners of the first round.
// The MSB comes from the final round, the LSB from the selected first round.
//
// Ties are given to the lower numbered input.
//------------------------------------------------------------------------------
module PriorityArbiter(in0, in1, in2, in3, out) ;
  parameter n = 8 ; // width of inputs
  input [n-1:0] in0, in1, in2, in3 ;
  output [1:0] out ;
  wire [n-1:0] match0winner, match1winner ;
  wire [1:0] out ;

  // first round of tournament
  MagComp #(n) round0match0(in1, in0, c1gt0) ; // compare in0 and in1
  MagComp #(n) round0match1(in3, in2, c3gt2) ; // compare in2 and in3

  // select first round winners
  Mux2 #(n) match0(in0, in1, {c1gt0, ~c1gt0}, match0winner) ;
  Mux2 #(n) match1(in2, in3, {c3gt2, ~c3gt2}, match1winner) ;

  // compare round0 winners
  MagComp #(n) round1(match1winner, match0winner, out[1]) ;

  // select winning LSB index
  Mux2 #(1) winningLSB(c1gt0, c3gt2, {out[1], ~out[1]}, out[0]) ;
endmodule
//------------------------------------------------------------------------------
```

Figure 9.9: Verilog description of a four-input priority arbiter.

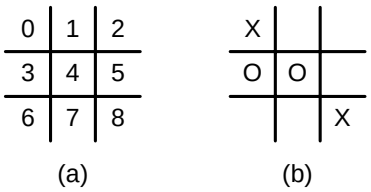


Figure 9.10: Representation of tic-tac-toe board: (a) Mapping a bit vector to the board. (b) Board represented by xin = 100000001, oin = 00011000.

1. *Win:* If a move will complete three-in-a-row do it.
2. *Don't lose:* If a move will block an opponent with two-in-a-row do it.
3. *Pick first open square:* Traversing the board in a particular order, pick the first square that is open.

A selection circuit combines the inputs from our modules and selects the highest priority module with an output. With this modular design, we can easily add more strategy modules later to refine the ability our our circuit.

The top-level module for our tic-tac-toe move generator is shown in Figure 9.11 and the Verilog for this module is given in Figure 9.12. It instantiates four modules: two instances of `TwoInArray`, and one instance each of `Empty` and `Select3`. The first `TwoInArray` module finds spaces (if any) where a play would cause us to win - that is spaces where there is a row, column, or diagonal with two Xs and no Os. The second `TwoInArray` module finds spaces (if any) where if we didn't play, the opponent could win on their next play - spaces where a row, column or diagonal has two Os and no Xs. We use the same module for both the win and the block strategies because they require the same function - just with the Xs and Os reversed. The next module `Empty` finds the first empty space according to a particular ordering of the spaces. The ordering picks empty spaces in order of their strategic value. Finally, the module `Select3` takes the three outputs of the previous modules and selects the highest-priority move.

Most of the work in our tic-tac-toe implementation is done by the `TwoInArray` module shown in Figure 9.13. This module creates eight instances of the `TwoInRow` module (Figure 9.14). Each `TwoInRow` module checks one line (row, column, or diagonal). If the line being checked has two bits of `a` true and no bits of `b` true, a 1 is generated in the position of the open space. The module consists of three four-input AND gates, one for each of the three positions being checked. Note that we only check one bit of `b` in each AND gate since we are assuming that the inputs are legal, so that if a bit of `a` is true, the corresponding bit of `b` is false.

Three instances of `TwoInRow` check the rows producing their result into the nine-bit vector `rows`. If a bit of `rows` is true, playing an `a` into the corresponding space will complete a row. Similarly three instances of `TwoInRow` check the three

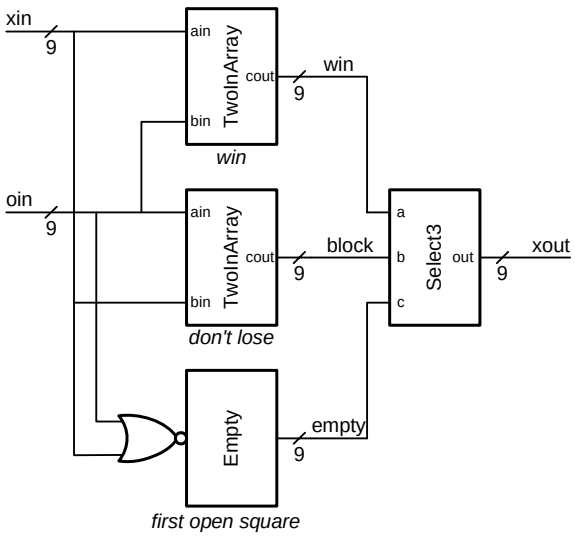


Figure 9.11: High-level design of the tic-tac-toe module. Three strategy modules accept the inputs, `xin` and `oin`, and compute possible moves to win, not lose, or pick an empty square. The `Select3` module then picks the highest priority of these possible moves to be the next move.

```
//------------------------------------------------------------------------
// TicTacToe
// Generates a move for X in the game of tic-tac-toe
// Inputs:
//   xin, oin - (9-bit) current positions of X and O.
// Out:
//   xout - (9-bit) one hot position of next X.
//
// Inputs and outputs use a board mapping of:
//
//   0 | 1 | 2
//  ---+---+---
//   3 | 4 | 5
//  ---+---+---
//   6 | 7 | 8
//
// The top-level circuit instantiates strategy modules that each generate
// a move according to their strategy and a selector module that selects
// the highest-priority strategy module with a move.
//
// The win strategy module picks a space that will win the game if any exists.
//
// The block strategy module picks a space that will block the opponent
// from winning.
//
// The empty strategy module picks the first open space - using a particular
// ordering of the board.
//------------------------------------------------------------------------
module TicTacToe(xin, oin, xout) ;
  input [8:0] xin, oin ;
  output [8:0] xout ;
  wire [8:0] win, block, empty ;

  TwoInArray winx(xin, oin, win) ;           // win if we can
  TwoInArray blockx(oin, xin, block) ;       // try to block o from winning
  Empty      emptyx(~(oin | xin), empty) ;      // otherwise pick empty space
  Select3    comb(win, block, empty, xout) ; // pick highest priority
endmodule
```

Figure 9.12: Top-level Verilog description for our tic-tac-toe move generator.

```
//------------------------------------------------------------------------
// TwoInArray
// Indicates if any row or column or diagonal in the array has two pieces of
// type a and no pieces of type b. (a and b can be x and o or o and x)
// Inputs:
//   ain, bin - (9 bits) array of types a and b
// Output:
//   cout - (9 bits) location of space to play in to complete row, column
//          or diagonal of a.
// If more than one space meets the criteria the output may have more than
// one bit set.
// If no spaces meet the criteria, the output will be all zeros.
//------------------------------------------------------------------------
module TwoInArray(ain, bin, cout) ;
  input [8:0] ain, bin ;
  output [8:0] cout ;

  wire [8:0] cout ;
  wire [8:0] rows, cols ;
  wire [2:0] ddiag, udiag ;

  // check each row
  TwoInRow topr(ain[2:0],bin[2:0],rows[2:0]) ;
  TwoInRow midr(ain[5:3],bin[5:3],rows[5:3]) ;
  TwoInRow botr(ain[8:6],bin[8:6],rows[8:6]) ;

  // check each column
  TwoInRow leftc({ain[6],ain[3],ain[0]},
                 {bin[6],bin[3],bin[0]},
                 {cols[6],cols[3],cols[0]}) ;
  TwoInRow midc({ain[7],ain[4],ain[1]},
                 {bin[7],bin[4],bin[1]},
                 {cols[7],cols[4],cols[1]}) ;
  TwoInRow rightc({ain[8],ain[5],ain[2]},
                 {bin[8],bin[5],bin[2]},
                 {cols[8],cols[5],cols[2]}) ;

  // check both diagonals
  TwoInRow dndiagx({ain[8],ain[4],ain[0]},{bin[8],bin[4],bin[0]},ddiag) ;
  TwoInRow updiagx({ain[6],ain[4],ain[2]},{bin[6],bin[4],bin[2]},udiag) ;

  //OR together the outputs
  assign cout = rows | cols |
         {ddiag[2],1'b0,1'b0,1'b0,ddiag[1],1'b0,1'b0,1'b0,ddiag[0]} |
         {1'b0,1'b0,udiag[2],1'b0,udiag[1],1'b0,udiag[0],1'b0,1'b0} ;
endmodule
```

Figure 9.13: Verilog description of the `TwoInArray` module.

```
//------------------------------------------------------------------------
// TwoInRow
// Indicates if a row (or column, or diagonal) has two pieces of type a
// and no pieces of type b. (a and b can be x and o or o and x)
// Inputs:
//   ain, bin - (3 bits) row of types a and b.
// Outputs:
//   cout - (3 bits) location of empty square if other two are type a.
//------------------------------------------------------------------------
module TwoInRow(ain, bin, cout) ;
  input [2:0] ain, bin ;
  output [2:0] cout ;

  wire [2:0] cout ;

  assign cout[0] = ~bin[0] & ~ain[0] & ain[1] & ain[2] ;
  assign cout[1] = ~bin[1] & ain[0] & ~ain[1] & ain[2] ;
  assign cout[2] = ~bin[2] & ain[0] & ain[1] & ~ain[2] ;
endmodule
```

Figure 9.14: Verilog description of the `TwoInRow` module. This module outputs a 1 in the empty position of a row that contains two bits of `a` and no bits of `b`.

columns for two bits of `a` and no bits of `b`, producing results into nine-bit vector `cols`. The final two instances of `TwoInRow` check the two diagonals producing results into three-bit vectors `ddiag` and `udiag` for the downward sloping and upward sloping diagonals.

After checking the rows, columns, and diagonals, the final assign statement combines the results into a single 9-bit vector by ORing together the individual components. The `rows` and `cols` vectors are combined directly. The 3-bit diagonal vectors are first expanded to 9-bits to place their active bits in the appropriate positions.

The Empty module, shown in Figure 9.15 uses an arbiter (Section 8.4) to find the first non-zero bit in its input vector. Note that the top-level has ORed the two input vectors together and taken the complement so each 1 bit in the input to this module corresponds to an empty space. The input vector is permuted, using a concatenation statement, to give the priority order we want (middle first, then corners, then edges). The output is permuted in the same order to maintain correspondance.

The `Select3` module, shown in Figure 9.16 is also just an arbiter. In this case, a 27-bit arbiter scans all three inputs to find the first bit. This both selects the highest priority non-zero input and also selects the first set bit of this input. The 27-bit output of the arbiter is reduced to 9 bits by ORing the bits corresponding to each input together.

```
//------------------------------------------------------------------------------
// Empty
// Pick first space not in input.  Permute vector so middle comes first,
// then corners, then edges.
// Inputs:
//   in - (9 bits) occupied spaces
// Outputs:
//   out - (9 bits) first empty space
//------------------------------------------------------------------------------
module Empty(in, out) ;
  input [8:0] in ;
  output [8:0] out ;

  RArb #(9) ra({in[4],in[0],in[2],in[6],in[8],in[1],in[3],in[5],in[7]},
          {out[4],out[0],out[2],out[6],out[8],out[1],out[3],out[5],out[7]}) ;
endmodule
```

Figure 9.15: Verilog description of a `Empty` module. This module uses an arbiter to find the first empty space searching first the middle space, then the four corners, then the four edges.

```
//------------------------------------------------------------------------------
// Select3
// Picks the highest priority bit from 3 9-bit vectors
// Inputs:
//   a, b, c - (9 bits) Input vectors
// Outputs:
//   out - (9 bits) One hot output has a bit set (if any) in the highest
//         position of the highest priority input.
//------------------------------------------------------------------------------
module Select3(a, b, c, out) ;
  input [8:0] a, b, c;
  output [8:0] out ;
  wire [26:0] x ;

  RArb #(27) ra({a,b,c},x) ;

  wire [8:0] out = x[26:18] | x[17:9] | x[8:0] ;
endmodule
```

Figure 9.16: Verilog description of the `Select3` module. A 27-input arbiter is used to find the first set bit of the highest priority strategy module. A three-way OR combines the arbiter outputs.

It is worth pointing out that the entire tic-tac-toe module is at the bottom level built entire from just two module types: `TwoInRow`, and `RArb`. This demonstrates the utility of combinational building blocks.

A simple test bench for the tic-tac-toe module is shown in Figure 9.17. The test bench instantiates two copies of the `TicTacToe` module. One plays X and the other plays O. The test bench starts by checking the module that plays X, called `dut` in the test bench, with some directed testing. The five vectors check empty, win, and block strategies and check row, column, and diagonal patterns.

After the five directed patterns, the test bench plays a game of TicTacToe by ORing the outputs of each module into its input to compute the input for the next round. The results of the game (obtained by writing a script to massage the output of the `$display` statements) is shown in Figure 9.18.

The game starts with an empty board. The *empty* rule applies and X plays to the center - our highest priority empty space. The *empty* rule applies for the next two turns as well and O and X take the top two corners. At this point X has two in a row, so the *block* rule applies and O plays to the bottom left corner (position 6) completing the first row of the figure.

The second row of the figure starts with the *block* rule causing X to play on the left edge (position 3). O then blocks X in the middle row. At this point *empty* cause X to take the remaining corner. In the last two moves *empty* causes O and X to fill the two remaining open spaces. The game ends in a draw.

The verification performed by this test bench is by no means adequate to verify proper module operation. Many combinations of inputs have not been tried. To thoroughly verify the module a checker is required. This would typically be implemented in a high-level programming language (like "C") and interfaced to the simulator. Proper operation would then be verified by comparing the simulation results to the high-level language model. One hopes that the same mistake would not be made in both models.

Once a checker is in place, we still need to pick the test vectors. After a bit more directed testing (e.g., win, block, near-win, and near-block on all eight lines) we could take two approaches. We could exhaustively test the module. There are 2^{18} input cases. Depending on how fast our simulator runs we may have time to try them all. Alternatively, if we don't have time for exhaustive testing, we could apply random testing, randomly generating input patterns and checking the resulting outputs.

9.5 Exercises

9–1 *Multiple of 5 circuit.* Using an approach similar to the multiple-of-3 circuit of Section 9.1, design a multiple-of-5 circuit that outputs true iff its 8-bit input is a multiple of 5. (Optional, code your design in Verilog and exhaustively verify it with a test bench.)

9–2 *Calendar circuit.* Recode the `NextDayOfWeek` module so it will work with arbitrary definitions of the constants `` `SUNDAY, `MONDAY, ..., `SATURDAY ``.

```
module TestTic ;
  reg [8:0] xin, oin ;
  wire [8:0] xout, oout ;

  TicTacToe dut(xin, oin, xout) ;
  TicTacToe opponent(oin, xin, oout) ;

  initial begin
    // all zeros, should pick middle
    xin = 0 ; oin = 0 ;
    #100 $display("%b %b -> %b", xin, oin, xout) ;
    // can win across the top
    xin = 9'b101 ; oin = 0 ;
    #100 $display("%b %b -> %b", xin, oin, xout) ;
    // near-win: can't win across the top due to block
    xin = 9'b101 ; oin = 9'b010 ;
    #100 $display("%b %b -> %b", xin, oin, xout) ;
    // block in the first column
    xin = 0 ; oin = 9'b100100 ;
    #100 $display("%b %b -> %b", xin, oin, xout) ;
    // block along a diagonal
    xin = 0 ; oin = 9'b010100 ;
    #100 $display("%b %b -> %b", xin, oin, xout) ;
    // start a game - x goes first
    xin = 0 ; oin = 0 ;
    repeat (6) begin
      #100
      $display("%h %h %h", {xin[0],oin[0]},{xin[1],oin[1]},{xin[2],oin[2]}) ;
      $display("%h %h %h", {xin[3],oin[3]},{xin[4],oin[4]},{xin[5],oin[5]}) ;
      $display("%h %h %h", {xin[6],oin[6]},{xin[7],oin[7]},{xin[8],oin[8]}) ;
      $display(" ") ;
      xin = (xout | xin) ;
      #100
      $display("%h %h %h", {xin[0],oin[0]},{xin[1],oin[1]},{xin[2],oin[2]}) ;
      $display("%h %h %h", {xin[3],oin[3]},{xin[4],oin[4]},{xin[5],oin[5]}) ;
      $display("%h %h %h", {xin[6],oin[6]},{xin[7],oin[7]},{xin[8],oin[8]}) ;
      $display(" ") ;
      oin = (oout | oin) ;
    end
  end
endmodule
```

Figure 9.17: Verilog test bench for the tic-tac-toe module performs directed testing and then plays one module against another.

```
. . .      . . .      O . .      O . X      O . X
. . .      . X .      . X .      . X .      . X .
. . .      . . .      . . .      . . .      O . .


O . X      O . X      O . X      O O X      O O X
X X .      X X O      X X O      X X O      X X O
O . .      O . .      O . X      O . X      O X X
```

Figure 9.18: Results of playing one `TicTacToe` module against another.

9–3 *Calendar circuit.* Modify the calendar circuit to work correctly on leap years. Assume your input includes the year — in 12-bit binary format.

9–4 *Calendar representations.* Design a combinational logic circuit that takes a date as the number of days since January 1, 0000 and returns the date in month, day-of-month format. (Optional, also generate day-of-week).

9–5 *Ties in the priority arbiter.* The priority arbiter of Section **??** currently breaks ties in favor of the lower-numbered input. Modify the circuit so that it breaks ties in favor of the higher-numbered input. (Optional, code your design in Verilog and verifiy it on selected test cases.)

9–6 *A five-input priority arbiter.* Modify the priority arbiter of Section **??** to take five inputs.

9–7 *Inverted priority.* Modify the priority arbiter to pick the input with the lowest value.

9–8 *Tic-tac-toe.* Extend the tic-tac-toe module of Section 9.4 by adding strategy modules to implement the following:
(a) Play to a space that creates two in a row. Create a module called `OneInARow` that finds rows, columns, and diagonals with one X and no Os. Use this module to build a module `OneInArray` that implements this strategy.
(b) On an empty board play to space 0 (upper left corner).
(c) On a board that is empty except for an opponent O in two opposite corners and you X in the middle, play to an adjacent edge space. (Play to the space marked H in the diagram below. [1])

```
O . .
H X .
. . O
```

[1] If you play to a corner in this situation, your opponent O will win in two moves.

9–9 *Tic-tac-toe.* Add a module to the tic-tac-toe module that checks if the inputs are legal.

9–10 *Tic-tac-toe.* Add a module to the tic-tac-toe module that outputs a signal when the game is over and indicates the outcome. The signal should encode the options: playing, win, lose, draw.

9–11 *Verification.* Build a checker for the tic-tac-toe module and write a test bench that performs random testing on the module.

Chapter 10

Arithmetic Circuits

Many digital systems operate on numbers, performing arithmetic operations such as addition and multiplication. For example, a digital audio system represents a waveform as a sequence of numbers and performs arithmetic to filter and scale the waveform.

Digital systems internally represent numbers in binary form. Arithmetic functions including addition and multiplication are performed as combinational logic functions on these binary numbers. In this chapter we introduce binary representations for positive and negative integers and develop the logic for simple addition, subtraction, multiplication, and division operations. In Chapter 11, we expand on these basics by looking at floating-point number representations that approximate real numbers. Finally, in Chapter 12 we look at methods for accelerating arithmetic operations.

10.1 Binary Numbers

As human beings we are used to representing numbers in *decimal* or base-10 notation. That is, we use a positional notation in which each digit is weighted by ten times the weight of the digit to its right. For example, the number $1,234_{10}$ (the subscript implies base 10) represents $1 \times 1000 + 2 \times 100 + 3 \times 10 + 4$. It is likely that we use the decimal system because we have 10 fingers on which to count.

With digital electronics, we don't have 10 fingers, instead we have two states, 1 and 0, with which to represent values. Thus, while computers can (and sometimes do) represent numbers in base 10, it is more natural to represent numbers in base 2 or *binary* notation. With binary notation, each digit is weighted by 2 times the weight of the digit to its right. For example, the number $1,011_2$ (the subscript implies base 2) represents $1 \times 8 + 0 \times 4 + 1 \times 2 + 1 = 11_{10}$.

More formally, a number $a_{n-1}, a_{n-2}, \ldots, a_1, a_0$ in base b represents the value

$$v = \sum_{i=0}^{n-1} a_i b^i. \tag{10.1}$$

For a binary number, $b = 2$ and we have:

$$v = \sum_{i=0}^{n-1} a_i 2^i. \tag{10.2}$$

Bit a_{n-1} is the leftmost or *most-significant bit* (MSB) of the binary representation while bit a_0 is the rightmost or *least-significant bit* (LSB).

We can convert from one base to another by evaluating Equation (10.1) or (10.2) in the target base, as we did above to convert $1,011_2$ to 11_{10}.

We can apply the same technique to convert from decimal to binary. For example, $1,234_{10} = 1 \times 1111101000_2 + 10_2 \times 1100100_2 + 11_2 \times 1010_2 + 100_2 \times 1 = 1111101000_2 + 11001000_2 + 11110_2 + 100_2 = 10011010010_2$. However, this procedure is a bit tedious - requiring lots of binary calculations.

Its usually more convenient to repeatedly subtract the highest power of 2 less than the number - and then add these powers of 2 to form the representation in base 2. For example, below we show converting the number $1,234_{10}$ to binary. We start with $1,234_{10}$ in the left column and repeatedly subtract the largest power of 2 that is smaller than the remaining number. Each time we subtract a value from the left column, we add the same number - but with binary representation - to the right column. At the bottom, the left column is zero, we've subtracted away the entire value of $1,234_{10}$, and the right column is $10,011,010,010_2$, the binary representation of $1,234_{10}$. We've added the entire value of $1,234_{10}$ up in this column, one bit at a time.

$$\begin{array}{rr}
1,234_{10} & 0_2 \\
\underline{-1,024_{10}} & \underline{+10,000,000,000_2} \\
210_{10} & 10,000,000,000_2 \\
\underline{-128_{10}} & \underline{+10,000,000_2} \\
82_{10} & 10,010,000,000_2 \\
\underline{-64_{10}} & \underline{+1,000,000_2} \\
18_{10} & 10,011,000,000_2 \\
\underline{-16_{10}} & \underline{+10,000_2} \\
2_{10} & 10,011,010,000_2 \\
\underline{-2_{10}} & \underline{+10_2} \\
0_{10} & 10,011,010,010_2.
\end{array} \tag{10.3}$$

Because binary numbers can get quite long - representing a 4-digit decimal number takes 11 digits - we sometimes display them using *hexadecimal* or base-16 notation. Because $16 = 2^4$, it is easy to convert between binary and

hexadecimal. We simply take a binary number in 4-bit chunks, and convert each chunk to base 16. For example $1,234_{10}$ is $10,011,010,010_2$ and $4D2_{16}$. As shown below, we simply take each 4-bit group of $10,011,010,010_2$, starting from the right, and convert the group to hexadecimal. We use the characters $A-F$ to represent single digits with values from 10-15 respectively. The character D in $4D2_{16}$ implies that the second digit (with weight 16) has a value of 13.

$$\begin{array}{cccc} 0100 & 1101 & 0010 & {}_{2} \\ 4 & D & 2 & {}_{16} \end{array} \tag{10.4}$$

Digital systems use a *binary-coded decimal* or BCD representation to encode decimal numbers. This is a representation where each decimal digit is represented by a four-bit binary number. In BCD, the value is given by:

$$v = \sum_{i=0}^{n-1} d_i 10^i = \sum_{i=0}^{n-1} 10^i \times \sum_{j=0}^{3} a_{ij} 2^j. \tag{10.5}$$

That is, each decimal group d_i of four-bits is weighted by a power of 10 and each binary digit b_{ij} within decimal group d_i is additionally weighted by a power of 2. For example, the number $1,234_{10} = 0001001000110100_{BCD}$.

$$\begin{array}{cccccc} 1 & 2 & 3 & 4 & {}_{10} \\ 0001 & 0010 & 0011 & 0100 & {}_{BCD} \end{array} \tag{10.6}$$

The reason we use binary notation to represent numbers in digital systems it that it makes common operations (addition, subtraction, multiplication, etc...) easy to perform. As always, we pick a representation suitable for the task at hand. If we had a different set of operations to perform, we might pick a different representation.

10.2 Binary Addition

The first operation we will consider is addition. We add binary numbers the same way we add decimal numbers - one digit at a time, starting from the right. The only difference is that the digits are binary, not decimal. This actually simplifies addition considerably since we only have to remember four possible combinations of digits (rather than 100).

To add two bits, a and b, together, there are only four possibilites for the result, r, as shown in Table 10.1. In the first row, we add 0 + 0 to get r = 0. In the second and third row we add 0 + 1 (or equivalently 1 + 0) to get r = 1. Finally in the last row, if both a and b are 1, we get r = 1+1 = 2.

To represent results r ranging from 0 to 2 requires 2 bits, s and c as shown in Table 10.1. The LSB, s, we refer to as the *sum* and the MSB, c, we refer to

a	b	r	c	s
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	0

Table 10.1: Truth table for a half adder

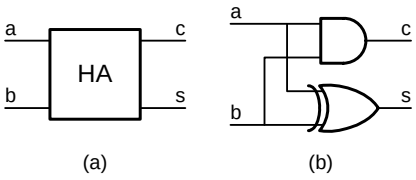


Figure 10.1: Half adder: (a) symbol, (b) logic circuit.

as the *carry*. (The reason for these names will become clear shortly when we discuss multi-bit addition.)

A circuit that adds two bits together to produce a sum and carry is called a *half adder*. The reader will have noticed that the truth table for the sum is the same as the truth table for an XOR gate and the truth table for the carry is just the truth table for an AND gate. Thus we can realize a half adder with just these two gates as shown in Figure 10.1.

To handle a carry input, we require a circuit that accepts three input bits: a, b, and ci (for carry in). And generates a result r that is the sum of these bits. Now r can range from 0 to 3, but it can still be represented by two bits, s and co (for carry out). A circuit that adds three equally weighted bits together to generate a sum and a carry is called a *full adder*, and a truth table for this circuit is shown in Table 10.2.

a	b	ci	r	co	s
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

Table 10.2: Truth table for a half adder

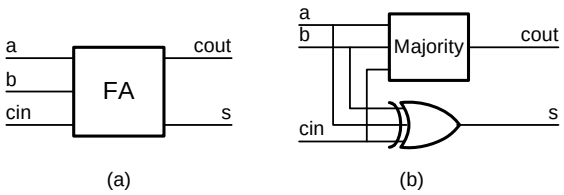


Figure 10.2: Full adder: (a) symbol, (b) logic circuit.

A full adder circuit is shown in Figure 10.2. From Table 10.2, we observe that the sum output has the truth table of a 3-input exclusive-or (i.e., the output is true when an odd number of the inputs are true). The carry output is true whenever a majority of the inputs are true (2 or 3 inputs true out of 3), and thus can be implemented using a majority circuit. (The majority function is given in Equation (3.6).)

The astute reader will have observed by now that adder circuits are really counters. A half adder or full adder just counts the number of ones on its inputs (all inputs are equivalent) and reports the count in binary form on its output. For a half adder the count is in the range of 0 to 2, and for a full adder the count ranges from 0 to 3.

We can use this counting property to construct a full adder from half adders as shown in Figure 10.3(a). In the figure the numbers in parentheses indicate the weight of a signal. The inputs are all weighted (1). The result output is a binary number with the sum weighted (1) and the carry out weighted (2). Because an adder counts the ones on its inputs, they should all be of equal weight - otherwise one input should count more than another. We use one half adder to count two of the original inputs producing a sum which we call p (for propagate) and a carry which we call g (for generate). If the propagate signal is true a one bit on carry in (cin) will cause carry out to go high. That is the carry in *propagates* to the carry out. If the generate signal is true the carry out will be true regardless of the carry in. We say that bits a and b *generate* the carry out. We will see in Section 12.1 how the generate and propagate signals are used to build very fast adders. For now, however, we will continue with our simple adder.

A second half adder combines p (of weight (1)) with the carry input (also of weight (1)) to produce the sum output s (of weight (1)) and a carry output which we call cp (for propagated carry) (of weight (2)). At this point we have a single weight (1) signal, s, and two weight (2) signals, cp and g. We use a third half adder to combine the two weight (2) signals. The sum output of this third half adder is the carry out (of weight (2)). The carry output (which would be of weight (4)) of this half adder is unused. This is because with only three inputs, there is no way for a count of four to occur.

We can simplify the circuit of Figure 10.3(a) by taking advantage of the facts

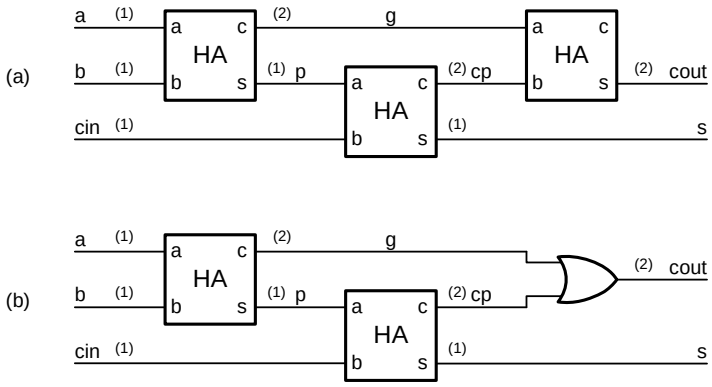


Figure 10.3: A full adder can be constructed using (a) three half adders, (b) two half adders and an OR gate. The numbers in parentheses show the weight of each signal.

that (a) we need only the sum output of the last half adder, and (b) the two inputs of the last half adder will never both be high. Fact (a) lets us replace the half adder with an exclusive-or gate. The AND gate in the half adder isn't needed because the carry output is unused. Fact (b) lets us replace the XOR gate with an OR gate (a much simpler gate to implement in CMOS) because their truth tables are identical except for the state where both inputs are high. The result is the circuit of Figure 10.3(b). Verilog for this circuit is shown in Figure 10.4.

Figure 10.5 shows an optimized CMOS logic circuit for a full adder. The circuit consists of five CMOS gate circuits, Q1 through Q5, including two two-input NAND gates, Q1 and Q3, and three three-input OR-AND-invert (OAI) gates, Q2, Q4, and Q5. Gates Q1 and Q2 form the first half adder - providing complemented outputs p' and g'. Gates Q3 and Q4 form an exclusive-NOR (XNOR) gate which acts as the XOR of the second half adder - producing output s. The input OR (low-true AND) part of gate Q5 performs the AND part of the second half adder. Signal cp is not produced, however. It remains internal to gate Q5. The output AND (low-true OR) part of gate Q5 performs the OR to combine g with cp and generates the carry out.

It is illustrative to see how these artithmetic circuits are realized in CMOS gates. Fortunately, with modern logic synthesis, you will very rarely have to work with arithmetic circuits at the gate level.

Now that we have circuits that can add single bits, we can move on to multi-bit addition. To add multi-bit numbers, we simply apply this single-bit binary addition one-bit at a time from right to left. For example, suppose we are working with 4-bit binary numbers. To add 3_{10} 0011 to 6_{10} 0110 we compute as shown below:

```
//------------------------------------------------------------------------
// half adder
module HalfAdder(a,b,c,s) ;
  input a,b ;
  output c,s ;  // carry and sum
  wire s = a ^ b ;
  wire c = a & b ;
endmodule
//------------------------------------------------------------------------
// full adder - from half adders
module FullAdder1(a,b,cin,cout,s) ;
  input a,b,cin ;
  output cout,s ;  // carry and sum
  wire g,p ;    // generate and propagate
  wire cp ;
  HalfAdder ha1(a,b,g,p) ;
  HalfAdder ha2(cin,p,cp,s) ;
  or o1(cout,g,cp) ;
endmodule
```

Figure 10.4: Verilog description of a full adder constructed from half adders.

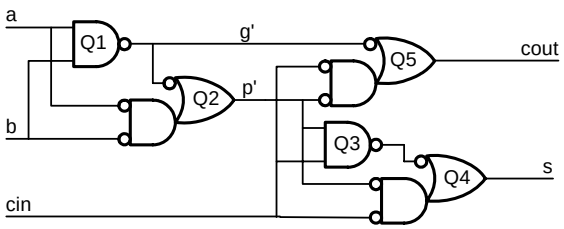


Figure 10.5: A CMOS gate-level implementation of a full adder.

```
  1 1 0
  0 1 1 0
+ 0 0 1 1
---------
  1 0 0 1
```

We start in the right-most colum by adding the two LSBs $0 + 1$ to give 1 as the LSB of the result. The result can be represented in a single bit (i.e. its less than 2), so the carry into the next column (denoted by the small blue number) is 0. With the carry, we have three bits to sum for the second column, 0, 1, and 1 the result is 2 - so the second bit of the sum is 0 and we carry a 1 to the top of the third column. In the third column the bits are 1, 1, and 0 - again 2 1's, so the sum and carry are again 0 and 1 respectively. In the fourth and final column, only the carry is a 1, so the sum is 1 and the carry (not shown) is 0. The result is $0110 + 0011 = 1001$ or $6_{10} + 3_{10} = 9_{10}$.

We can build a multi-bit adder circuit from full adders by operating in the same manner - starting from the LSB and working toward the MSB. Such a circuit is shown in Figure 10.6. The bottom full adder, FA0, sums a carry in, cin, with the LSBs of the two inputs a[0] and b[0] generating the LSB of the sum s[0] and a carry into bit 1, c[1]. We could have used a half-adder for this bit; however, we chose to use a full adder to allow us to accept a carry in. Each subsequent full adder bit, FAi, sums the carry into that bit, c[i], with that bits inputs a[i] and b[i] to generate that bit of the sum, s[i], and the carry into the next bit, c[$i + 1$].

This circuit is often referred to as a *ripple-carry adder* because, if the inputs are set correctly (exactly one of a[i] or b[i] true for all i) the carry will *ripple* from cin to cout propagating through all n full adders. For large n (more than 8) this can be quite slow. We will see in Section 12.1 how to build adders with delay proportional to $\log(n)$ rather than n.

For most applications, the appropriate way to describe an adder in Verilog is behaviorally, as shown in Figure 10.7. After declaring the inputs and outputs, the actual description here is a single line that uses the "+" operator to add single-bit `cin` to n-bit `a` and `b`. The concatenation of `cout` and `s` accepts the output.

Modern synthesis tools are quite good at taking a behavioral description, like the one shown here, and generating a very efficient logic netlist. There is rarely any reason to describe an adder in more detail.

For illustrative purposes, an alternative Verilog description of an Adder is shown in Figure 10.8. This module describes the bit-by-bit logic of a ripple-carry adder in terms of AND, OR, and XOR operations. The description defines n-bit propagate and generate variables and then uses them to compute the carry. The definition of the carry uses a concatenation and subfield specification to make bit i of the carry a function of bit $i - 1$. This is not a circular definition.

While this description is useful for showing the logical definition of an adder, it may generate a logic netlist that is inferior to the behavioral description of Figure 10.7. This is because the synthesis tool may not recognize it as an adder - and hence not perform its special adder synthesis. In contrast, when you use

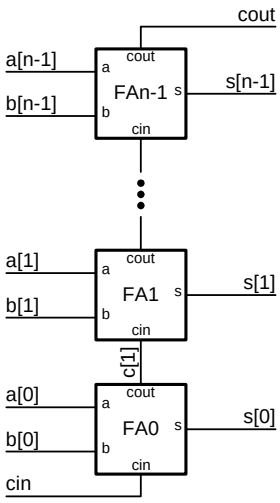


Figure 10.6: A multi-bit binary adder.

```
// multi-bit adder - behavioral
module Adder1(a,b,cin,cout,s) ;
  parameter n = 8 ;
  input [n-1:0] a, b ;
  input cin ;
  output [n-1:0] s ;
  output cout ;
  wire [n-1:0] s;
  wire cout ;

  assign {cout, s} = a + b + cin ;
endmodule
```

Figure 10.7: Behavioral Verilog description of a multi-bit adder. This description uses Verilog's "+" primitive to describe addition.

```
// multi-bit adder - bit-by-bit logical
module Adder2(a,b,cin,cout,s) ;
  parameter n = 8 ;
  input [n-1:0] a, b ;
  input cin ;
  output [n-1:0] s ;
  output cout ;

  wire [n-1:0] p = a ^ b ;                      // propagate
  wire [n-1:0] g = a & b ;                      // generate
  wire [n:0]   c = {g | (p & c[n-1:0]), cin} ;  // carry = g | p & c
  wire [n-1:0] s = p ^ c[n-1:0] ;               // sum
  wire cout = c[n] ;

endmodule
```

Figure 10.8: Bit-wise logical Verilog description of a ripple-carry adder.

the "+" operator, there is no doubt that the circuit being described is an adder. The logical description is also harder to read and maintain. Without the module name, variable names, and comments, you would have to study this module for a while to discern its function. In contrast, using "+", its immediately obvious to a reader (as well as a synthesis tool) what you mean.

Our n-bit adder (of Figures 10.6 through 10.8) accepts two n-bit inputs and produces an $n+1$-bit output. This ensures that we have enough bits to represent the largest possible sum. For example with a 3-bit adder, adding binary 111 to 111 gives a four-bit result 1110. In many applications, however, we need an n-bit output. For example, we may want to use the output as a later input. In these cases we need to discard the carry out and retain just the n-bit sum. Restricting ourselves to an n-bit output raises the spectre of *overflow* - a condition that occurs when we compute an output that is too large to be represented as n bits.

Overflow is usually an error condition. It is easily detected; any time carry out is one, an overflow has occured. Most adders perform modulo arithmetic on an overflow condition - they compute $a + b \pmod{2^n}$. For example with a three-bit adder adding 111 + 010 gives 001 ($7_{10} + 2_{10} = 1_{10} \pmod{8_{10}}$). In Exercise 10–6 we will look at a saturating adder which takes a different approach to producing an output during an overflow condition.

10.3 Negative Numbers and Subtraction

With $n-$bit binary numbers, using Equation (10.2), we can represent only non-negative integers up to a maximum value of $2^n - 1$. We often refer to binary

numbers that represent only positive integers as *unsigned* numbers (because they don't have a + or − sign). In this section we will see how to use binary numbers to represent both positive and negative integers, often referred to as *signed* numbers. To represent signed numbers, we have three main choices: 2's complement, 1's complement, and sign-magnitude.

The simplest system conceptually is sign-magnitude. Here we simply add a sign bit, s to the number with the convention that if $s = 0$ the number is positive and if $s = 1$ the number is negative. By convention we place the sign bit in the left-most (MSB) position. Consider the numbers $+23_{10}$ and -23_{10}. In sign-magnitude representation, $+23_{10} = 010111_2$ and $-23_{10} = 110111_{2SM}$. All that changes between the two numbers is the sign bit. Our value function becomes:

$$v = -1^s \times \sum_{i=0}^{n-1} a_i 2^i. \tag{10.7}$$

To negate a 1's complement number, we complement all of the bits of the number. So to negate our example number, $+23_{10} = 010111_2$, we get $-23_{10} = 101000_{2OC}$. The value function becomes:

$$v = -a_{n-1} * (2^{n-1} - 1) + \sum_{i=0}^{n-2} a_i 2^i. \tag{10.8}$$

Here the sign bit, a_{n-1}, is weighted, but by $-(2^{n-1} - 1)$ (a number that is all 1s in binary representation - hence the name 1's complement).

Finally to negate a 2's complement number, we complement all of the bits of the number and then add one. For our example number, $+23_{10} = 010111_2$, $-23_{10} = 101001_2$. The value function becomes:

$$v = -a_{n-1} * 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i. \tag{10.9}$$

Compared to one's complement, the weight on the sign bit has been decreased by one to -2^{n-1}.

So which of these three formats should we use in a given system? The answer depends on the system. However, the vast majority of digital systems use a 2's complement number system because it simplifies addition and subtraction. We can add positive or negative 2's complement numbers directly, using binary addition and get the correct answer. The same is not true of sign/magnitude or 1's complement.

Consider for example adding +4 and −3 to get a result of +1 represented as 4-bit signed binary numbers. The inputs and outputs of this computation are shown for the three number systems below. For 2's complement adding +4

(0100) to −3 (1101) gives +1 (10001) - the correct answer if we ignore the carry (more on carry and overflow below). In contrast, just adding the 1's complement numbers gives 10000[1], and adding the sign-magnitude numbers gives 1111[2].

	2's comp	1's comp	sign-mag
+4	0100	0100	0100
−3	1101	1100	1011
+1	0001	0001	0001

To see why 2's complement numbers make adding negative numbers easy, it is instructive to review how we generate a two's complement integer. We complement the bits and add one. Complementing the bits of a number x gives $2^n - 1 - x$ (15-x for 4-bit integers). For example, $15 - 3 = 12$ which is 1100 in binary (the 1's complement of 3). The 2's complement of x is one more than this or $2^n - x$ ($16 - x$ for 4-bit integers). For example, $16 - 3 = 13$ which is 1101 in binary (the 2's complement of 3). Because all addition is performed mod 2^n, the 2's complement of a number, $2^n - x$ is the same as $-x$. Hence we get the correct result. Returning to our example we have

$$\begin{aligned} 4 - 3 &= 4 + (16 - 3) \pmod{16} \\ &= 17 \pmod{16} \\ &= 1 \end{aligned} \tag{10.10}$$

It is often helpful when thinking about 1's complement or 2's complement arithmetic to visualize the numbers on a wheel as shown in Figure 10.9. Here we show the four bit numbers from 0000 (at the 12 o'clock position) to 1111 incrementing in a clockwise direction around a circle. In Figure 10.9(a), (b), and (c) we show the values assigned to these bit patterns by the 2's complement, 1's complement, and sign-magnitude number systems respectively.

One thing that becomes immediately apparent from the figure is that 1's complement and sign-magnitude do not have a unique representation for zero. In 1's complement for example, both 0000 and 1111 represent the value 0. This makes comparison difficult. An equality comparator (Section 8.5) cannot by itself determine if two 1's complement or sign-magnitude numbers are equal because one may be +0 and the other −0 which is equivalent.

More importantly the circle lets us see the effect of modular arithmetic. Adding $-x$ to a number has the effect of moving $16 - x$ steps clockwise around the circle which is exactly the same as moving x steps counterclockwise around the circle. For example, −3 is equivalent to moving 13 steps clockwise or 3 steps counterclocwise, so adding −3 to any value between −5 and +7 gives the correct result. (Adding −3 to a value between −6 and −8 results in an overflow because we cannot represent results less than −8.)

[1] In Exercise 10–9 we see how a 1's complement adder can be built by using an end-around carry.

[2] Sign-magnitude addition of negative numbers is performed by first converting to 1's complement or 2's complement

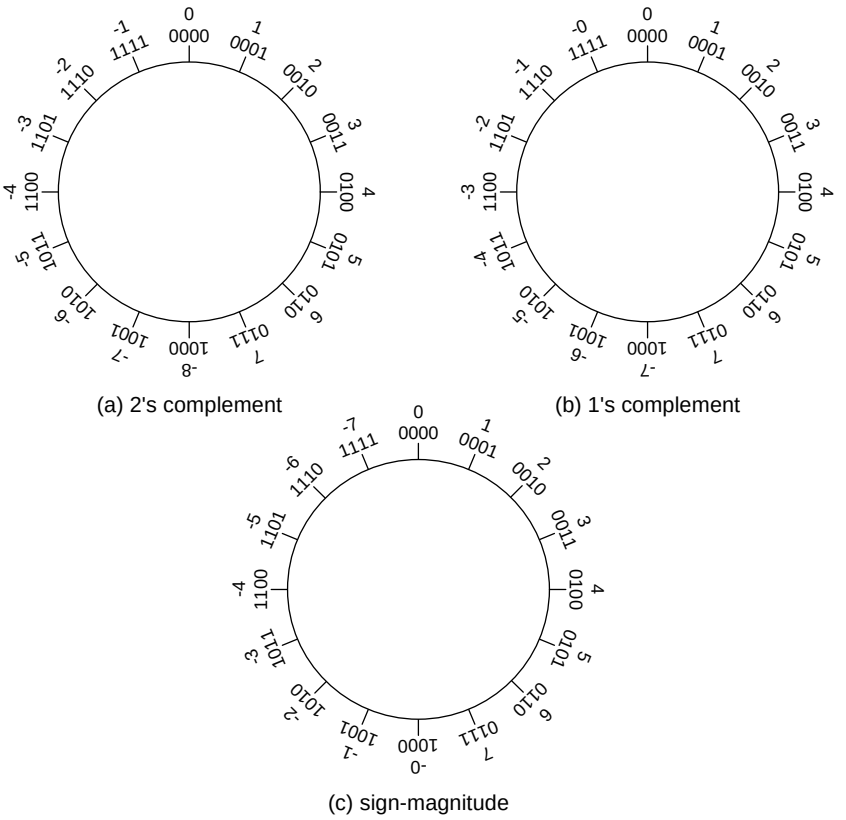


Figure 10.9: Number wheel showing three encodings of negative numbers (a) 2's complement, (b) 1's complement, and (c) sign-magnitude.

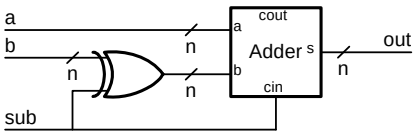


Figure 10.10: A 2's complement add/subtract unit.

How do we detect overflow when adding 2's complement numbers? We saw above that we may generate a carry as a result of modular arithmetic and get the correct answer? However, we can still generate results that are out of range. For example what happens when we add -3 to -6 or $+4$ to $+4$? We get $+7$ and -7 respectively - both incorrect. How do we detect this to signal an overflow?

The key thing to observe here is that the signs changed. We can always add a positive number to a negative number (or vice-versa) and get a result that is in range. An overflow will only occur if we add two numbers of the same sign and get a result of the opposite sign. Thus we can detect overflows by comparing the signs of the inputs and outputs. [3]

Now that we can add negative numbers, we can build a circuit to subtract. A subtractor accepts two 2's complement numbers, a and b, as input and outputs $q = a - b$. A circuit to both add and subtract is shown in Figure 10.10. In add mode, the sub input is low, so the XORs pass the b input unchanged and the adder generates $a + b$. When the sub input is high, the XORs complement the b input and the carry into the adder is high, so the adder generates $a + \bar{b} + 1 = a - b$.

Figure 10.11 shows how we can augment our add/subtract circuit to detect overflow with three gates. The first XOR gate detects if the two input signs are different (sid) the second XOR determines if an input sign is different than the output sign (siod). The AND gate checks if the two input signs are the same (sid = 0) and different than the output sign (siod = 1). If so, then overflow has occured.

We can simplify the overflow detection to a single XOR gate as shown in Figure 10.12. This simplification is based on an observation about the carries into and out of the sign bit. Table 10.3 enumerates the six cases - inputs positive, different, or negative and carry in 0 or 1. When the input signs are different, (p = 1, g = 0) and the carry into the sign bit will propagate. Thus the carry in and out are the same in this case. When the inputs are both positive, (p = 0, g = 0) a carry into the sign bit indicates an overflow and will not propagate. Finally, if the inputs are both negative (g = 1), an overflow will occur unless there is a carry into the sign bit. Thus, we see that overflow occurs iff the carry into the sign bit (cis) and the carry out of the sign bit (cos) are different.

Verilog code for the add/subtract unit is shown in Figure 10.13. This code instantiates a 1-bit adder to add the sign bits and an $n-1$ bit adder to add the

[3] We shall see below that we can accomplish the same function by comparing the carry into the last bit with the carry out of the last bit.

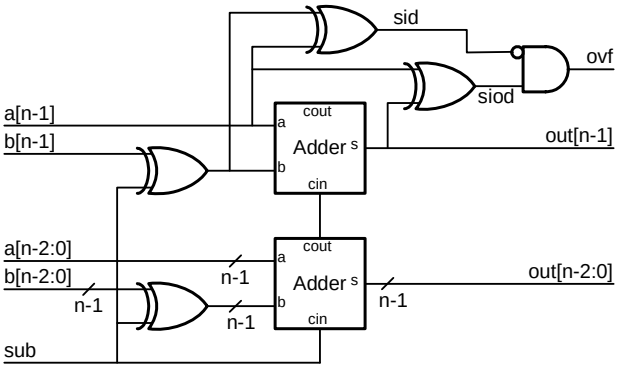


Figure 10.11: A 2's complement add/subtract unit with overflow detection based on comparison of sign bits.

as	bs	cis	qs	cos	ovf	comment
0	0	0	0	0	0	Both inputs positive, both carrys 0, no overflow
0	0	1	1	0	1	Both inputs positive, carry in 1, overflow
0	1	0	1	0	0	Input signs different, carry in 0, no overflow
0	1	1	0	1	0	Input signs different, carry in 1, no overflow
1	1	0	0	1	1	Both inputs negative, carry in 0, overflow
1	1	1	1	1	0	Both inputs negative, carry in 1, no overflow

Table 10.3: Cases for inputs and carry into sign bit of adder to detect overflow. Columns show sign bit of a and b (as and bs) carry into and out of sign bit (cis and cos) and output of sign bit (qs). Overflow only occurs if the carrys into and out of the sign bit are different.

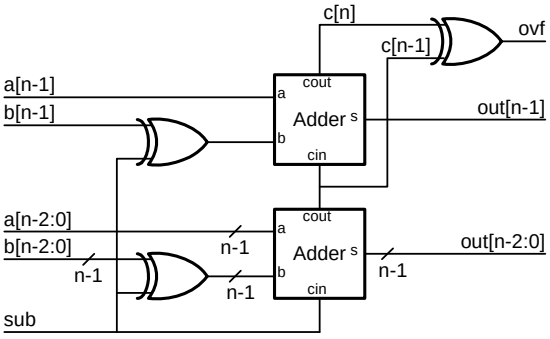


Figure 10.12: A 2's complement add/subtract unit with overflow detection based on carry in and out of the last bit.

```
// add a+b or subtract a-b, check for overflow
module AddSub(a,b,sub,s,ovf) ;
  parameter n = 8 ;
  input [n-1:0] a, b ;
  input sub ;           // subtract if sub=1, otherwise add
  output [n-1:0] s ;
  output ovf ;          // 1 if overflow
  wire c1, c2 ;         // carry out of last two bits
  wire ovf = c1 ^ c2 ;  // overflow if signs don't match

  // add non sign bits
  Adder1 #(n-1) ai(a[n-2:0],b[n-2:0]^{n-1{sub}},sub,c1,s[n-2:0]) ;
  // add sign bits
  Adder1 #(1)   as(a[n-1],b[n-1]^sub,c1,c2,s[n-1]) ;
endmodule
```

Figure 10.13: Structural Verilog code for add/subtract unit with overflow dection. This implementation instantiates adder modules.

remaining bits. The XOR of the `b` input with the `sub` input is performed in the argument list for each adder.

An alternative Verilog implementation is shown in Figure 10.14. This code uses assign statements with the "+" operator in place of instantiating predefined adders. It still performs an explicit XOR on the `b` input before the add.

One might be tempted to avoid the explicit XOR and instead code an add/subtract unit (ignoring overflow) using a statement like:

```
  assign {c, s} = sub ? (a - b) : (a + b) ;
```

Don't do this! Almost all synthesis systems will generate two separate adders for this code: one to do the "+" and a second to do the "-". While this code is quite clear and easy to read, it does not synthesize well - generating twice the logic of the alternative version.

Once we have a subtractor, then, with the addition of a zero-checker on the output, we also have a comparator. If we subtract, computing $s = a - b$, then if $s = 0$ then $a = b$ and if the sign bit of s is true, then $(a - b) < 0$, so $a < b$.

When adding 2's complement signed numbers of different lengths, one must first *sign extend* the shorter number. It is required that the sign bits - which have negative weight - be in the same position. If the numbers are added without sign extension the negative weight sign bit of the shorter number will be incorrectly added to a positive weight bit of the longer number. For example, if we add 1010, a four-bit representation of -6_{10}, to 001000, a six-bit representation of $+8_{10}$, we get $010010 = 18_{10}$. This is because 1010 is misinterpreted as 10_{10}.

```
// add a+b or subtract a-b, check for overflow
module AddSub(a,b,sub,s,ovf) ;
  parameter n = 8 ;
  input [n-1:0] a, b ;
  input sub ;           // subtract if sub=1, otherwise add
  output [n-1:0] s ;
  output ovf ;          // 1 if overflow
  wire c1, c2 ;         // carry out of last two bits
  wire ovf = c1 ^ c2 ;  // overflow if signs don't match

  // add non sign bits
  assign {c1, s[n-2:0]} = a[n-2:0] + (b[n-2:0] ^ {n-1{sub}}) + sub ;
  // add sign bits
  assign {c2, s[n-1]} = a[n-1] + (b[n-1] ^ sub) + c1 ;
endmodule
```

Figure 10.14: Behavioral Verilog description of add/subtract unit with overflow detection. This implementation uses the "+" operator to perform the 2's complement add.

A 2's complement number can be sign extended by just copying the sign bit into the new positions to the left. For example, sign extending 1010 to six bits gives 111010. Our addition now becomes $111010 + 001000 = 000010 = 2_{10}$ which is the correct result.

In hardware sign extension is accomplished with no additional gates, just wiring to repeat the sign bit. In Verilog it is easily expressed using the concatenate operator. For example, if `a` is n bits long and `b` is $m < n$ bits long we sign extend `b` to n bits by writing.

```
...
parameter n = 6 ;
parameter m = 4 ;

wire [n-1:0] a ;
wire [m-1:0] b ;

... {{(n-m+1){b[m-1]}},b[m-2:0]} ... ; \\ sign extend b to n bits
```

10.4 Multiplication

We multiply binary numbers the same way we multiply decimal numbers: by shifting and adding. Shifting a binary number left by one position is the same as multiplying it by 2. For example, the number $101_2 = 5_{10}$ if we shift it left by

one position we get $1010_2 = 10_{10}$ another left shift gives $10100_2 = 20_{10}$ and so on.

To multiply two unsigned binary numbers $a_{n-1}, \ldots, a_0$ and $b_{n-1}, \ldots, b_0$, we add a copy of a shifted to the appropriate position for each position in which b is one. That is, we compute $b_0 a + b_1(a << 1) + \cdots + b_{n-1}(a << (n-1))$.

For example, consider multiplying $a = 101_2$ by $b = 110_2$ ($5_{10} \times 6_{10}$). In long notation we write:

```
    1 0 1
  x 1 1 0
-----------
    0 0 0
  1 0 1
1 0 1
-----------
1 1 1 1 0
```

Here $b_0 = 0$, hence the row of 0s in the unshifted position. We add in 101 shifted by 1 since $b_1 = 1$ and 101 shifted by 2 since $b_2 = 2$. Summing these three *partial products* gives $11110 = 30_{10}$.

A circuit to perform multiplication on two 4-bit unsigned binary numbers is shown in Figure 10.15. An array of 16 AND gates forms 16 partial products. The first row of 4 AND gates forms $b_0 a$. The second row forms $b_1 a$ shifted one position left, and so on. An array of 12 full adders then sums the partial products by columns to produce the 8-bit product $p_7, \ldots, p_0$. Partial product pp_{00} formed by $b_0 \wedge a_0$ is the only partial product of weight (1) so it directly becomes p_0. Partial products pp_{01} and pp_{10} are both of weight (2) and are summed by a full adder to give p_1. Product bit p_2 is computd by summing pp_{02}, pp_{11}, and pp_{20} along with the carry out of the weight (1) adder. The remaining bits are computed in a similar manner - by summing the partial products in their column along with carries from the previous column.

Note that all partial products in a column have indices that sum to the weight of that column: e.g., 02, 11, and 20 all sum to (2). This is because the weight of a partial product is equal to the sum of the indices of the input bits from which its derived. To see this, consider that multiplication can be expressed as

$$p = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (a_i \wedge b_j) \times 2^{i+j}. \tag{10.11}$$

Verilog code for the four-bit multiplier is shown in Figure 10.16. Four assignments form the partial products `pp0` to `pp3`, each a four-bit vector. Three four-bit adders are then instantiated to add up the partial products. The second input to each of these adders is a concatenation of the high three bits out of the previous adder and either 0 (for the first adder) or the carry out of the previous adder.

The multiplier of Figures 10.15 and 10.16 multiplies unsigned numbers. It will not produce the correct result with a 2's complement signed number on the

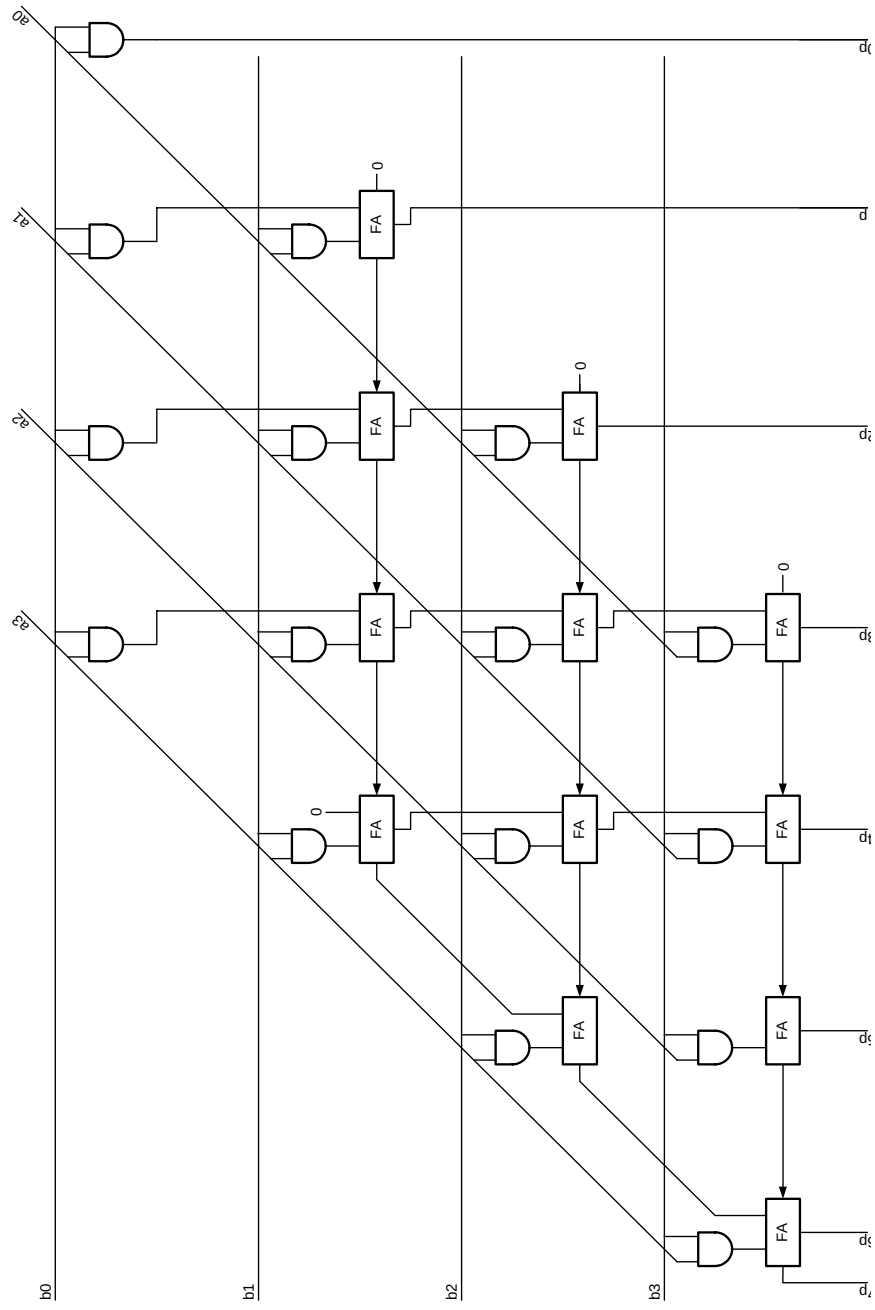


Figure 10.15: A 4-bit unsigned binary multiplier.

```
// 4-bit multiplier
module Mul4(a,b,p) ;
  input [3:0] a,b ;
  output [7:0] p ;

  // form partial products
  wire [3:0] pp0 = a & {4{b[0]}} ; // x1
  wire [3:0] pp1 = a & {4{b[1]}} ; // x2
  wire [3:0] pp2 = a & {4{b[2]}} ; // x4
  wire [3:0] pp3 = a & {4{b[3]}} ; // x8

  // sum up partial products
  wire cout1, cout2, cout3 ;
  wire [3:0] s1, s2, s3 ;
  Adder1 #(4) a1(pp1, {1'b0,pp0[3:1]}, 1'b0, cout1, s1) ;
  Adder1 #(4) a2(pp2, {cout1,s1[3:1]}, 1'b0, cout2, s2) ;
  Adder1 #(4) a3(pp3, {cout2,s2[3:1]}, 1'b0, cout3, s3) ;

  // collect the result
  wire [7:0] p = {cout3, s3, s2[0], s1[0], pp0[0]} ;
endmodule
```

Figure 10.16: Verilog code for a four-bit unsigned multiplier.

a input because the partial products are not sign extended to full width before being added. Also, it will not produce correct results with a 2's complement negative number on the b input. This is because the multiplication counts on b_3 being weighted by 8 rather than -8. We leave the modification of the multiplier to handle a 2's complement number as an exercise (Exercise 10–13). Also, the use of Booth recoding (Section 12.2) results in a multiplier that naturally handles signed numbers.

10.5 Division

At this point we know how to represent signed and unsigned integers in binary form and how to add, subtract, and multiply these numbers. To complete the four functions needed to build a simple calculator, we also need to learn how to divide binary numbers.

As with decimal numbers, we divide binary numbers by shifting, comparing and subtracting. Given a b-bit *divisor* x and a c-bit *dividend* y we find the c-bit quotient q so that $q = \lfloor \frac{y}{x} \rfloor$. We may also compute a b-bit remainder r so that $r = y - qx = y \pmod{x}$. The quotient can be the same length as the dividend. Consider for example the case where $x = 1$.

We perform the division one bit at a time - generating q from the left (MSB) to the right. We start by comparing $x'_{2b-1} = 2^{2b-1}x = x << (2b-1)$ to $r'_{2b-1} = y$. We set $q_{2b-1} = x'_{2b-1} <= r'_{2b-1}$. We then prepare for the next iteration by computing the remainder so far $r'_{2b-2} = r'_{2b-1} - q_{2b-1}x'_{2b-1}$ and the shifted divisor $x'_{2b-2} = 2^{2b-2}x = x'_{2b-1} >> 1$. At each bit i we repeat the comparison computing $q_i = x'_i <= r'_i$ and then computing $r'_{i-1} = r'_i - q_i x'_i$ and $x'_{i-1} = x'_i >> 1$.

For example, consider dividing $132_{10} = 10000100_2$ by $11_{10} = 1011_2$. The process is shown below:

```
                1 1 0 0
      ┌──────────────────
1 0 1 1 1 0 0 0 0 1 0 0   y
      - 1 0 1 1 0 0 0     x'3
        1 0 1 1 0 0       r'2
      - 1 0 1 1 0 0       x'2
                  0
```

For the first four iterations, $i = 7, \ldots, 4$ (not shown), $x'_i > y$ and no subtractions are performed. Finally, on the fifth iteration, $i = 3$ we have $x'_3 = 1011000 < r'_3 = y$, so we set bit $q_3 = 1$ and subtract to compute $r'_2 = y - x'_3 = 101100$. We shift x'_3 right to get $x'_2 = 101100$. These two values are equal, so bit $q_2 = 1$. Subtracting gives $r'_1 = 0$ so all subsequent bits of q are zero.

A six-bit by three-bit divider is shown in Figure 10.17. The circuit consists of six nearly identical stages. Stage i generates bit q_i of the quotient by comparing an appropriately shifted version of the input x'_i with the remainder from the previous stage r'_i. The remainder from stage i, r'_{i-1} is generated by a subtractor and a multiplexer. The subtractor subtracts the shifted input from the previous

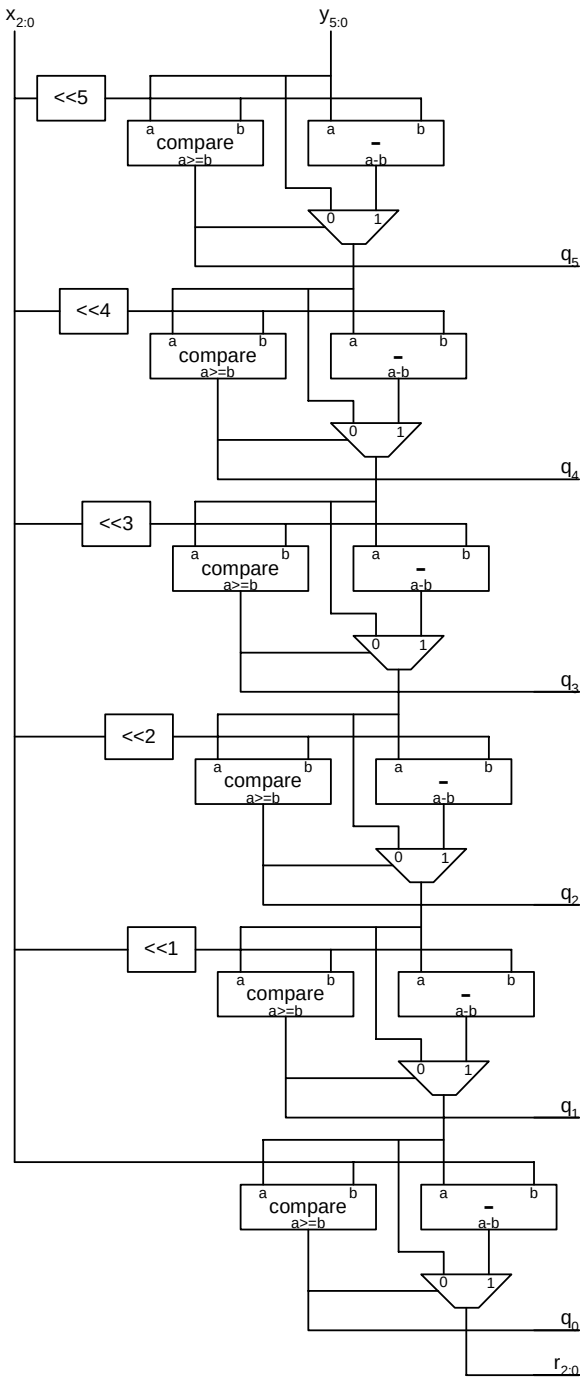


Figure 10.17: A binary divider

remainder. If the q_i is true, the multiplexer selects the result of the subtraction, $r'_i - x'_i$ to be the new remainder. If q_i is false, the previous remainder is passed unchanged. Stage 0 generates the LSB of the quotient q_0 and the final remainder, r.

A divider is quite costly in terms of gate count, but not quite as costly as Figure 10.17 indicates. For clarity, the figure shows six subtractors and six comparators. In practice the subtractor can be used as the comparator. The carry out of the subtractor can be used as the quotient bit. If the carry out of the subtractor is one, then $a >= b$. With this optimization, the circuit can be realized with six subtractors and six multiplexers.

A further optimization can be made by reducing the width of the subtractors. The first subtractor can be made a single bit wide. To see this, note that $x << 5$, has zeros in its low 5 bits. Thus the low five bits of the result will be equal to the low 5 bits of y, $y_{4:0}$, and we don't need to subtract these bits. Also note that if x has any non-zeros anywhere other than its LSB, then x¡¡5 is guaranteed to be larger than y and the result of the subtraction is not needed. Thus we need not feed the upper bits of x into the subtractor. We do, however, need to check if the upper bits of x are non-zero as part of computing the MSB of the quotient q_5. If an upper bit of x is non-zero, then $x << 5 > y$ and $q_5 = 0$. With these observations we see that the first subtractor only needs to subtract the LSB of x from the MSB of y. Hence a one-bit subtractor suffices. By a similar set of arguments we can use a two-bit subtractor for the stage that computes q_4, a three-bit for the q_3 stage, and a four-bit subtractor for the q_2 stage. The q_1 and q_0 stages can also use a four-bit subtractor. The remainder into the q_1 stage is guaranteed to be no more than 5-bits in length and the remainder into the q_0 stage is at most 4-bits long. Hence the upper bits of these subtractors can be omitted.

Verilog code for a six-bit by three-bit divider is shown in Figure 10.18. This code uses the subtractors to perform the comparison and optimizes the width of the subtractors for each stage. Each subtractor is realized using an `Adder1` module (Figure 10.7) with the second input complemented and the carry-in set to `1'b1`. Each multiplexer is implemented with an assign statement using the `? :` operation. For the first stage the upper two bits of input `x` are checked as part of the comparison and in the second stage the MSB of `x` is checked. The remainder of the verilog code follows directly from the schematic.

In addition to consuming a great deal of space, dividers are also very slow. This is because the subtract in one stage must be completed to determine the multiplexer command - and hence the intermediate remainder before the subtract in the next stage can be started. Thus, the delay through a c-bit by b-bit divider using ripple-carry adders is proportional to $c + b$ since c subtracts, of $b + 1$ bits in length (after the first b stages) must be performed. This is in contrast to a multiplier where the partial products can be summed in parallel.

```
// Six-bit by three-bit divider
//   At each stage we use an adder to both subtract and compare.
//   The adders start 1-bit wide and grow to 4 bits wide.
//   We check the bits of x to the left of the adder as part of
//   the comparison.
//   Starting with the fourth iteration (that computes q[2]) we
//   drop a bit of the remainder each iteration.  It is guaranteed
//   to be zero.
module Divide(y, x, q, r) ;
  input [5:0] y ; // dividend
  input [2:0] x ; // divisor
  output [5:0] q ; // quotient
  output [2:0] r ; // remainder
  wire [5:0] q ;
  wire co5, co4, co3, co2, co1, co0 ;  // carry out of adders

  wire sum5 ; // sum out of adder - stage 1
  Adder1 #(1) sub5(y[5],~x[0],1'b1, co5, sum5) ;
  assign q[5] = co5 & ~(|x[2:1]) ; // if x<<5 bigger than y, q[5] is 0
  wire [5:0] r4 = q[5]? {sum5,y[4:0]} : y ;

  wire [1:0] sum4 ; // sum out of the adder - stage 2
  Adder1 #(2) sub4(r4[5:4],~x[1:0],1'b1, co4, sum4) ;
  assign q[4] = co4 & ~x[2] ;  // compare
  wire [5:0] r3 = q[4]? {sum4,r4[3:0]} : r4 ;

  wire [2:0] sum3 ;   // sum out of the adder - stage 3
  Adder1 #(3) sub3(r3[5:3],~x,1'b1, co3, sum3) ;
  assign q[3] = co3 ;  // compare
  wire [5:0] r2 = q[3]? {sum3,r3[2:0]} : r3 ;

  wire [3:0] sum2 ;   // sum out of the adder - stage 4
  Adder1 #(4) sub2(r2[5:2],{1'b1,~x},1'b1, co2, sum2) ;
  assign q[2] = co2 ;  // compare
  wire [4:0] r1 = q[2]? {sum2[2:0],r2[1:0]} : r2[4:0] ; // msb is zero, drop it

  wire [3:0] sum1 ; // sum out of the adder - stage 5
  Adder1 #(4) sub1(r1[4:1],{1'b1,~x},1'b1, co1, sum1) ;
  assign q[1] = co1 ;  // compare
  wire [3:0] r0 = q[1]? {sum1[2:0],r1[0]} : r1[3:0] ; // msb is zero, drop it

  wire [2:0] sum0 ; // sum out of the adder - stage 6
  Adder1 #(4) sub0(r0[3:0],{1'b1,~x},1'b1, co0, sum0) ;
  assign q[0] = co0 ;  // compare
  wire [2:0] r = q[0]? sum0[2:0] : r0[2:0] ; // msb is zero, drop it
endmodule
```

Figure 10.18: Verilog code for a six-bit by three-bit divider.

10.6 Bibliographic Notes

Books on computer arithmetic.

10.7 Exercises

10–1 *Decimal to binary conversion.* Convert the following numbers from decimal to binary notation. Use the minimum number of bits possible. (Optional: Also express the result in hexadecimal.)
(a) 817, (b) 1492, (c) 1963, (d) 2005.

10–2 *Binary to decimal conversion.* Convert the following numbers from binary to decimal notation. (Note spaces are placed between four bit groups for convenience, ignore the spaces when interpreting the numbers.)
(a) 0011 0011 0001, (b) 0111 1111, (c) 0100 1100 1011 0010 1111, (d) 0001 0110 1101.

10–3 *Hexadecimal to decimal conversion.* Convert the following numbers from hexadecimal to decimal notation. (Optional: Also express the numbers in binary-coded decimal (BCD) notation.)
(a) 2C, (b) BEEF, (c) BABE, (d) F00D, (e) DEAD.

10–4 *Binary addition.* Add the following pairs of binary or hexadecimal numbers.

(a) $\begin{array}{r} 1010 \\ \underline{+0111} \end{array}$ (b) $\begin{array}{r} 011\ 1010 \\ \underline{+110\ 1011} \end{array}$ (c) $\begin{array}{r} 2A \\ \underline{+3C} \end{array}$ (d) $\begin{array}{r} BC \\ \underline{+AD} \end{array}$

10–5 Bit counting circuit. Using full adders, build a circuit that accepts a 7-bit input and outputs the number of inputs that are 1 as a 3-bit binary number. (Optional: Write Verilog code to represent your circuit and demonstrate its correct operation via simulation.)

10–6 Saturating adder. In some applications, particularly signal processing, it is desirable to have an adder *saturate*, producing a result of $2^n - 1$, on an overflow condition rather than producing a modular result. Design a saturating adder. You may use n-bit adders and n-bit multiplexers as basic components. (Optional: Write Verilog code for your saturating adder and demonstrate its correct operation by simulating it on representative test cases. Your code should take the width of the adder as a parameter.)

10–7 Negative numbers. Express the following decimal numbers as 7-bit two's complement binary numbers.
(a) +17, (b) −17, (c) −31, (d) −32.

10–8 Subtraction. Subtract the following pairs of two's complement binary or hexadecimal numbers:

(a) $\begin{array}{r} 0101 \\ \underline{-0110} \end{array}$ (b) $\begin{array}{r} 0101 \\ \underline{-1110} \end{array}$ (c) $\begin{array}{r} 1010 \\ \underline{-0010} \end{array}$ (d) $\begin{array}{r} 0101 \\ \underline{-0111} \end{array}$

10–9 Ones complement adder. Design an adder for one's complement numbers. (Hint: First add the two numbers normally. If there is a carry out of this first add, you need to increment the result to give the correct answer. While the straightforward solution requires an adder and an incrementer, it can be done with a single adder.) (Optional: Write Verilog code for your one's complement adder and demonstrate its correct operation by simulating it on representative test cases.)

10–10 Saturating two's complement adder. In Exercise 10–6 we saw how to build a saturating adder for positive numbers. In this exercise you are to extend this design so that it handles negative numbers - saturating in both the positive and negative direction. On overflows in the positive direction your adder is to generate $2^{n-2} - 1$, and on overflows in the negative direction it should generate -2^{n-2}. (Optional: Write Verilog code for your saturating two's complement adder and demonstrate its correct operation by simulating it on representative test cases.)

10–11 Sign-magnitude subtraction. Design a circuit that accepts two sign-magnitude binary numbers and outputs their sum, also in sign-magnitude form. (Optional: Write Verilog code for your sign-magnitude adder and demonstrate its correct operation by simulating it on representative test cases.)

10–12 Multiplication. Multiply the following pairs of unsigned binary or hexadecimal numbers:

(a) $\begin{array}{r} 0101 \\ \times\ 0101 \end{array}$ (b) $\begin{array}{r} 0110 \\ \times\ 0011 \end{array}$ (c) $\begin{array}{r} 1001 \\ \times\ 1001 \end{array}$ (d) $\begin{array}{r} A \\ \times\ C \end{array}$

10–13 Two's complement multiplier. Design a multiplier for two's complement binary numbers. Consider two approaches: (a) Sign extend the partial products to deal with input a being negative and add a "complementer" to negate the last set of partial products if b is negative. (b) Convert the two inputs to sign magnitude notation, multiply unsigned numbers, and convert the result back to two's complement. Compare the cost and performance (delay) of the two approaches. Select the approach that gives the lowest cost and show its design in terms of basic components (gates, adders, etc...). (Optional: Write Verilog code for your two's complement multiplier and demonstrate its correct operation by simulating it on representative test cases.)

10–14 Binary Division. Divide the following pairs of unsigned binary or hexadecimal numbers show each step of the process:
(a) $101110_2 \div 101_2$,
(b) $101110_2 \div 011_2$,
(c) $AE_{16} \div E_{16}$,
(d) $F7_{16} \div 6_{16}$.

10–15 Subtractor Widths for Dividers. For each of the following pairs of argument widths, determine the width of subtractor needed in each stage of a

divider:
(a) dividend 4-bits, divisor 4-bits,
(b) dividend 6-bits, divisor 4-bits,
(c) dividend 4-bits, divisor 3-bits.

Chapter 11

Fixed- and Floating-Point Numbers

In Chapter 10 we introduced the basics of computer arithmetic: adding, subtracting, multiplying, and dividing binary integers. In this chapter we continue our exploration of computer arithmetic by looking at number representation in more detail. Often integers do not suffice for our needs. For example, suppose we wish to represent a pressure that varies between 0 (vacuum) and 0.9 atmospheres with an error of at most 0.001 atmospheres. Integers don't help us much when we need to distinguish 0.899 from 0.9. For this task we will introduce the notion of a *binary point* (similar to a decimal point) and use *fixed-point* binary numbers.

In some cases, we need to represent data with a very large dynamic range. For example, suppose we need to represent time intervals ranging from 1ps (10^{-12}s) to 1 century (about 3×10^9s) with an accuracy of 1%. To span this range with a fixed-point number would require 72 bits. However if use a *floating-point* number — in which we allow the position of the binary point to vary, we can get by with 14 bits: 7 bits to represent the number and 7 bits to encode the position of the binary point.

11.1 Representation Error: Accuracy, Precision, and Resolution

With digital electronics, we represent a number, x, as a string of bits, b. Many different *number systems* are used in digital systems. A number system can be thought of as two functions R and V. The representation function R maps a number x from some set of numbers (e.g., real numbers, integers, etc...) into a bit string b: $b = R(x)$. The value function V returns the number (from the same set) represented by a particular bit string: $y = V(b)$.

Consider mapping to and from the set of real numbers in some range. Be-

cause there are more possible real numbers than there are bit strings of a given length, many real numbers necessarily map to the same bit string. Thus, if we map a real number to a bit string with R and then back with V we will almost always get a slightly different real number than we started with. That is, if we compute $y = V(R(x))$ then y and x will differ. The difference is the error of the representation. We can express error either in an abolute sense (e.g., the representation is has an error of 2mm), or relative the magnitude of the number (e.g., the representation has an error of 2%). We express the absolute error of a representation at a point x as:

$$e_a = |V(R(x)) - x| . \tag{11.1}$$

And the relative error as:

$$e_r = \left| \frac{V(R(x)) - x}{x} \right| . \tag{11.2}$$

The quality of a number representation is given by its *accuracy* or *precision*[1], the maximum error over its input range X. The absolute accuracy is given by:

$$a_a = \max_{x \in X} |V(R(x)) - x| . \tag{11.3}$$

and the relative accuracy is:

$$a_r = \max_{x \in X} \left| \frac{V(R(x)) - x}{x} \right| . \tag{11.4}$$

Naturally, the relative accuracy is not defined near $x = 0$. When we want to economically represent numbers with a given relative accuracy, floating-point numbers are often used. When we want to economically represent numbers with a given absolute accuraccy, fixed-point numbers are more efficient. We describe these two representations in the next two sections.

Sometimes people refer to the number of bits used in a number system, its *length* (the term *precision* is often misused for length - e.g., saying a system has 32-bit precision). Other times people refer to the smallest difference that can be distinguished by the number system, the *resolution* of the system. When determining the quality of the representation, neither length nor resolution is useful. What matters is accuracy.

For example, suppose we represent real numbers over the range $X = [0, 1000]$ as 10-bit binary integers by representing each real number with the nearest integer. Picking the nearest integer to a real number is often referred to as *rounding* the real number to an integer. We would then represent 512.742 as 513 or 1000000001_2 and the error of representing this number would be

[1] We use the terms accuracy and precision interchangeably in this book.

$e_a(512.742) = |512.742 - 513| = 0.258$. The error over the entire range is $a_a(X) = 0.5$ since a value half way between two integers, e.g., 512.500 has this much error whether it is rounded up or down. Note that the error here depends on the representation function R. If we choose R so that each number x is represented by the nearest integer *less than* x, then we get $e_a(512.742) = 0.742$ and $a_a(X) = 1$. For positive real numbers, applying this second representation function is often referred to as *truncating* the real number to an integer.

Again, one should not confuse *accuracy* with *resolution*. The resolution of both the rounding and truncating representations discussed above is 1.0 — integers are spaced 1 unit apart. However the accuracy of rounding is 0.5 and the accuracy of truncation is 1.0.

11.2 Fixed-Point Numbers

11.2.1 Representation

A b-bit binary fixed-point number is a representation where the value of the number $a_{n-1}, a_{n-2}, \ldots, a_1, a_0$ is given by:

$$v = 2^p \sum_{i=0}^{n-1} a_i 2^{i-n}. \tag{11.5}$$

where p is a constant giving the position of the binary point — in bits from the left end of the number.

Consider for example a fixed-point number system with $n = 4$ bit numbers with the binary point to the right of the most significant bit at $p = 1$. That is, there are three bits to the right of the binary point — the fractional part of the number — and one bit to the left of the binary point — the integral part of the number. We often use a shorthand of $p.f$ to refer to the integral and fractional bits of a number. Using this shorthand the system with $n = 4$ and $p = 1$ is a 1.3 fixed-point system. If we add an additional sign bit to the left of the integral bits we will refer to the resulting $p + f + 1$ bit system as an *sp.f* system.

The number of fractional bits $f = n - p$ determines the resolution of our number system. The resolution, or the smallest interval we can distinguish is $r = 2^{-f}$. With $f = 3$, for example, our 1.3 fixed point system has a resolution of 1/8 or 0.125. Each increment of the binary number changes the value represented by 1/8. The number of integral bits p determines the range of our number system the largest number we can represent with our system is $2^p - r$. For a signed number system, the lowest (most negative, not closest to zero) number we can represent is -2^p. The range and precision is sometimes easier to see if we rewrite Equation (11.5) as:

$$v = r \sum_{i=0}^{n-1} a_i 2^i. \tag{11.6}$$

format	number	r	integer	value	
1.3	1.011	0.125	11	1.375	(11/8)
s1.3	01.011	0.125	11	1.375	(11/8)
s1.3	11.011	0.125	-5	-0.625	(-5/8)
2.4	10.0111	0.0625	39	2.4375	(39/16)

Table 11.1: Example fixed-point numbers

To convert a binary fixed-point number to decimal, we just convert to an integer and multiply by r. Table 11.1 shows some example fixed-point numbers and their conversion to decimal and fractional representation.

To convert a decimal number to a fixed-point binary number the easiest approach is to (a) multiply the decimal number by 2^f, (b) round the resulting product to the nearest whole integer, and (c) convert the resulting decimal integer to a binary integer. For example, suppose we want to convert 1.389 to our 1.3 fixed-point format. We first multiply by 8 giving 11.112. Then we round to 11, and convert to binary giving 1.011 which represents 1.375. Hence our *error* in this representation (the difference between the represented value and the actual value) is 1.389-1.375 = 0.014 or just over 1% of the actual value. If we always round to the nearest value, the largest error over all values in the range (the accuracy of the representation) should be $r/2$ — in this case 0.0625. As we get closer to zero, this error as a percentage of the value being represented grows. For numbers close to zero, the error is 100%.

Fixed-point binary numbers are often used in signal processing applications — for example, to process audio and video streams. In these applications the range and precision is well known and the binary point can be placed so that the full range of the number system is used while eliminating (or minimizing) the possiblity of an overflow. Typically the values being represented are scaled so they fall between -1 and 1 so they can be represented in an s0.f format. For most signal processing 16-bits suffices and an s0.15 format is used.

Consider our example of representing a voltage between 0 and 10V with 10mV precision. Suppose we would like this representation to use the fewest number of bits. Its clear that we will need four bits to the left of the binary point to represent 10. To get 10mV precision, we will need 20mV resolution. Hence we need six bits to the right of the binary point - giving us a resolution of $2^{-6} = 0.015625$ and a precision of $2^{-7} = 0.0078125$. Thus, a 4.6 fixed-point format can directly represent this range of voltages to the specified precision using 10 bits.

An alternate representation would be to use a *scaled* number. If we use a 9-bit binary number, we can represent values from 0 to 511. If we then scale this number by 20mV — i.e., a count of 1 corresponds to 20mV we can then represent our 10V range with 10mV precision with just 9 bits.

11.2.2 Operations

We can perform the four basic operations on fixed-point binary numbers just as if they were integers. The same arithmetic circuits described in Chapter 10 can be used. However, we need to be careful to consider that the range and precision of the results of arithmetic operations may be different than the range and precision of the inputs.

Adding two $p.f$ fixed-point numbers gives a result that is a $(p+1).f$ fixed-point number. If we wish to force the result to be a $p.f$ fixed-point number we may encounter an *overflow* condition in which the result is outside of our representable range. For example, consider our 4.6 fixed-point representation for representing voltages. If we add two voltages together we will get a result between 0 and 20V. A 5.6 fixed-point representation is required to represent this full range.

When adding sequences of fixed-point numbers, the numbers are often added using a greater range and then scaled and rounded to fit into the desired range and precision for the result. For example, suppose we have 16 values we wish to sum up, each in s4.6 fixed-point format and each representing a voltage between -10V and 10V. However, we know the sum will result in a number between -10 and 10. We perform the summation using s8.6 format to avoid any overflows on intermediate results and then convert back to s4.6 format at the end. In some cases this final conversion is performed using *saturation* where the value is clamped to the maximum representable value if it is out of range. (See Exercise 10–6.)

To add two fixed-point numbers with different representations, it is necessary to first align the binary points of the two numbers. This is most often done by converting both numbers to a fixed-point representation that has both p and f large enough to overlap both representations. For example, consider adding the 2.3 format number 01.101 to the 3.2 format number 101.01 We first convert both numbers to 3.3 format and then add 001.101 + 101.010 giving 110.111.

When we multiply two fixed-point numbers the result has twice as many bits on both sides of the binary point as the inputs. For example, if we multiply two 4.6 fixed-point numbers, the result will be an 8.12 fixed-point number. For example, suppose we multiply a voltage signal with a range of 10V and a precision of 10mV with a current signal with a range of 10A and a precision of 10mA — both of these signals are in 4.6 format. The result is a power signal with a range of 100W and a precision of 100μW — in 8.12 format.

Many signal processors scale numbers to a 0.16 format (or s0.15 for signed numbers). Multiplying two 0.16 numbers gives an 0.32 number. A common operation is to take a dot product of two vectors in 0.16 format. To allow this operation to take place with no loss of precision, many popular signal processors have 40-bit accumulators. They accumulate up to 256 0.32 multiplication results giving a sum in 8.32 format. (For signed numbers the result is in s7.32 format.) This sum is then usually scaled and rounded to get a final result back in 0.16 format.

In most cases, a result calculated to a high precision eventually must be

rounded to the original precision. Rounding is the process of reducing the precision of a number by discarding some of the rightmost bits of the number. When rounding decimal numbers to the nearest integer we know that we should always round up if the next digit is a 5 or more and down if its a 4 or less. Binary rounding works the same way. We round up if the most significant bit discarded is a 1 and down if its a zero. For example the number .10001000 in 0.8 format is rounded to 0.1001 in 0.4 format while .10000111 in 0.8 format is rounded to 0.1000 in 0.4 format. Rounding requires an add (or at least an increment) to increment the result when rounding up — hence its not a free operation. The round can potentially change all of the remaining bits. For example rounding 0.01111000 in 0.8 to 0.4 yields 0.1000.

11.3 Floating-Point Numbers

11.3.1 Representation

High-dynamic-range numbers are often represented in *floating-point format*. In particular a floating-point format is efficient for representing a number when we need a fixed proportional (not absolute) precision. We also sometimes use floating point numbers where fixed-point numbers could be used because we don't want to go to the trouble of scaling the numbers.

A floating-point number has two components, the *exponent* e and the mantissa m. The value represented by a floating point number is given by

$$v = m \times 2^{e-x} \tag{11.7}$$

where m is a binary fraction, e is a binary integer, and x is a bias on the exponent that is used to center the dynamic range. If the bits of m are $m_{n-1}, \ldots, m_0$ and the bits of e are $e_{k-1}, \ldots, e_0$ the value is given by:

$$v = \sum_{i=0}^{n-1} m_i 2^{i-n} \times 2^{\left(\sum_{i=0}^{k-1} e_i 2^k - x\right)} \tag{11.8}$$

where x is the exponent offset or *bias* of the representation.

We refer to a floating-point number system with an a-bit mantissa and a b bit exponent as an aEb format. For example, a system with a 5-bit mantissa and a 3-bit exponent is a 5E3 system. We will also use the "E" notation to write numbers. For example the 5E3 number with a mantissa of 10010 and an exponent of 011 is 10010E011. Assuming zero bias, this number has a value of $v = 18/32 \times 8 = 4.5$.

We could also represent 4.5 as 01001E100 ($9/32 \times 16$). Most floating-point number systems disallow this second representation of 4.5 by insisting that all floating-point numbers be *normalized* by shifting the mantissa left (and decrementing the exponent) until either there is a 1 in the MSB of the mantissa or the exponent is 0. With normalized numbers we can quickly check for equality by

just comparing two numbers bit-by-bit. If numbers are unnormalized they must first be normalized (or at least aligned) before they can be compared. Some number systems take advantage of normalization by omitting the MSB of the mantissa, since its almost always one (see Exercise 11–11).

Typically when a floating-point number is stored, the exponent is stored to the left of the mantissa. For example 11001E011 would be stored in 8-bits as 01111001. Storing the exponent to the left allows integer comparison to work on floating-point numbers as long as numbers are normalized. That is if for two floating-point numbers a and b, if $a > b$ then $i_a > i_b$ where i_a and i_b are the integer interpretation of the bits of a and b respectively.

If we want to represent signed values, we typically add a sign bit to the left of the exponent. For example, in 8-bits we can represent an S4E3 number which, from left to right would contain a sign bit, a three-bit exponent, and then a four-bit mantissa. In this representation, the bit string 11001001 represents -9E4 or (with zero bias) $-9 \times 2^4 = 96$.

Floating-point numbers are just *scientific notation* applied to binary numbers. Like scientific notation floating-point numbers have an error that is proportional to the magnitude of the number. For this reason, floating point numbers are an efficient way to represent values with a specified proportional accuracy - particularly when the values in question have a high dynamic range.

For example, suppose we need to represent times from 1ns to 1000s with an accuracy of 1%. At the low end of this range we need an accuracy of 10ps and at the high end of the range we need to represent 1000s - 10^{14} times the required accuracy at the low end. A fixed-point representation would require 46 bits to represent 1000s with 10ps accuracy (20ps resolution). With a floating-point number we can take advantage of the fact that we only need an accuracy of 10s (a resolution of 20s) at the high end of the range. Hence our mantissa needs only 6 bits. We can cover the large dynamic range (of $10^{12} < 2^{40}$) by using a 6-bit exponent that can represent a range of 2^{64}. We set our exponent offset (x in Equation (**??**)) to 54 so we can represent numbers up to 2^{10}. Hence we can achieve the same relative accuracy with a 12-bit 6E6 floating point number that we can with a 46-bit 10.36 fixed-point number.

11.3.2 Denormalized Numbers and Gradual Underflow

If we disallow all denormalized numbers, we have a large gap in our representation function in that the closest number to zero we can represent (in 4E3 for example) is 1000E000, which with no bias represents 0.5. This gives a large relative error for numbers smaller than 0.5. We can reduce this relative error by allowing denormalized numbers only with an exponent of zero. We can then represent 1/4 as 0100E000, 1/8 as 0010E000 and 1/16 as 0001E000. In this case, the magnitude of the error for small numbers is reduced by a factor of 8. In general the error for small numbers is reduced by a factor of 2^{n-1} for n-bit mantissas.

This representation is often referred to as *gradual underflow* because it reduces the error due to *underflow* — when an arithmetic operation gives a result

closer to zero than the smallest number that can be represented. This solves the problem of having multiple representations for the same number. Because these denormalized numbers are restricted to have an exponent of zero, there is only one representation for each value.

To simplify the presentation, the arithmetic units we describe here do not support gradual underflow. We leave their extension to support this representation as Exercises 11–17 and **??**.

11.3.3 Floating-Point Multiplication

Multiplying floating point numbers is simple: we just multiply the mantissas and add the exponents. This doubles the number of mantissa bits and increments the number of exponent bits. We typically produce a result in the same format as the inputs by rounding the mantissa (as described above in Section **??** and discarding the extra bit generated by the exponent add. Whent the mantissa is rounded, the exponent must be adjusted to account for the bits being droppped. If there is an exponent bias, the exponent must also be adjusted to compensate for the effect of applying the bias twice. If the number cannot be represented without an extra exponent bit, an overflow is signaled. The increment required to round the mantissa may itself result in a carry into the next mantissa bit position. If this occurs, the mantissa is shifted right again, and the exponent incremented accordingly.

For example, consider multiplying 101E011 (5) by 101E100 (10), both in 3E3 format with no bias. Our goal is to produce a normalized result in the same format. Our inputs are 101E011 and 101E100. Multiplying the mantissas 101 and 101 gives 011001 (25/64) and adding the exponents 011 and 100 gives 111 (7_{10}). This is, in fact, the correct answer as $25/64 \times 2^7 = 50$. We now need to convert this result back to 3E3 format.

To give a three-bit normalized mantissa, we shift the mantissa left once and discard the low two bits. The exponent is adjusted by decrementing it once to 110.[2] Because the most significant bit discarded is zero, no increment is required for the rounding. Thus, our result, in the original format is 110E110 ($6/8 \times 2^6 = 48$). The error of 2 here was caused by dropping the LSB of the mantissa during rounding.

Figure 11.1 shows a block diagram of a floating-point multiplier and the Verilog description of this multiplier is given in Figure 11.2. The FF1 block in the figure finds the left-most one bit in `pm`, the product out of the multiplier. Because both inputs are normalized, this bit is guaranteed to be one of the left two bits of the product. Hence, we can use `pm[7]` directly to select which group of four bits from `pm` to use as `sm`, the shifted product.[3] Signal `rnd`, the first discarded bit of `pm` is used to determine whether a rounding increment is needed. Signal `xm`, the rounded version of `sm` is a five bit signal. Like `pm` it is guaranteed

[2] If the MSB of the 6-bit product were one, we would have had to shift the product three bits to normalize the mantissa and then add three to the exponent to compensate.

[3] The FF1 block in this case just selects one bit out of `pm` as shown in the Verilog. With denormalized numbers, however, a full priority encoder function is required.

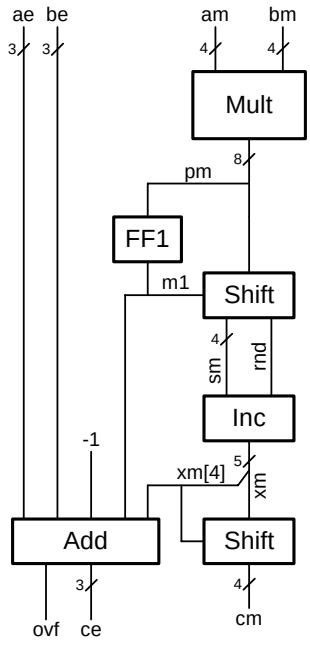


Figure 11.1: Floating-point adder

to have a one in one of its most significant two bits. Thus, we use its MSB `xm[4]` to select which group of four bits to output as the normalized mantissa. Note that we are guaranteed not to need another round after this final shift, since if `xm[4]` is a one, `xm[0]` is guaranteed to be a zero (see Exercise 11–13).

11.3.4 Floating-Point Addition/Subtraction

Floating-point addition is a bit more involved than multiplication because of the need to align the inputs and normalize the output. The process has three steps: alignment, addition, and normalization. During the alignment step the mantissa of the number with the smaller exponent is shifted right to align it with the mantissa of the number with the larger exponent — so bits of the same weight are aligned with one another. Once the two mantissas are aligned, they can be added or subtracted as if they were integers. This addition may produce a result that is unnormalized. A carry out of the add may result in a mantissa that must be shifted right by one bit for the most significant one to be placed into the MSB of the result. Alternatively a subtraction may leave many of the MSBs of the result zero, requiring a left shift of an arbitrary number of bits to place the most signficant one into the MSB of the result. The normalization step finds the most significant one of the result, shifts the result to place this one into the MSB of the mantissa, and adjusts the exponent accordingly. If the normalization is a right shift, it may discard one LSB of the result. Hence a rounding increment is needed to round, rather than truncate, when discarding this bit.

As an example of floating-point addition, suppose we wish to add the numbers 5 and 11 both represented in 5E3 notation. In this notation the number 5 is 10100E011 and the number 11 is 10110E100. During the alignment step, we shift the mantissa of 5 to the right by one bit so to align it with the mantissa of 10. In effect we are rewriting 5 as 01010E100 - denormalizing the mantissa to make the exponent agree with the other argument. With the two arguments aligned we can now add the mantissas. There is a carry out of the mantissa add, giving a result of 100000E100 in 6E3 format. To normalize this result so it fits in 5E3 format we shift the mantissa to the right one position and increment the exponent giving the final result of 10000E101 or 16.

As a second example consider subtracting 9 from 10, both represented in s5E3 format. Here 9 is +10010E100 and 10 is +10100E100. The two numbers have the same exponent, so they are already aligned. No shifting is required before the subtract. Subtracting the two numbers gives 00010E100 which is unnormalized. To normalize this number we shift the mantissa three places to the left and decrement the exponent by three giving 10000E001 or 1.

A block diagram of a floating-point adder is shown in Figure 11.3 and a Verilog description of this adder is shown in Figure 11.4. The input exponent logic compares the two exponents, generating signal `agtb` to determine which mantissa needs to be shifted, and signal `de` which gives the number of bits to shift. The input switch uses `agtb` to switch the two mantissas so the mantissa with the greater exponent is on signal `gm` and the mantissa with the smaller

```
module FP_Mul(ae, am, be, bm, ce, cm, ovf) ;
  parameter e = 3 ;
  input [e-1:0] ae, be ; // input exponents
  input [3:0] am, bm ; // input mantissas
  output [e-1:0] ce ; // result exponent
  output [3:0] cm ; // result mantissa
  output ovf ; // overflow indicator
  wire [7:0] pm ; // result of initial multiply
  wire [3:0] sm ; // after shift
  wire [4:0] xm ; // after inc
  wire rnd ; // true if MSB shifted off was one
  wire [1:0] oe ; // to detect exponent ovf

  // multiply am and bm
  Mul4 mult(am, bm, pm) ;

  // Shift/Round: if MSB is 1 select bits 7:4 otherwise 6:3
  assign sm = pm[7] ? pm[7:4] : pm[6:3] ;
  assign rnd = pm[7] ? pm[3] : pm[2] ;

  // Increment
  assign xm = sm + rnd ;

  // Final shift/round
  assign cm = xm[4] ? xm[4:1] : xm[3:0] ;

  // Exponent add
  assign {oe, ce} = ae + be + (pm[7] | xm[4]) - 1 ;
  assign ovf = |oe ;
endmodule
```

Figure 11.2: Verilog description of a floating-point multiplier

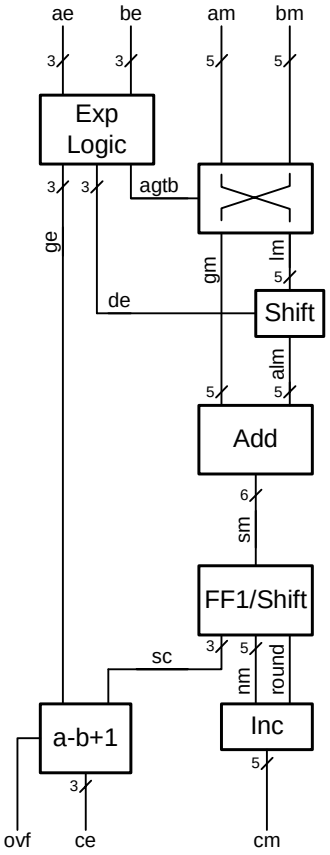


Figure 11.3: Floating-point adder

```
module FP_Add(ae, am, be, bm, ce, cm, ovf) ;
  parameter e = 3 ;
  parameter m = 5 ;
  input [e-1:0] ae, be ; // input exponents
  input [m-1:0] am, bm ; // input mantissas
  output [e-1:0] ce ; // result exponent
  output [m-1:0] cm ; // result mantissa
  output ovf ; // overflow indicator
  wire [e-1:0] ge, le, de, ce, sc ;
  wire [m-1:0] gm, lm, alm, cm, nm ;

  // input exponent logic
  wire agtb = (ae >= be) ;
  assign ge = agtb ? ae : be ;  // greater exponent
  assign le = agtb ? be : ae ;  // lesser exponent
  assign de = ge - le ;         // difference exponent

  // select input mantissa
  assign gm = agtb ? am : bm ; // mantissa with greater exponent
  assign lm = agtb ? bm : am ; // mantissa with lesser exponent

  // shift mantissa to align
  assign alm = lm>>de ; // aligned mantissa with lesser exponent

  // add
  wire [m:0] sm = gm + alm ;

  // find first one
  RevPriorityEncoder #(6,3) ff1(sm, sc) ;

  // shift first 1 to MSB
  assign {nm, rnd} = sm<<sc ;

  // adjust exponent
  assign {ovf,ce} = ge - sc + 1 ;

  // round result
  assign cm = nm + rnd ;
endmodule
```

Figure 11.4: Verilog description of a floating-point adder

exponent is on signal `lm`. Mantissa `lm` is then shifted by `de` to align the mantissas. The aligned mantissas are added, producing signal `sm`, which is one bit wider than the mantissas. A reverse priority encoder is then used to find the most significant one in `sm`. A shift is then performed to move this bit to the most significant position of the result giving signal `nm`. This shift ranges from a one-bit right shift to a full-width left shift. Signal `rnd` captures the bit discarded on a right shift by one. The exponent is adjusted to reflect the shift amount. If the exponent cannot be represented in the given number of bits, an overflow occurs.

11.4 Bibliographic Notes

The past two chapters have barely scratched the surface of the interesting topic of computer arithmetic. The interested reader is referred to one of the many excellent textbooks and monographs on the subject

Ergovic (sp?) Flynn Hwang Cavenaugh

11.5 Exercises

11–1 Fixed-point representation. Convert the following fixed-point numbers to decimal:
(a) 1.0101 in 1.4,
(b) 11.0101 in s1.4,
(c) 101.011 in 3.3,
(d) 101.011 in s2.3.

11–2 Fixed-point representation. Convert the following decimal numbers to the nearest fixed-point s1.5 representation. In each case give the error:
(a) 1.5999,
(b) 0.3775,
(c) 1.109375,
(d) -1.171875.

11–3 Fixed-point representation. Find a decimal value between -1 and 1 for which the absolute value of the error of representation as a s1.5 fixed-point number is maximum.

11–4 Fixed-point representation. Find a decimal value between 0.1 and 1 for which the percent of the error of representation as a s1.5 fixed-point number is maximum.

11–5 Fixed-point representation. You need to represent a relative pressure signal with a range from -10 PSI to 10 PSI with an accuracy of 0.1 PSI. Select a fixed-point representation that covers this range with the specified accuracy with a minimum number of bits.

11–6 Fixed-point representation. Select a fixed-point representation that covers a range from 0.001 to 1 with an accuracy of 1% across the range and uses a minimum number of bits.

11–7 Floating-point representation. Convert the following floating-point numbers to decimal:
(a) 1111E111 in 4E3
(b) 1010E100 in 4E3
(c) 1100E001 in s3E3
(d) 0101E101 in s3E3.

11–8 Floating-point representation. Convert the following decimal numbers to s3E5 floating-point format. In each case give the error:
(a) -23
(b) 100,000,000
(c) 999
(d) 64.

11–9 Floating-point representation. Select a floating-point representation that covers a range from -10 to 10 with an accuracy of 0.1 and uses a minimum number of bits.

11–10 Floating-point representation. Select a floating-point representation that covers a range from 0.001 to 1 with an accuracy of 1% across the range and uses a minimum number of bits.

11–11 Implied One. Many floating-point formats omit the MSB of the mantissa. That is, they don't bother to store it. The IEEE single-precision floating-point standard, for example, stores a 24-bit mantissa in 23-bits by omitting the MSB of the mantissa. This is referred to as an *implied one*. Some formats insist that this missing MSB is always one. However this leads to *interesting* error behavior near zero. Better error characteristics can be achieved (at some cost in complexity) by implying that the MSB of the mantissa is zero, and the exponent is $1 - x$ when e is zero (the same exponent as when e is 1). A number system having this feature is said to provide *gradual underflow*.
(a) Suppose you have a 5E3 floating-point number system with a bias x of 0 and an implied 1. (The mantissa consists of an implied 1 followed by four bits). Plot the error curve over the interval [-2,2] for a system without gradual underflow.
(b) On the same axes as (a) plot the error curve for the same number system but with gradual underflow.
(c) At what value is the percent error of the system without gradual underflow largest?
(d) Is there a value range in which the gradual underflow system has a larger error than the system without gradual underflow? If so, what is this range?

11–12 Floating-Point Subtract: Extend the floating-point adder shown in Figures 11.3 and **??** to handle signed floating-point numbers and to perform floating-point subtractions. Assume that the sign bit for each input operand is provide on separate lines `as` and `bs` and the sign of hte result is to be output on line `cs`.

11–13 Floating-Point Multiply: In Section 11.3.3 we stated that if the MSB of the rounded product `xm` was a one, then its LSB must be a zero. However we gave no justification for this claim. Prove that this is true.

11–14 Floating-Point Multiply with Denormalized Numbers: Modify the design of the floating-point multiplier of Section 11.3.3 to work with denormalized inputs.

11–15 Floating-Point Addition with Underflow: A floating-point add of two normalized numbers may result in a number that cannot be represented with a one in the MSB of the mantissa, but yet is not zero. This situation is an *underflow*. Modify the adder of Section 11.3.4 to detect and signal an underflow condition.

11–16 Add with Gradual Underflow: Extend the adder design of Section 11.3.4 to handle gradual underflow — that is to handle denormalized inputs when the input exponent is zero.

11–17 Multiply with Gradual Underflow: Extend the multiplier design of Section 11.3.3 to handle gradual underflow.

11–18 Gradual Underflow and Implied One: Consider a system that uses a representation with an implied one in the MSB of the mantissa as described in Exercise 11–11. Extend this system to allow numbers with gradual underflow. Make sure that you do not create any gaps or redundant representations in your system. (Hint: Having exponents of 0 and 1 represent the same value, but one with an implied one, and one without.) Convert the following numbers to a 4E3 version of your representation:
(a) 1/8
(b) 4
(c) 1/16
(d) 32.

11–19 Add with Gradual Underflow and Implied One: Extend the adder design of Section 11.3.4 to handle gradual underflow with an implied one (Exercise 11–18.

11–20 Mul with Gradual Underflow and Implied One: Extend the multiplier design of Section 11.3.3 to handle gradual underflow with an implied one (Exercise 11–18.

11–21 Mu-Law:

Chapter 12

Fast Arithmetic Circuits

we revisit binary adders and see how to reduce their delay from $O(n)$ to $O(\log(n))$ by using carry look-ahead circuits.

12.1 Look Ahead

Recall that the adder developed in Section 10.2 is called a *ripple-carry adder* because a transition on the carry signal must *ripple* from bit to bit to affect the final value of the MSB of the sum. This ripple-carry results in an adder delay that increases linearly with the number of bits in the adder. For large adders, this linear delay becomes prohibitive.

We build an adder with a delay that increases logarithmically, rather than linearly, with the width of the adder using a dual tree structure as shown in Figure 12.1. This circuit works by computing carry *propagate* and carry *generate* signals recursively across groups of bits in the upper tree and then using these signals to generate the carry signal into each bit in the lower tree. The propagate signal p_{ij} is true, if a carry into bit i will propagate from bit i to bit j and generate a carry out of bit j. The generate signal g_{ij} is true, if a carry will be generated out of bit j regardless of the carry into bit i. We can define p and g recursively as:

$$
\begin{aligned}
p_{ij} &= p_{ik} \wedge p_{(k+1)j}(\forall k : i <= k < j), && (12.1)\\
p_{ii} &= p_i = a_i \oplus b_i, && (12.2)\\
g_{ij} &= (g_{ik} \wedge p(k+1)j) \vee g_{(k+1)j}(\forall k : i <= k < j), && (12.3)\\
g + ii &= g_i = a_i \wedge b_i. && (12.4)\\
& && (12.5)
\end{aligned}
$$

The first two equations define the propagate signals. A carry signal will propagate across a range of bits from i to j, if it propagates from i to k (p_{ik}) and then from $k+1$ to j ($p_{(k+1)j}$). This works for any choice of k from i to $j-1$.

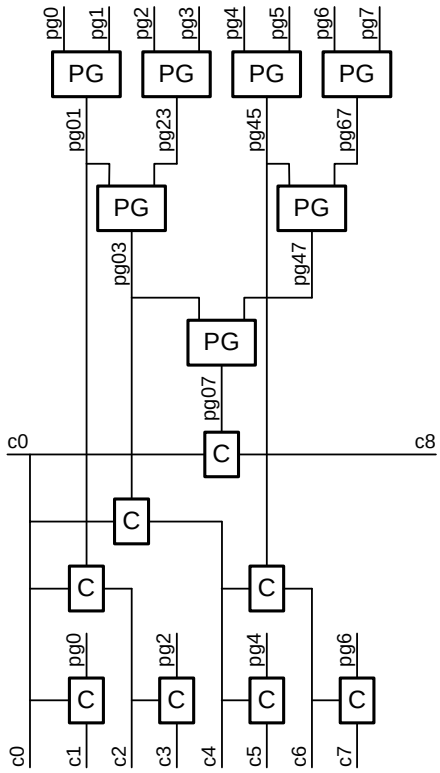


Figure 12.1: Block diagram of a carry-look-ahead adder. Each *pg* signal represents two bits, a *p* bit that specifies that the carry propagates across the specified bit range, and a *g* bit that specifies that the carry is generated out of the specified bit range. The *pg* signals are combined in a tree to span increasing ranges of bits. The *pg* signals are then used in a second tree to generate the carry, *c*, signals for each bit.

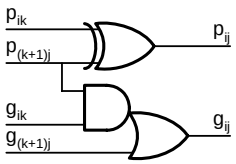


Figure 12.2: Logic to recursively generate propagate, p_{ij}, and generate, g_{ij}, signals across a group of bits from i to j as a function of the p and g signals across adjacent subranges.

Of course we usually split our intervals evenly, choosing $k = \lfloor (i+j)/2 \rfloor$. When our interval is down to a single bit, we compute p_{ii} or just p_i as discussed in conjuction with Figure 10.3. The carry propagates across a single bit i when exactly one input to that bit is true (i.e., when $a_i \oplus b_i$).

The first generate equation states that a carry signal will be generated out of bit j regardless of the carry into bit i if either (1) it is generated out of k regardless of the carry into i and then propagated across bits $k+1$ to j, or (2) it is generated out of bit j regardless of the carry into bit $k+1$. The base case for generate is as discussed with Figure 10.3. The carry is generated out of a single bit only when both inputs to that bit are high.

It is easy to construct the top part of the adder of Figure 12.1 by using Equations (12.1) to (12.4). For the 8-bit adder in the Figure, we would like to know what the carry out of bit 7, c_8 is. Hence we would like to compute p_{07} and g_{07}. To simplify the drawing, we refer to these two signals collectively as pg_{07}. We choose $k = 3$ and compute pg_{07} from pg_{03} and pg_{04}. The logic of the block labeled PG is that of Equations (12.1) and (12.3) and is shown in Figure 12.2. We then recursively subdivide each of these intervals until we bottom out at the single-bit p and g terms.

Once we have generated the pg signals recursively we use these signals to generate the carries. We proceed building a tree starting with the carry across all eight bits, then groups of four, two, and one. At each level, we compute the carry from carries at the previous levels and the pg signals as:

$$c_j = g_{ij} \vee (c_i \wedge p_{ij}) \tag{12.6}$$

The logic in each of the C blocks of Figure 12.1 is that of Equation (12.6 and is shown in Figure 12.3.

Can factor any iterative function about carry variable
propagate, generate, and kill
carry look ahead
computing group pg, group carries
accumulating result

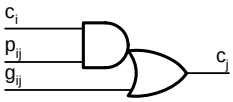


Figure 12.3: Logic to generate carry signals from upper-level carry signals and *pg* signals.

12.2 Booth Recoding

12.3 Fast Dividers

12.4 Exercises

12–1 iterative function for less-than compare - apply lookahead

12–2 Ling Adder

Chapter 14

Sequential Logic

The output of sequential logic depends not only on its input, but also on its *state* which may reflect the history of the input. We form a sequential logic circuit via feedback - feeding state variables computed by a block of combinational logic back to its input. General sequential logic, with asynchronous feedback, can become complex to design and analyze due to multiple state bits changing at different times. We simplify our design and analysis tasks in this chapter by restricting ourselves to *synchronous* sequential logic, in which the state variables are held in a register and updated on reach rising edge of a clock signal.[1]

The behavior of a synchronous sequential logic circuit or *finite-state machine* (FSM) is completely described by two logic functions: one which computes its next state as a function of its input and present state, and one that computes its output - also as a function of input and present state. We describe these two functions by means of a *state table*, or graphically with a *state diagram*. If states are specified symbolically, a state assignment maps the symbolic states onto a set of bit vectors - both binary and one-hot state assignments are commonly used.

Given a state table (or state diagram) and a state assignment, the task of implementing a finite-state machine is a simple one of synthesizing the next-state and output logic functions. For a one-hot state encoding, the synthesis is particularly simple as each state maps to a separate flip-flop and all edges in the state diagram leading to a state map into a logic function on the input of that flip-flop. For binary encodings, Karnaugh maps for each bit of the state vector are written and reduced to logic equations.

Finite-state machines are implemented in Verilog by declaring a state register to hold the current state, and describing the next-state and output functions with combinational logic descriptions, such as `case` statements as described in Chapter 7. State assignments should be specified with `` `define `` statements to allow them to be changed without altering the machine description itself. Special attention should be given to resetting the FSM.

[1] We revisit asynchronous sequential circuits in Chapter 22.

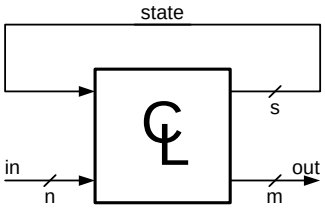


Figure 14.1: Sequential circuits are formed when feedback paths carrying state information are added to combinational circuits. The output of a sequential circuit depends on both the current input and on the state, which is a function of previous inputs.

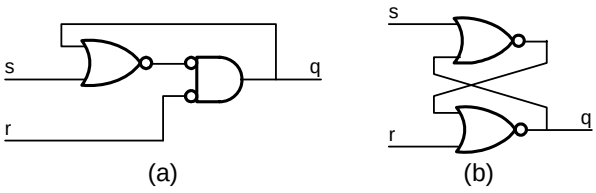


Figure 14.2: An RS flip-flop is a simple example of a sequential circuit: (a) schematic (b) alternate schematic that doesn't obey the *bubble rule*.

14.1 Sequential Circuits

Recall that a combinational circuit produces an output that depends only on the current state of its input. Recall also that combinational circuits must be acyclic. If we add feedback to a combinational circuit, creating a cycle as shown in Figure 14.1, the circuit becomes sequential. The output of a sequential circuit depends not only on its current input, but also on the history of previous inputs. The cycle created by the feedback allows the circuit to store information about its previous input. We refer to the information stored on the feedback signals as the state of the circuit.

A sequential circuit generates an output that is a function of both its input and its present state. It also generates its next state, also as a function of its input and its present state.

Figure 14.2 shows a reset-set (RS) flip-flop, a very simple sequential logic circuit that is composed of two NOR gates.[2] The output q is fed back to the input as a state variable. The circuit's behavior is described by the equation: $q = \overline{r} \wedge (s \vee q)$. The state variable q appears on both sides of the equation. To make the dynamics clearer we rewrite this as $q_{\text{new}} = \overline{r} \wedge (s \vee q_{\text{old}})$. That is, the

[2]We draw the circuit as shown in Figure 14.2(a). Many people (who do not obey the bubble rule) draw it as shown in Figure 14.2(b).

r	s	q_{old}	q_{new}
0	0	0	0
0	0	1	1
0	1	X	1
1	X	X	0

Table 14.1: State table for RS flip-flop.

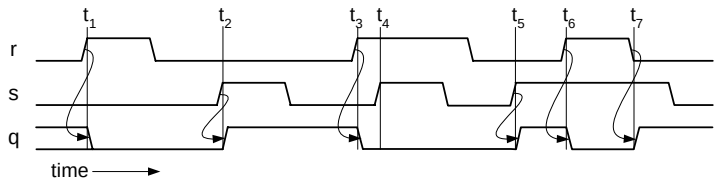


Figure 14.3: Timing diagram showing operation of the RS flip-flop. The value of signals is shown as time advances from left to right.

equation tells us how to derive the *new* state of q as a function of the inputs and the *old* state of q.

From the equation (and the schematic) it is easy to see that if $r = 1$, $q = 0$ and the flip-flop is reset, if $s = 1$ and $r = 0$, $q = 1$ and the flip-flop is set, and if $s = 0$ and $r = 0$, the output q stays in whatever state it was in. The output q reflects the last input to be high. If r was high last, $q = 0$. If s was high last, $q = 1$. We summarize this behavior in the state table of Table 14.1.

Because the function of sequential circuits depends on the evolution of signals over time, we often describe their behavior using a *timing diagram*. A timing diagram illustrating operation of the RS flip-flop is shown in Figure 14.3. The figure shows the waveforms, signal level as a function of time, for the signals r, s, and q. Time advances from left to right. Arrows from one signal to another show cause and effect.

Initially, q is in an unknown state, it could be either high or low, denoted by both high and low lines. At time t_1, r goes high causing q to fall - resetting the flip-flop. Signal s goes high at t_2 causing q to rise - setting the filp-flop The flip-flop is reset again at t_3. Signal s goes high at t_4, but this has no effect on the output, since r is also high. Signal s going high with r low at t_5 does set the flip-flop. It is reset again at t_6 when r goes high - even though s is still high. The flip-flop is set for a final time at t_7 when r goes low.

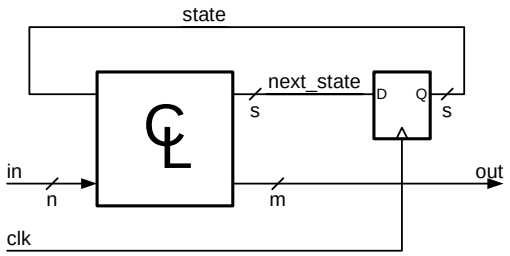


Figure 14.4: A *synchronous* sequential circuit breaks the state feedback loop with a clocked storage element (in this case a D-type flip-flop). The flip-flop ensures that all state variables change value at the same time - when the clock signal rises.

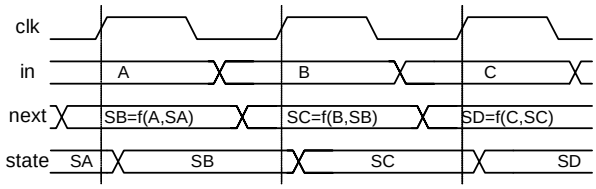


Figure 14.5: Timing diagram showing operation of a synchronous sequential circuit. The state advances on each rising edge of the clock signal, clk.

14.2 Synchronous Sequential Circuits

While the RS flip-flop of Figure 14.2 is simple enough to understand, arbitrary sequential circuits, with many bits of state feedback, can give complex behavior. Part of the complexity is due to the fact that the different bits of the next-state signal may change at different times. Such *races* can lead to a next-state output that depends on circuit delay. We defer discussion of general *asynchronous* sequential circuits until Chapter 22. Until then, we restrict our attention to *synchronous* sequential circuits in which clocked storage elements are used to ensure that all state variables change state at the same time - synchronized to a clock signal. Synchronous sequential circuits are sometimes called *finite-state machines* or FSMs.

A block diagram of a synchronous sequential logic circuit is shown in Figure 14.4. The circuit is synchronous because the state feedback loop is broken by an s-bit wide D flip-flop (where s is the number of state bits). This flip-flop circuit, described in the next section, updates its output with the value of its input on the rising edge of the clock signal. At all other times the output remains stable. Inserting the D flip-flop into the feedback loop constrains all of the state bits to change at the same time - eliminating the possiblity of races.

state	next state		out	
	in=0	in=1	in=0	in=1
00	00	01	0	0
01	00	11	0	0
11	01	10	0	0
10	11	00	0	1

Table 14.2: State table for example synchronous logic circuit.

cycle	state	in	out
0	00	0	0
1	00	1	0
2	01	1	0
3	11	0	0
4	01	1	0
5	11	1	0
6	10	0	0
7	11	1	0
8	10	1	1
9	00		0

Table 14.3: State sequence for the sequential logic circuit described by Table 14.2 on the input sequence 011011011.

Operation of a synchronous sequential circuit is illustrated in the timing diagram of Figure 14.5. During each clock cycle, the time from one rising edge of the clock to the next, a block of combinational logic computes the next state and output (not shown) as a combinational function of the input and current state. At each rising edge of the clock the current state (state) is updated with the next state computed during the previous clock cycle.

During the first clock cycle of Figure 14.5 for example, the current state is SB and the input changes to B before the end of the cycle. The combinational logic then computes the next state $SC = f(B, SB)$. At the end of the cycle the clock rises again updating the current state to be SC. The state will be SC until the clock rises again.

We can analyze synchronous sequential circuits on a clock-by-clock basis. The next state and output during the a given clock cycle depend only on the current state and input during that clock cycle. At each clock, the current state advances to the next state.

For example, suppose our next state and output logic are as given by Table 14.2. If our circuit starts in state 00 and has an input sequence of 011011011 for the first 9 cycles, what will its state and output be each cycle?

Operation of our example circuit is shown in Table 14.3. In cycle 0 we start

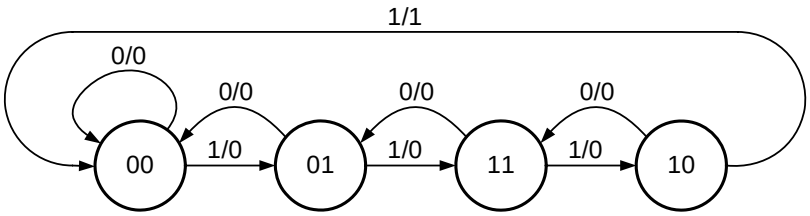


Figure 14.6: State diagram for finite-state machine of Table 14.2. The circles represent the four states. Each arrow represents a *state transition* from a current state to a next state and is labeled *input*/em output - the input that causes the transition, and the output in the current state for that input.

in state 00. The input and output are both 0 this cycle. With an input of 0 in state 00, the next state is also 00, so in cycle 1 we remain in state 00, but now with an input of 1. A state of 00 and an input of 1 gives us a next state of 01 for cycle 2. With the input high in state 01, we get a next state of 11 for cycle 3. The input goes low in state 11 in cycle 3, taking us back to state 01 for cycle 4. The input is high for the next two cycles taking us to 11 and 10 in cycles 5 and 6 respectively. A low input in cycle 6 takes us back to 11 for cycle 7. High inputs for cycles 7 and 8 takes us to states 10 and 00 for cycles 8 and 9. The output goes high in cycle 8 since the state is 10 and the input is 1.

One representation of a finite-state machine is a *state table* such as Table 14.2 that gives the next-state and output functions in tabular form. An equivalent graphical representation is a state diagram as shown in Figure 14.6.

Each circle in Figure 14.6 represents one state. Its labeled with the name of the state. Here we use the value of the state variables as the state name. Later we will introduce symbolic state names, independent of the state coding. The next state function is shown by the arrows. Each arrow represents a *state transition* and is labeled with the input and output values during that transition. For example, the arrow from state 00 to state 01, labeled 1/0, implies that in state 00 when the input is 0 the next state is 01 and the output is 0. Note that an arrow may go from a state to itself, as in the case of the input 0 transition from state 00 to state 00. Also, a transition may go quite a distance in the diagram, as with the input 1 transition from state 10 to state 00.

14.3 Traffic Light Controller

As a second example of a FSM, consider the problem of controlling the traffic lights at an intersection of a north-south road with an east-west road. There are six lights to control green, yellow, and red for the north-south road, and for the east-west road. Our FSM will take as input signal carew that indicates that a car is waiting on the east-west road. A second input, rst, resets the FSM to

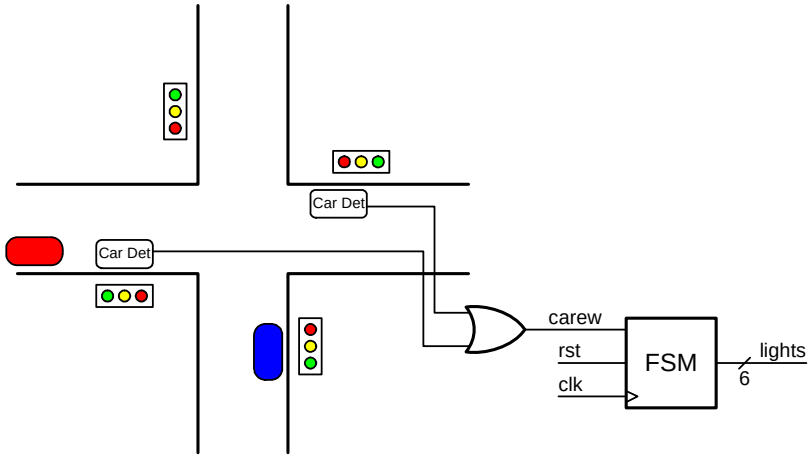


Figure 14.7: Controlling traffic lights at a road intersection with a FSM. Our FSM has two inputs, a reset (rst), and a signal that indicates that a car is waiting on the east-west road (carew). The FSM has six outputs that control the three north-south lights (green, yellow, red), and the three east-west lights.

a known state.

We start with an English-language description of our FSM:

1. Reset to a state where the light is green in the north-south direction and red in the east-west direction.
2. When a car is detected in the east-west direction (carew = 1) go through a sequence that makes the light green in the east-west direction and then return to green in the north-south direction.
3. A direction with a green light must first transition to a state where the light is yellow before going to a state where the light goes red.
4. A direction can have a green light only if the light in the other direction is red.

A state diagram for a FSM that meets our specification is shown in Figure 14.8. Compared to the state diagram of Figure 14.6 there are two major differences First, the states are labeled with symbolic names. the output values are placed under the states rather than on the transitions. This is because the output is a function only of the state, not of the input. Second, the output values are placed under the states rather than on the transitions. This is because the output is a function only of the state, not of the input.[3]

[3] A FSM where the output depends only on the current state and not the input is sometimes called a *Moore machine*. A FSM where the output depends on both the current state and the input is sometimes called a *Mealy machine*.

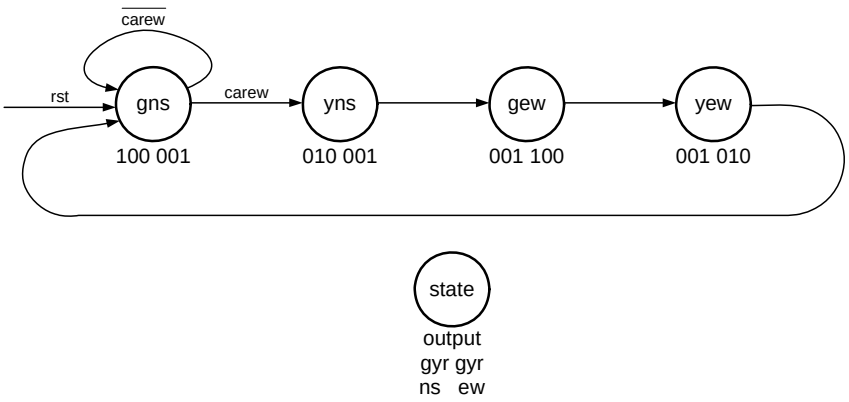


Figure 14.8: State diagram for a traffic-light controller FSM. The states are labeled with symbolic names. The outputs are given under each state (green-yellow-red (3-bits) for north-south followed by green-yellow-red for east-west). The reset arrows are omitted. The FSM resets to state gns.

The FSM resets to state gns (for green-north-south). In this state the output is 100 001. The first 100 represents the north-south lights (green-yellow-red). The 001 represents the east-west lights (also green-yellow-red). Hence the light is green in the north-south direction and red in the east-west direction. Resetting to this state satisfies specification number 1. The arrow labeled $\overline{\text{carew}}$ keeps us in state gns until a car is detected in the east-west direction.

When a car is detected in the east-west direction, signal carew becomes true, and the next rising edge of the clock causes the machine to enter state yns (yellow north-south). In this state the output is 010 001. The 010 implies yellow in the north-south direction, and 001 implies red in the east-west direction. Transitioning to this state before makign east-west green satisfies specification 3 on the transition from gns to gew. The arrow out of state yns has no label. This implies that this state transition always occurs (unless the FSM is reset).

State yns is always followed by state gew (green east-west). In this state the output is 001 100 - red in the north-south direction and green in the east-west direction. This state, and the sequence it is part of, satisfies specification 2. State gew is always followed by state yew. State yew (yellow east west) has output 001 010 - red in the north-south direction and yellow in the east-west direction. This state statisfies specification 3 on the transition between gew and gns.

A state table for the traffic-light controller is shown in Table 14.4. Reset is not shown.

state	next state		out
	carew=0	carew=1	
gns	gns	yns	100 001
yns	gew	gew	010 001
gew	yew	yew	001 100
yew	gns	gns	001 010

Table 14.4: State table for the traffic light controller FSM. The FSM resets to state gns.

state	encoding
gns	0001
yns	0010
gew	0100
yew	1000

Table 14.5: A one-hot state assignment for the traffic light controller FSM.

14.4 State Assignment

When a FSM is specified with symbolic state names as in Figure 14.8 or Table 14.4 we need to assign actual binary values to the states before we can implement the FSM. This process of assigning values to states is called *state assignment*.

With a synchronous machine, we can assign the states to any set of values, as long as the value for each state is unique. [4] It takes at least $s_{\min} = \log_2(N)$ bits to represent N states; however, the best state assignment is not always one with the fewest bits. We refer to each bit of a state vector as a *state variable*.

A *one-hot* state assignment uses N bits to represent N states. Each state gets its own bit. When the machine is in the i^{th} state, the corresponding bit b_i of the state variable is 1. In all other states, b_i is 0. A one-hot state assignment for the traffic-light controller FSM is shown in Table 14.5. It takes four bits to represent the four states. In any state, only one bit is set. A one-hot assignment makes the logic design of a finite state machine particularly simple as we shall see below.

A binary state assignment uses the minimum number of bits, $s_{\min} \log_2(N)$, to represent N states. Of the $N!$ possible binary state assignments (24 for 4 states), it doesn't really matter which one you choose. While lots of academic papers have been written about choosing state assignments to minimize implementation logic, in practice it doesn't really matter. Except in very rare cases the number of gates saved by optimizing the state assignment is unimportant. Don't waste

[4] This is not true of an asynchronous machine where careful state assignment is required to avoid races.

state	encoding
gns	00
yns	01
gew	11
yew	10

Table 14.6: A binary state assignment for the traffic light controller FSM.

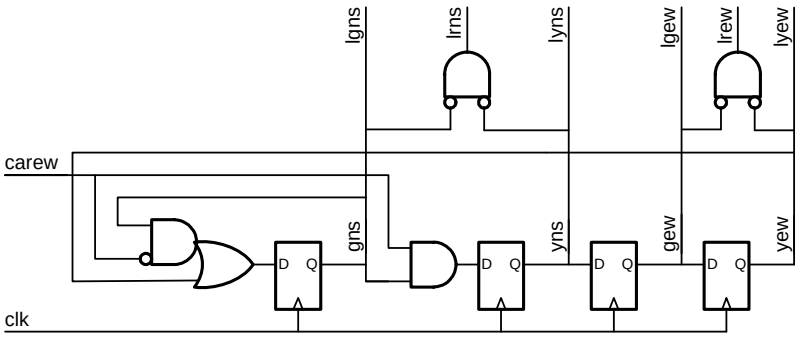


Figure 14.9: Implementation of the traffic light controller FSM using a one-hot state encoding. Four flip-flops are used, one corresponding to each state. The state transition arrows into a state directly translate into the logic preceding the corresponding flip-flop.

much time on state assignment. Design time is more important than a few gates.

One possible binary state assignment for the traffic light controller FSM is shown in Table 14.6. This particular state assignment uses a Gray code so that only one-bit of state changes on each state transition. This sometimes reduces power and minimizes logic. We could just as easily have chosen a straight binary count (gew = 10, yew = 11) and it wouldn't make much difference.

14.5 Implementation of Finite State Machines

Given a state table (or state diagram) and a state assignment the implementation of a FSM is reduced the problem of designing two combinational logic circuits, one for the next-state and one for the output. These combinational logic circuits are combined with a s-bit wide D-flip-flop to update the current state from the next state on each rising edge of the clock. A multi-bit D-flip-flop like this is often called a *register* and when it is used to hold the state of an FSM it is called the *state register*.

With a one-hot state assignment, the implementation of the next-state logic

state	carew	next state	comment
00	0	00	green north/south carew = 0
00	1	01	green north/south carew = 1
01	0	11	yellow north/south carew = 0
01	1	11	yellow north/south carew = 1
11	0	10	green east/west carew = 0
11	1	10	green east/west carew = 1
10	0	00	yellow east/west carew = 0
10	1	00	yellow east/west carew = 1

Table 14.7: Truth table for the next state function of the traffic light controller FSM with the state assignment of Table 14.6.

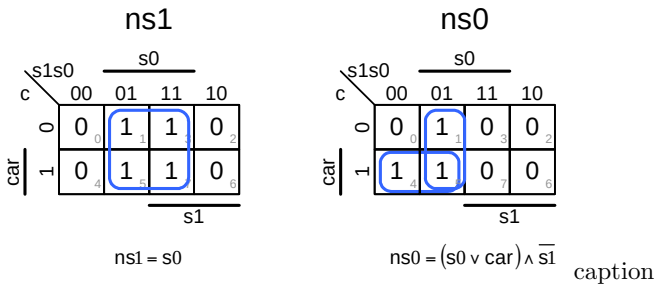
is a direct translation of the state diagram as shown in Figure 14.9 for our traffic light controller FSM. Four flip-flops correspond to the four states: gns, yns, gew, and yew. When the first flip flop is set, the FSM is in state gns. The logic feeding the D input of each flip-flop is an OR of the transition arrows feeding the corresponding state in the state diagram. For states gew and yew this is just a wire. These states always follow the preceding state. For state yns, the input logic is an AND gate that ANDs the previous state (gns) with the condition for a transition from gns to yns (carew). State gns is the target of two destination arrows and hence requires an OR gate to combine them. Its input logic is the OR of a wire from state yew, and an AND gate combining state gns and condition $\overline{\text{carew}}$ - this corresponds to the back edge from state gns to itself.

It is always possible to directly implement a one-hot FSM in this manner. This made design and maintenance of FSMs very easy in the era before logic synthesis. One would simply instantiate a flip-flop for each state, and approprate input gates for each transition arrow. The function is immediately apparent from the logic. Adding, deleting, or changing the condition on a transition arrow was straightforward and affected only only the part of the logic associated with the transition arrow. With modern logic synthesis, the advantage of this approach is greatly diminished.

The output logic for the circuit of Figure 14.9 consists of two NOR gates. States gns, yns, gew, and yew drive the green and yellow light outputs directly. The red light outputs are generated by observing that for each direction, if the yellow and green lights are off, the red light should be on: $r = \overline{y} \vee \overline{g}$.

To implement our FSM with a binary state encoding, we proceed with logic synthesis of each of the state variables. First, we convert our state table into a truth table - showing each next state variables as a function of the current state variables and all inputs. For example, a truth table for the traffic light controller FSM with the state encoding of Table 14.6 is shown in Table 14.7.

From this state table we draw the two Kargaugh maps shown in Figure 14.5. The left K-map shows the truth-table for the MSB of the next-state (ns1) and



caption

state	output
00	100 001
01	010 001
11	001 100
10	001 010

Table 14.8: Truth table for the output function of the traffic light controller FSM with the state assignment of Table 14.6.

the right K-map shows the truth table for the LSB of the next-stage (ns0). The next state logic here is very simple. The ns1 function has a single prime implicant, and the ns0 function has 2. All three are essential. The logic here is very simple:

$$n_1 = s_0 \tag{14.1}$$
$$n_0 = (s_0 \vee \text{carew}) \wedge \overline{s_1} \tag{14.2}$$

Where n_1, n_0 are the next state variables and s_1, s_0 are the current state variables.

Now that we have the next-state function, what remains is to derive the output function. To do this we write down the truth table for the outputs - a function of only the current state. This is shown in Table 14.8. The logic functions for the output variables can be derived directly from this table - a K-map is not necessary, we get:

$$g_{\text{ns}} = \overline{s_1} \wedge \overline{s_0} \tag{14.3}$$
$$y_{\text{ns}} = \overline{s_1} \wedge s_0 \tag{14.4}$$
$$r_{\text{ns}} = s_1 \tag{14.5}$$
$$g_{\text{ew}} = s_1 \wedge s_0 \tag{14.6}$$
$$y_{\text{ew}} = s_1 \wedge \overline{s_0} \tag{14.7}$$
$$r_{\text{ew}} = \overline{s_1} \tag{14.8}$$

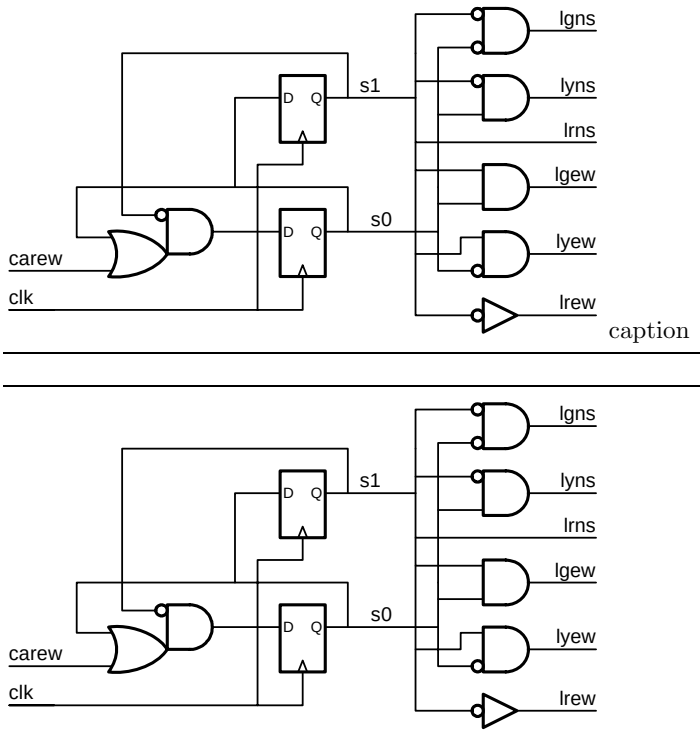


Figure 14.10: Logic diagram for traffic light controller FSM implemented with state assignmet of Table 14.6.

Combining the next-state and logic equations we get the logic diagram of Figure 14.10.

14.6 Verilog Implementation of Finite State Machines

With Verilog, designing a FSM is a simple matter of specifying the next-state and output functions and selecting a state assignment. Logic synthesis does all the work of generating the next-state and output logic. A verilog description of the traffic light controller FSM is shown in Figure 14.11. The bulk of the logic is in a single `case` statement that defines both the next-state and output functions.

There are three key points to be made about this code.

1. In implementing all sequential logic, all state variables should be explicitly

```
//-------------------------------------------------------
// Traffic_Light
// Inputs:
//   clk - system clock
//   rst - reset - high true
//   carew - car east/west - true when car is waiting in east-west direction
// Outputs:
//   lights - (6 bits) {gns, yns, rns, gew, yew, rew}
// Waits in state GNS until carew is true, then sequences YNS, GEW, YEW
// and back to GNS.
//-------------------------------------------------------
module Traffic_Light(clk, rst, carew, lights) ;
  input clk ;
  input rst ;            // reset
  input carew ;          // car present on east-west road
  output [5:0] lights ;  // {gns, yns, rns, gew, yew, rew}
  wire [`SWIDTH-1:0] state, next ; // current and next state
  reg [`SWIDTH-1:0] next1 ;        // next state w/o reset
  reg [5:0] lights ;               // output - six lights 1=on

  // instantiate state register
  DFF #(`SWIDTH) state_reg(clk, next, state) ;

  // next state and output equations - this is combinational logic
  always @(state or carew) begin
    case(state)
      `GNS: {next1, lights} = {(carew ? `YNS : `GNS), `GNSL} ;
      `YNS: {next1, lights} = {`GEW, `YNSL} ;
      `GEW: {next1, lights} = {`YEW, `GEWL} ;
      `YEW: {next1, lights} = {`GNS, `YEWL} ;
    endcase
  end
  // add reset
  assign next = rst ? `GNS : next1 ;
endmodule
```

Figure 14.11: Verilog description of traffic light controller FSM.

declared as D-flip-flops. **DO NOT** let the Verilog compiler infer flip-flops for you. In this code, the state flip-flops are explicitly instantiated in the code:

```
// instantiate state register
DFF #(`SWIDTH) state_reg(clk, next, state) ;
```

This code instantiates an `` `SWIDTH ``-wide D-flip-flop that is clocked by `clk`, has input `next` and output `state`.

2. In designing finite state machines, use Verilog `` `define `` statements to define all constants. Do not hard-code any constants in the code. Constants that should be declared this way include the width of the state vector `` `SWIDTH ``, the state encodings (e.g., `` `GNS ``), and input and output encodings (e.g., `` `GNSL ``). In particular defining symbolic names for the state encodings enables you to change a state assignment by just changing the definitions. We'll see an example of this below.

3. Make sure you reset your FSM. Here we declare two next-state vectors, `next1`, and `next`. The `case` statement computes `next1` as the next state ignoring reset `rst`. A final assign statement overrides this next state with the reset state `` `GNS `` if `rst` is asserted:

   ```
   // add reset
   assign next = rst ? `GNS : next1 ;
   ```

   Factoring the reset out of the next state function in this manner greatly improves readability of the code. If we didn't do this, we'd have to repeat the reset logic in every state - rather than doing it just once.

The Verilog definitions for the traffic light controller FSM are shown in Figure 14.12. Using `` `define `` in this manner enables us to use symbolic names in our code, improving readability, and also makes it easy to change encodings. For example, substituting the one-hot state encodings of Figure 14.13 changes our FSM from a binary to a one-hot state assignment without changing any other lines of code.

The Verilog code for the `DFF` module is shown in Figure 14.14. The behavior is described by the `always` block:

```
always @(posedge clk)
  out = in ;
```

This block performs the update of the output `out = in` on every rising edge (`posedge`) of `clk`.

Our test bench for the traffic-light controller is shown in Figure 14.15. To thoroughly test an FSM, we would like to both visit every state, and traverse every edge of the state diagram. Achieving this coverage is not particularly difficult for our traffic light controller FSM.

```
//-------------------------------------------
// define state assignment - binary
//-------------------------------------------
‘define SWIDTH 2
‘define GNS 2’b00
‘define YNS 2’b01
‘define GEW 2’b11
‘define YEW 2’b10
//-------------------------------------------
// define output codes
//-------------------------------------------
‘define GNSL 6’b100001
‘define YNSL 6’b010001
‘define GEWL 6’b001100
‘define YEWL 6’b001010
```

Figure 14.12: Verilog definitions for traffic-light controller state variables and output encodings.

```
//-------------------------------------------------------------------------
// define state assignment - one hot
//-------------------------------------------------------------------------
‘define SWIDTH 4
‘define GNS 4’b1000
‘define YNS 4’b0100
‘define GEW 4’b0010
‘define YEW 4’b0001
```

Figure 14.13: Verilog definitions for a one-hot state assignment for the traffic light controller FSM.

```
module DFF(clk, in, out) ;
  parameter n = 1;  // width
  input clk ;
  input  [n-1:0] in ;
  output [n-1:0] out ;
  reg    [n-1:0] out ;

  always @(posedge clk)
    out = in ;
endmodule
```

Figure 14.14: Verilog description of a D-flip-flop.

```
module Test_Fsm1 ;
  reg clk, rst, carew ;
  wire [5:0] lights ;

  Traffic_Light tl(clk, rst, carew, lights) ;

  // clock with period of 10 units
  initial begin
    clk = 1 ; #5 clk = 0 ;
    forever
      begin
        $display("%b %b %b %b", rst, carew, tl.state, lights ) ;
        #5 clk = 1 ; #5 clk = 0 ;
      end
    end

  // input stimuli
  initial begin
    rst = 0 ; carew=0 ;        // start w/o reset to show x state
    #15 rst = 1 ; carew = 0 ;  // reset
    #10 rst = 0 ;              // remove reset
    #20 carew = 1 ;            // wait 2 cycles, then car arrives
    #30 carew = 0 ;            // car leaves after 3 cycles (green)
    #20 carew = 1 ;            // wait 2 cycles then car comes and stays
    #60                        // 6 more cycles
    $stop ;
  end
endmodule
```

Figure 14.15: Verilog test bench for the traffic light controller FSM.

```
0 0 xx xxxxxx
1 0 xx xxxxxx
0 0 00 100001
0 0 00 100001
0 1 00 100001
0 1 01 010001
0 1 11 001100
0 0 10 001010
0 0 00 100001
0 1 00 100001
0 1 01 010001
0 1 11 001100
0 1 10 001010
0 1 00 100001
0 1 01 010001
```

Figure 14.16: Results of simulating the traffic light controller FSM of Figure 14.11 using the test bench of Figure 14.15. Each line shows the values of `rst`, `carew`, `state`, and `lights` on a falling edge of the clock.

The test bench has three components. First, it instantiates a `Traffic_Light` module - the unit being tested. The second component is an `initial` block that performs clock generation and output. The `forever` block repeats its body until the simulation terminates. In this case the repeated code displays some variables and generates a clock with a 10 delay-unit period. The display is done in the middle of the clock cycle - just after `clk` goes low. The final component generates the inputs for the module being tested.

The inputs, and the response of the module are shown in textual form in Figure 14.16, which shows the signals on each falling edge of the clock, and as waveforms in Figure **??**. Initially, `state` and `next` are both in an unknown state (`x` in the text output, and a line midway between a 1 and 0 in the waveform). Signal `rst` is asserted in the second clock cycle to reset to a known state. Next-state signal `next` responds immediately, and `state` follows on the rising edge of the clock. The FSM stays in state 00 until `carew` rises on clock 5, causing the FSM to go to state 01 on clock 6. This starts a sequence through states 01, 11, 10, and back to 00 on clock 9. It stays in state 00 for two cycles this time, until `carew` going high in clock 10 causes the FSM to start the sequence again in clock 11. This time `carew` stays high and the sequence repeats until the simulation ends.

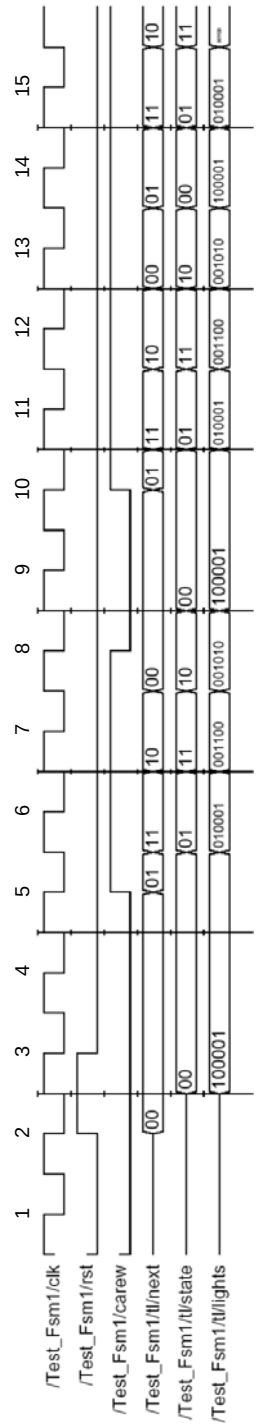


Figure 14.17: Waveforms from simulation of the traffic light controller of Figure 14.11 using test bench of Figure 14.15.

14.7 Bibliographic Notes

14.8 Exercises

14–1 *Homing sequences.* The finite-state machine described by Table 14.2 does not have a reset input. Explain how you can get the machine in a known state regardless of its initial starting state by providing a fixed input sequence. An input sequence that always takes an FSM to the same state is called a *homing sequence.*

14–2 *Homing sequences.* Suppose the traffic light controller FSM of Table 14.4 did not reset to state gns. Find a homing sequence for the machine that will get it to state gns.

14–3 *Finite State Machines.* Modify the traffic light controller FSM of Table 14.4 so that it makes lights go red in both directions for one cycle before turning a light green. Show a state table and state diagram for your new FSM.

14–4 *Finite State Machines.* Modify the traffic light controller FSM of Table 14.4 so that it takes an additional input, carns, that indicates when there is a car waiting in the north-south direction. Change the logic so that once the light changes to east-west, it stays with east-west green until a car is detected waiting in the north-south direction. Show a state table and state diagram for your new FSM.

14–5 *FSM Implementation.* Implement the traffic light controller FSM with a state encoding where gns = 00, yns = 01, gew = 10, and yew = 11. Show Karnaugh maps for the next state variables and output variables and a gate-level schematic for the FSM.

14–6 *A Digital Lock* Draw a state diagram and a state table for a digital lock. The lock has two inputs *a* and *b* and one output, *unlock*. The output is assered only if the sequence *a, b, a, a* is observed. Each element of the sequence must last for one or more cycles, and there may be zero or more cycles of both inputs low between the sequence elements. After unlocking, either input going high causes unlock to go low.

14–7 *Anti-Lock Brakes*A FSM for an anti-lock brake system accepts two inputs - *wheel* and *time*, and generates a single output - *unlock*. The *wheel* input pulses high for one clock cycle each time the wheel rotates a small amount. The *time* input pulses high for one clock cycle every 10ms. If the machine detects two *time* pulses since the last *wheel* pulse, it concludes that the wheel is locked and *unlock* is asserted for one clock cycle to "pump" the brakes. After *unlock* goes high, the machine waits for two *time* pulses before resuming normal operation. Thus, there are a minimum of four *time* pulses between *unlock* pulses. Draw a state diagram (bubble diagram) for this state machine.

14–8 *Direction Sensor.* Draw a state diagram and state table for a direction sensor machine. This machine has two inputs *il* and *ir* and two outputs *ol* and *or*. The machine outputs a one cycle pulse on *ol* any time a high level on *il* for one or more cycles is followed, after zero or more cycles, by a high level on *ir* for zero or more cycles. Similarly a one cycle pulse on *or* is output if a high level on *ir* is followed by a high level on *il*.

Chapter 15

Timing Constraints

How fast will a FSM run? Could making our logic too fast cause our FSM to fail? In this chapter, we will see how to answer these questions by analyzing the timing of our finite state machines and the flip-flops used to build them.

Finite state machines are governed by two timing constraints - a maximum delay constraint and a minimum delay constraint. The maximum speed at which we can operate a FSM depends on two flip-flop parameters (the setup time and propagation delay) along with the maximum propagation delay of the next-state logic. On the other hand, the minimum delay constraint depends on the other two flip-flop parameters (hold time and contamination delay) and the minimum contamination delay of the next-state logic. We will see that if the minimum delay constraint is not met, our FSM may indeed fail to operate at any clock speed due to hold-time violations. Clock skew, the delay between the clocks arriving at different flip-flops, affects both maximum and minimum delay constraints.

15.1 Propagation and Contamination Delay

In a synchronous system, logic signals advance from the stable state at the end of one clock cycle to a new stable state at the end of the next clock cycle. Between these two stable states, they may go through an arbitrary number of transitions.

In analyzing timing of a logic block we are concerned with two times. First, we would like to know how long the output retains its initial stable value (from the last clock cycle) after an input first changes (in the new clock cycle). We refer to this time as the *contamination delay* of the block- the time it takes for the old stable value to become contaminated by an input transition. Note that this first change in the output value does not in general leave the output in its new stable state.

The second time we would like to know is how long it takes the output to reach its new stable state after the input stops changing. We refer to this time

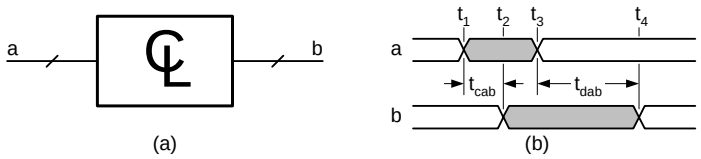


Figure 15.1: Propagation delay t_{dab} and contamination delay t_{cab}. The *contamination delay* of a logic block is the time from when the **first** input signal **first** changes to when the **first** output signal **first** changes. The *propagation delay* of a logic block is the time from when the **last** input signal **last** changes to when the **last** output signal **last** changes.

as the *propagation delay* of the block - the time it takes for the stable value of the input to propagate to a stable value at the output.

Propagation delay and contamination delay are illustrated in Figure 15.1. Figure 15.1(a) shows a combinational logic block with input a and output b. Figure 15.1(b) shows how the output b responds when input a changes state. Up to time t_1, both input a and output b are in their stable state from the last clock cycle. At time t_1 input a first changes. If a is a multi-bit signal, this is the time when the first bit of a to change state toggles - other bits may change at later times. Whether single-bit or multi-bit, t_1 is the time of the first transition on a. A given bit of a may toggle more than once before reaching its new stable state. At time t_2, a contamination delay t_{dab} after t_1 this first change on a may affect output b, and b may change state. Up until t_2, output b was guaranteed to have the steady-state value from the previous clock cycle. The first bit of b to change toggles for the first time at time t_2 as with a at t_1, this bit of b may toggle again before reaching a steady state, and other bits of b may change state later.

At time t_3 input a stops changing state. From t_3 until at least the end of the current clock cycle, signal a is guaranteed to be in its stable state for this clock cycle. Time t_3 represents the time at which the last bit of a to toggle toggles for the last time. At time t_4, a propagation delay t_{dab} after t_3, the last change of input a has its final affect on output b. From this point to at least the end of the clock cycle, output b is guaranteed to be in its stable state for this clock cycle.

We denote a propagation (contamination) delay from a signal a to a signal b as t_{dab} (t_{cab}). The d or c in the subscript denotes propagation or contamination. The rest of the subscript gives the source signal and destination signal of the delay. That is t_{dxy} is the delay starting with a transition on signal x to a transition on signal y.

Propagation and contamination delays sum up over linear paths as shown in Figure 15.2. The timing diagram in the figure shows that when two modules are composed in series, their delays sum:

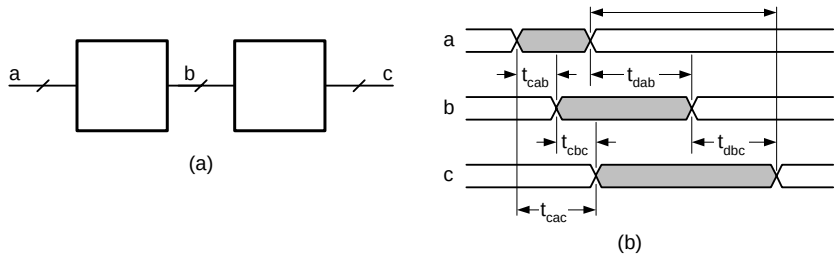


Figure 15.2: Propagation and contamination delay sum over a linear path. (a) Two modules in series with input a, intermediate signal b, and output c. (b) Timing diagram showing that $t_{cac} = t_{cab} + t_{cbc}$ and similarly for propagation delay.

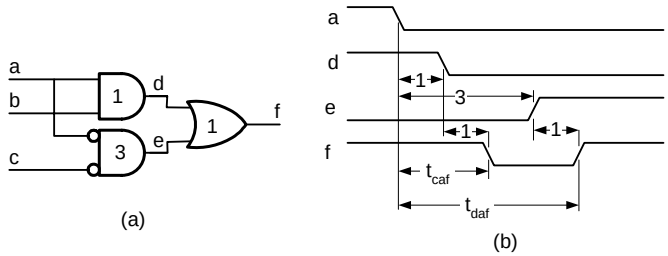


Figure 15.3: A circuit with a hazard illustrating propagation and contamination delay.

$$t_{cac} = t_{cab} + t_{cac}, \tag{15.1}$$

$$t_{dac} = t_{dab} + t_{dac}. \tag{15.2}$$

To handle circuits with parallel paths, we simply enumerate all possible single-bit paths. The overall contamination delay is the *minimum* over all paths, and the overall propagation delay is the *maximum* over all paths.

Figure 15.3(a) shows a circuit with a static-1 hazard (recall Section 6.10). The value in each gate symbol is the delay of the gate in arbitrary time units. (Here we assume the contamination and propagation delay of the basic gates are the same). The timing diagram in Figure 15.3(b) illustrates the timing when signal a falls while $b = 1$ and $c = 0$. The output changes for the first time after two time units and for the last time after four time units. Hence, $t_{caf} = 2$ and $t_{daf} = 4$.

We can get the same result by enumerating paths. The minimum delay path is a-d-f with a contamination delay of 2 while the maximum path is a-e-f with

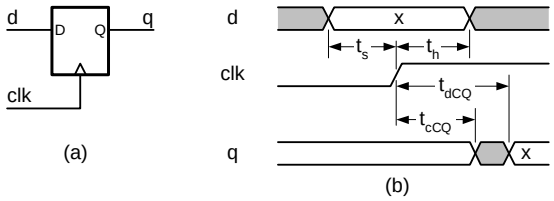


Figure 15.4: A D-flip-flop. (a) Schematic symbol, (b) Timing diagram. The D-flip-flop samples its input on the rising edge of the clock and updates the output with the value sampled. For correct sampling the input must be *stable* from t_s before the clock rises to t_h after the clock rises. The output may change as soon as t_{cCQ} after the clock. The output takes on the correct value no later than t_{pCQ} after the clock.

a propagation delay of 4.

Contamination and propagation delay are independent of input state. The contamination delay of the circuit in Figure 15.3(a) from a to f is 2 regardless of the state of signals b and c. This delay represents the possibility that the output *may* change 2 time units after a transition on a, not a guarantee that it will change.

May people confuse contamination delay with minimum propagation delay. They are not the same thing. The minimum propagation delay is the minimum value (over some range of parameters: voltage, temperature, process variation, input combinations) of the time for the correct steady-state value to appear on the output of a circuit after a transition on the input. In contrast, the contamination delay is the time until the output first changes from its old steady state value after a transition on its input. These are not the same thing. The transition that sets the contamination delay is not in general a change of the output to its steady state value, but rather a change to some intermediate value - for example the transition of a to 0 in the hazard of Figure 15.3.

15.2 The D Flip-Flop

The timing constraints that determine whether an FSM will operate or not, and at what speed it will operate, are governed by the clocked storage elements used to construct the FSM - in our case, the D flip-flop. A schematic symbol for a D-flip-flop is shown in Figure 15.4. A multi-bit D-flip-flop is sometimes called a *register*.

A D flip-flop samples its input on the rising edge of the clock signal and updates its output with the value sampled. This sampling and update is illustrated in the timing diagram of Figure 15.4(b). For the sampling to take place correctly, the input data (shown in the top waveform of the timing diagram) must be stable for a period before and after the rising edge of the clock. Specifically,

the data must have reached its correct value (labeled x in the figure) at least a *setup time* t_s before the clock reaches its 50% point, and the data must be held stable at this value until a *hold time* t_h after the clock reaches its 50% point.[1] During the gray areas in the data waveform, D can take on any value. However it must remain stable with a value of x during the setup and hold intervals for the flip-flop to correctly sample the value x.

If the input meets its setup and hold time constraints, the flip-flop will update the output with the sampled value x as shown on the bottom waveform of Figure 15.4(b). The old value (which was sampled on the previous rising edge of the clock) will remain stable on the output until a *contamination delay* t_{cCQ} after the rising edge of the clock. The circuit may continue to rely on the old value being stable up until this point in time. After the contamination delay the output of the flip-flop may change, but not necessarily to the correct value. This period where the output value is not guaranteed is labeled gray in the figure. A *propagation delay* t_{dCQ} after the rising edge of the clock, the output is guaranteed to have the value x sampled from the input. It will then hold this value stable until t_{cCQ} after the next rising edge of the clock.

15.3 Setup and Hold Time Constraint

Now that we have introduced the nomenclature, the timing constraints on a finite-state machine are quite simple. To ensure that the clock cycle $t_{\rm cy}$ is long enough for the longest path to satisfy the setup time of the D-flip-flop we must satisfy:

$$t_{\rm cy} \geq t_{dCQ} + t_{d\rm Max} + t_s. \tag{15.3}$$

Where $t_{d\rm Max}$ is the maximum propagation delay from the output of a D-flip-flop to the input of a D-flip-flop.

We also must ensure that no signal is contaminated so quickly as to violate the hold-time constraint on the input of a D-flip-flop by satisfying:

$$t_h \leq t_{cCQ} + t_{c\rm Min}. \tag{15.4}$$

Where $t_{c\rm Min}$ is the minimum contamination delay from the output of a D-flip-flop to the input of a D-flip-flop.

The two constraints (15.3) and (15.4) govern system timing. Setup-time constraint (15.3) determines performance by giving the minimum cycle time $t_{\rm cy}$ at which the circuit will operate. The hold-time constraint, on the other hand, is a correctness constraint. If (15.4) is violated, the circuit may not meet its hold time constraint regardless of the cycle time.

Figure 15.5 shows a simple finite-state machine that we shall use to illustrate setup and hold constraints below. The FSM consits of two flip-flops. The

[1] Note that t_s or t_h may be negative, but $t_s + t_h$ will always be positive.

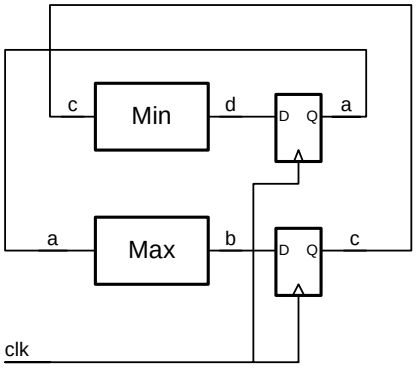


Figure 15.5: A simple FSM to illustrate setup and hold constraints

upper flip-flop generates state-bit a that propagates through a maximum-length (maximum propagation delay) logic path (Max) to generate signal b. Signal b is in turn sampled by the lower flip-flop. The lower flip-flop in turn generates signal c which propagates through a minimum-length (minimum contamination delay) block to generate signal d which is sampled by the upper flip-flop. Note that the destination flip-flop of a minimum-delay path is not necessarily the source flip-flop of a maximum-delay path (and vice versa). In general we need to test all possible paths from all flip-flops to all flip-flops to find the minimum and maximum paths.

The maximum-delay path from the upper flip-flop to the lower flip-flop of Figure 15.5 stresses the setup-time of the lower flip-flop. If this path is too slow, the next clock edge may arrive at the lower flip-flop before its input signal b has settled at its final value for the cycle. Figure 15.6 repeats Figure 15.5 with this path highlighed. A timing diagram corresponding to this path is shown in Figure 15.7. Suppose the rising edge of the clock samples value x on signal d then after a flip-flop propagation delay t_{dCQ} flip-flop output a will take on value x and hold this value through the remainder of the clock cycle. Signal a is input to combinational block Max which generates signal b. After an additional propagation delay from a to b t_{dab} (which corresponds to $t_{d\mathrm{Max}}$ in Inequality (15.3)) signal b takes on its final value for the clock cycle $f(x)$. Signal b must settle at this final value at least t_s before the rising edge of the next clock for Constraint (15.3) to be satisfied. The sum of the propagation delays along the maximum path and the setup time must be less than the cycle time. In the timing diagram, signal b settles slightly early leaving a timing margin or *slack* time of t_{slack}. The clock cycle t_{cy} could be reduced by t_{slack} and the setup constraint would still be met.

The minimum-delay path from the lower flip-flop to the upper flip-flop of Figure 15.5 stresses the hold time of the upper flip-flop. If this path is too fast, signal d might change before a hold time after the rising edge of the clock. Fig-

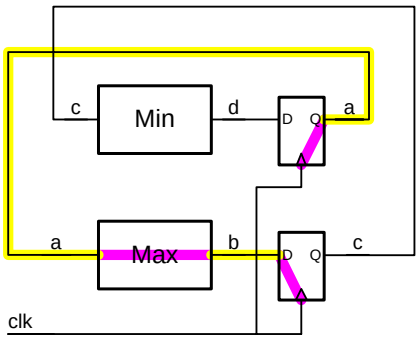


Figure 15.6: Setup-time constraint. The maximum path from the clock on a source flip-flop to the clock on a destination flip-flop is shaded. From the rising edge of the clock, the signal must propagate to the Q output of the flip-flop (t_{dCQ}) and propagate through the maximum-delay logic path (t_{dab}) at least a setup time (t_s) before the next clock edge. In this case the signal arrives slightly early (t_{slack}).

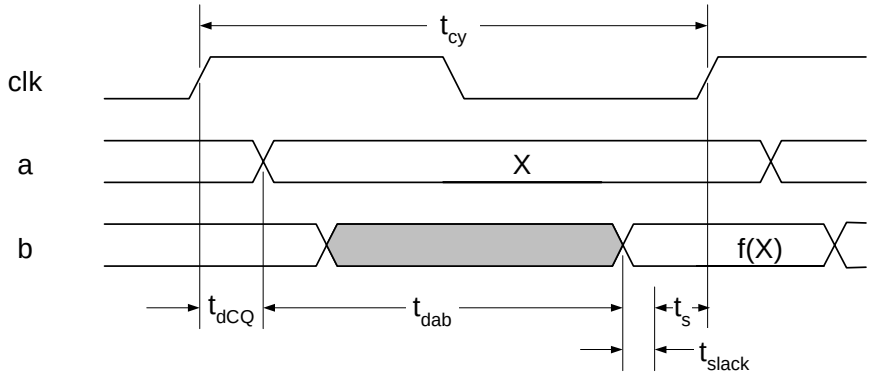


Figure 15.7: Timing diagram illustrating the setup-time constraint.

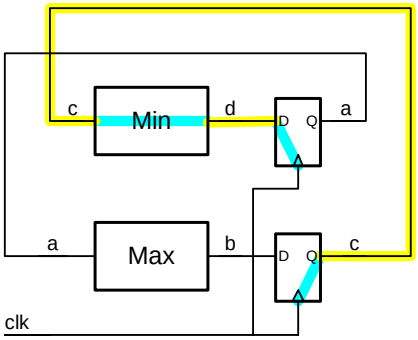


Figure 15.8:

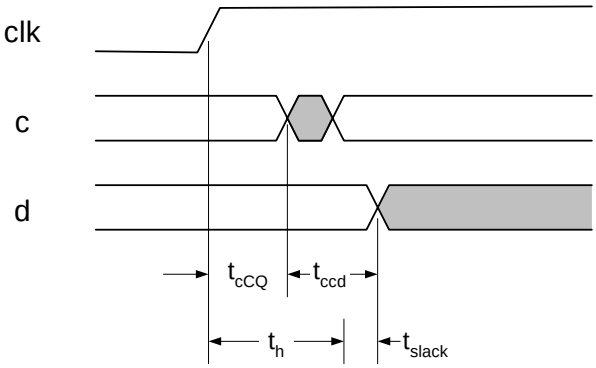


Figure 15.9:

ure 15.8 shows this timing path highlighted. A timing diagram illustrating the signals along this path is shown in Figure 15.9. A flip-flop contamination delay t_{cCQ} after the rising edge of the clock, signal c may first change. A contamination delay of the logic block t_{ccd} (which corresponds to $t_{c\mathrm{Min}}$ in Constraint (15.4)) signal d may change. For the hold time constraint to be satisfied, this first change on signal d is not allowed to occur until t_h after the rising edge of the clock. The sum of the contamination delays along the minimum path must be larger than the hold time. In the figure, the contamination delays exceed the hold time by a considerable timing margin or slack time t_{slack}.

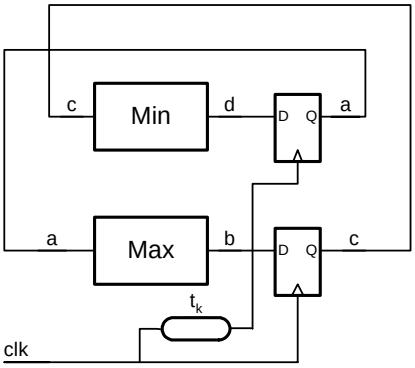


Figure 15.10: FSM of Figure 15.5 with clock skew.

15.4 The Effect of Clock Skew

On an ideal chip, the clock signal would change at the input of all flip-flops at the same time. In practice, device variations and wire delays in the clock distribution network cause the timing of the clock signal to vary slightly from flip-flop to flip-flop. We refer to this spatial variation in clock timing as *clock skew.* Clock skew adversely affects both the setup and hold timing constraints. With a skew of t_k, these two constraints become:

$$t_{\text{cy}} \geq t_{dCQ} + t_{d\text{Max}} + t_s + t_k. \tag{15.5}$$

$$t_h \leq t_{cCQ} + t_{c\text{Min}} - t_k. \tag{15.6}$$

Figure 15.10 shows the FSM of Figure 15.5 with clock skew added. A delay line (the oval shaped block) with a delay of t_k (the magnitude of the skew) is connected between the clock input and the clock to the upper flip-flop. Hence each edge of the clock arrives at the upper flip-flop t_k later than it arrives at the lower flip-flop. Delaying the clock to the source of the maximum-length path causes this path to effectively get longer. In a similar manner, delaying the clock to the destination of the minimum-length path effectively makes this path shorter.

The effect of skew on the minimum-length path, and hence on the hold-time constraint, is shown in Figure 15.11. The contamination delays from the clock to c to d add as above. However, now signal d must stay stable until t_h after the delayed clock *clkd* or $t_h + t_k$ after the original clock *clk*. The effect is the same as increasing the hold time by t_k.

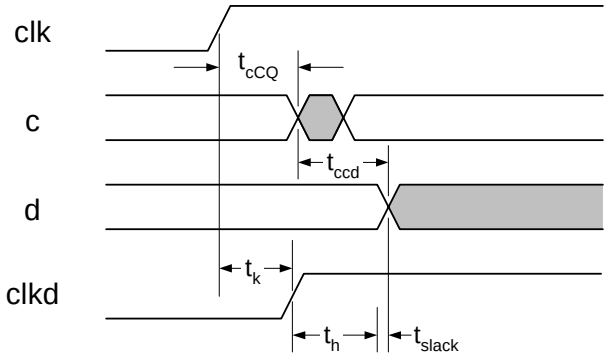


Figure 15.11: Timing diagram showing the effect of clock skew on the hold-time constraint.

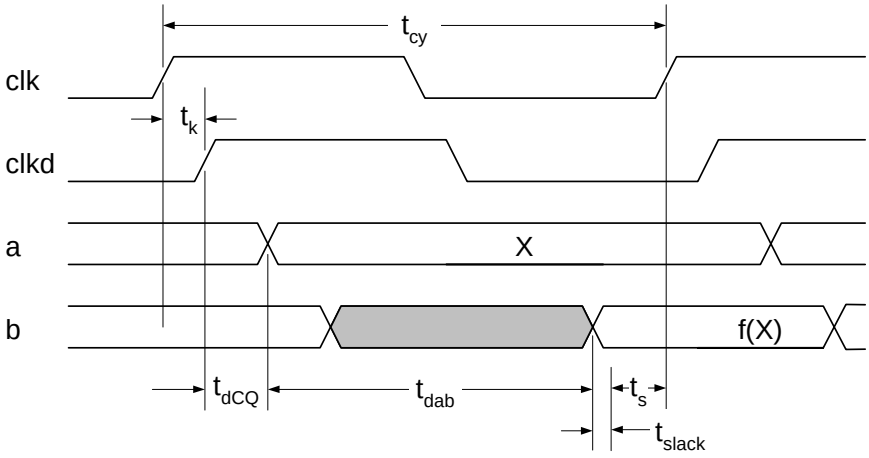


Figure 15.12: Timing diagram showing the effect of clock skew on the setup time constraint.

The timing diagram of Figure 15.12 illustrates the effect of clock skew on the setup time constraint. Delaying the clock to the upper flip-flop delays the transition of a by t_k, effectively adding t_k to the maximum path.

15.5 Timing Examples

example setup and hold calculations with and without skew
- find max frequency
- check hold constraint
- fix hold constraint

15.6 Timing and Logic Synthesis

15.7 Bibliographic Notes

15.8 Exercises

15–1 Setup time.

15–2 Hold time.

15–3 Clock skew.

Chapter 16

Data Path Sequential Logic

In the last chapter we saw how a finite state machine can be synthesized from a state diagram by writing down a table for the next state function and synthesizing the logic that realizes this table. For many sequential functions, however, the next state function can be more simply described by an expression rather than by a table. Such functions are more efficiently described and realized as *datapaths* where the next state is computed as a logical function, often involving arithmetic circuits, multiplexers, and other building block circuits.

16.1 Counters

16.1.1 A Simpler Counter

Suppose you want to build a finite-state machine with the state diagram shown in Figure 16.1. This circuit is forced to state 0 whenever input `r` is true. Whenever input `r` is false, the machine counts through the states from 0 to 31 and then cycles back to 0. Because of this counting behavior, we refer to this finite-state machine as a *counter*.

We could design the counter employing the methodology developed in Chapter 14. A Verilog description taking this approach for a three-bit counter (8 states) is shown in Figure 16.2.[1] A 3-bit wide bank of flip-flops holds the current state `count` and updates it from the next state `next` on each rising edge

[1] A 5-bit counter (32 states) using this approach would require 32 lines in the state table.

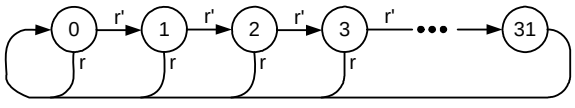


Figure 16.1: State diagram for a 5-bit counter.

```
module Counter1(rst,clk,count) ;
  input rst, clk ; // reset and clock
  output [2:0] count ;
  reg    [2:0] next ;

  DFF #(3) count(clk, next, count) ;

  always@(rst, count) begin
    casex({rst,count})
      4'b1xxxxx: next = 0 ;
      4'd0: next = 1 ;
      4'd1: next = 2 ;
      4'd2: next = 3 ;
      4'd3: next = 4 ;
      4'd4: next = 5 ;
      4'd5: next = 6 ;
      4'd6: next = 7 ;
      4'd7: next = 0 ;
      default: next = 0 ;
    endcase
  end
endmodule
```

Figure 16.2: A three-bit counter FSM specified as a state table.

```
module Counter(clk, rst, count) ;
  parameter n=5 ;
  input rst, clk ; // reset and clock
  output [n-1:0] count ;

  wire   [n-1:0] next = rst? 0 : count+1 ;

  DFF #(n) count(clk, next, count) ;
endmodule
```

Figure 16.3: An n-bit counter FSM specified with a single assign statement.

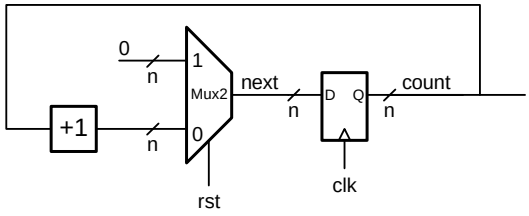


Figure 16.4: Block diagram of a simple counter. The counter state is held in a state register (flip-flops). The next state is selected by a multiplexer to be zero if `rst` is asserted, or the output of an incrementer (`count+1`) otherwise.

of the clock. The `casex` statement directly captures the state table, specifying the next state for each input and current state combination.

While this method for generating counters works, it is verbose and inefficient. The lines of the state table are repetitive. The behavior of the machine can be captured entirely by the single line:

```
assign next = rst ? 0 : count + 1 ;
```

This is a *datapath* description of the finite state machine in which we specify the next state as a function of the current state and inputs.

A verilog description of an n-bit counter module using such a datapath description is shown in Figure 16.3. An n-bit wide bank of flip-flops holds the current state, `count`, and updates it from the next state `next` on each rising edge of the clock. A single assign statement describes the next-state function. The next state is zero if `rst` is high, or `count+1` otherwise.

A block diagram of this counter is shown in Figure 16.4. The figure illustrates the datapath nature of this implementation. The next state is computed by data flowing through paths involving combinational building blocks — including arithmetic blocks and multiplexers. In this case, the two building blocks are a

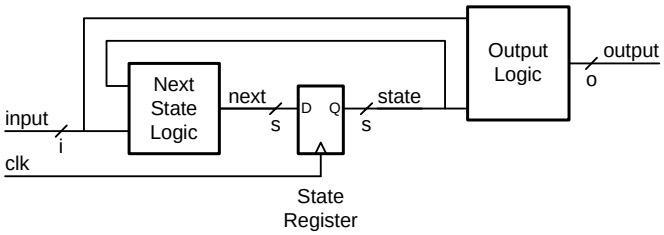


Figure 16.5: In general, a sequential datapath circuit consists of a state register, next-state logic, and output logic. The next-state and output modules are described by functions rather than tables.

multiplexer, which selects between reset (0) and increment (`count+1`), and an incrementer, which increments `count` to give `count+1`.

In general, a sequential datapath circuit takes the form shown in Figure 16.5. As with any synchronous sequential logic module, the state is held in a *state register*. An output logic module computes the output signals as a function of inputs and current state. The next state is computed by a next-state module as a function of inputs and current state. What distinguishes a datapath is that the next-state logic and output logic are described by functions rather than tables. For our simple counter, the output logic is just the current state, and the next state logic is a selection between incrementing or setting to zero. We will see more complex examples of sequential datapath circuits below. However, they all retain this functional description of the next-state and output functions.

16.1.2 An Up/Down/Load (UDL) Counter

Our simple counter had only two choices for next state: reset or increment. Often we require a counter with more options for the next state. A counter may need to count down (decrement) as well as count up, there may be cases where it needs to hold its value, and on occasion we may need to load an arbitrary value into the counter.

Figure 16.6 shows the Verilog description of such an up/down/load (UDL) counter. The counter has four control inputs (`rst, up, down,` and `load`, and one n-bit wide data input `in`. The next-state function is described by a `casex` statement. If reset (`rst`) is asserted ,the next state is 0. Otherwise, if `up` is asserted, the next state is `out+1`. If `down` is asserted (and not `up` or `rst`) the counter decrements by setting the next state to `out-1`. The counter is loaded when `load` is asserted by setting the next state to `in`. Finally, if none of the control inputs are asserted, the counter holds its present value by setting next to `out`.

This description accurately captures the function of the UDL counter and is adequate for most purposes. However, it is somewhat inefficient in that it will result in the instantiation of both an incrementer (to compute `out+1`) and a

```
module UDL_Count1(clk, rst, up, down, load, in, out) ;
  parameter n = 4 ;
  input clk, rst, up, down, load ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  wire [n-1:0] out ;
  reg  [n-1:0] next ;

  DFF #(n) count(clk, next, out) ;

  always@(rst, up, down, load, in, out) begin
    casex({rst, up, down, load})
      4'b1xxx: next = {n{1'b0}} ;
      4'b01xx: next = out + 1'b1 ;
      4'b001x: next = out - 1'b1 ;
      4'b0001: next = in ;
      default: next = out ;
    endcase
  end
endmodule
```

Figure 16.6: Verilog description of an up/down/load (UDL) counter.

```
module UDL_Count2(clk, rst, up, down, load, in, out) ;
  parameter n = 4 ;
  input clk, rst, up, down, load ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  wire [n-1:0] out, outpm1 ;
  reg  [n-1:0] next ;

  DFF #(n) count(clk, next, out) ;

  assign outpm1 = out + {{n-1{down}},1'b1} ;

  always@(rst, up, down, load, in, out, outpm1) begin
    casex({rst, up, down, load})
      4'b1xxx: next = {n{1'b0}} ;
      4'b01xx: next = outpm1 ;
      4'b001x: next = outpm1 ;
      4'b0001: next = in ;
      default: next = out ;
    endcase
  end
endmodule
```

Figure 16.7: Verilog description of an up/down/load (UDL) counter with a shared incrementer/decrementer.

decrementer (to compute `out-1`). From our brief study of computer arithmetic (Chapter 10) we know that these two circuits could be combined.

If we are in an operating mode where saving a few gates matters (which is unlikely) we can describe a more economical counter circuit as shown in Figure 16.7. This circuit factors the increment and decrement operations out of the `casex` statement. Instead, it realizes them in signal `outpm1` (out plus or minus 1). This signal is generated by an assign statement that adds 1 to `out` if `down` is false, and adds -1 to `out` if `down` is true. The code is otherwise identical to that of Figure 16.6.

A block diagram of the UDL counter is shown in Figure 16.8. Like the simple counter of Figure 16.4, and like most datapath circuits, the UDL counter consists of a multiplexer that selects different options for the next state. Some of the options are created by function units. Here the multiplexer has four inputs - to select the input (load), the output of the incrementer/decrementer (up or down), 0 (reset), and count (hold). The single function unit is an incrementer/decrementer that can either add or subtract 1 from the current count. The `down` line controls whether to increment or decrement. A block of combinational logic generates the select signals for the multiplexer by decoding the

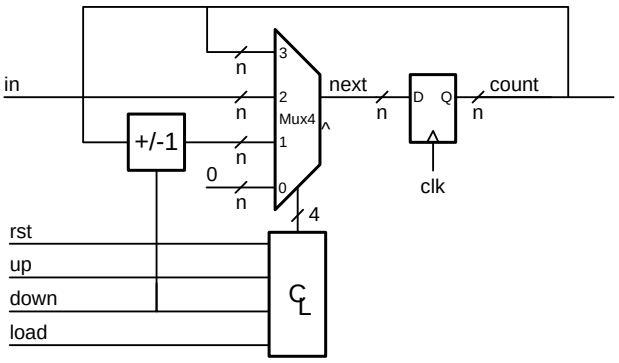


Figure 16.8: Block diagram of an up/down/load counter.

control inputs. Verilog code corresponding to this block diagram is shown in Figure 16.9.

16.1.3 A Timer

In many applications, like a more involved version of our traffic-light controller, we would like a timer that we can set with an initial time t and after t cycles have passed, it signals us that the time is complete. This is analogous to your kitchen timer that you set with an interval, and it signals you audibly when the interval is complete.

A block diagram of an FSM timer is shown in Figure 16.10. It follows our familiar theme of using a multiplexer to select the next state from among constants, inputs, and the outputs of function units (in this case a decrementer). What is different about this block diagram is that it includes an output function unit. A zero checker asserts signal `done` when the count has reached zero.

To operate the timer, the time interval is applied to input `in` and control signal `load` is asserted to load the interval. Each cycle after this load, the internal state `count` counts down. When `count` reaches zero, output `done` is asserted and counting stops. The reset input `rst` is only used to initialize the timer on power-up.

A Verilog description of the timer is shown in Figure 16.11. The style is similar to our simple counter and structural UDL counter with a multiplexer and a state register. The decrementer is implemented in the argument list of the multiplexer. To keep the timer from continuing to decrement, we select the zero input of the multiplexer when `done` is asserted and `load` is not. The final assign statement implements the zero checker.

```
module UDL_Count3(clk, rst, up, down, load, in, out) ;
  parameter n = 4 ;
  input clk, rst, up, down, load ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  wire [n-1:0] out, next, outpm1 ;

  DFF #(n) count(clk, next, out) ;

  assign outpm1 = out + {{n-1{down}},1'b1} ;

  Mux4 #(n) mux(out, in, outpm1, {n{1'b0}},
                {(!rst & !up & !down & !load),
                 (!rst & !up & !down & load),
                 (!rst & (up | down)),
                 rst},
                next) ;
endmodule
```

Figure 16.9: Verilog description of an up/down/load (UDL) counter using a shared incrementer/decrementer and an explicit multiplexer.

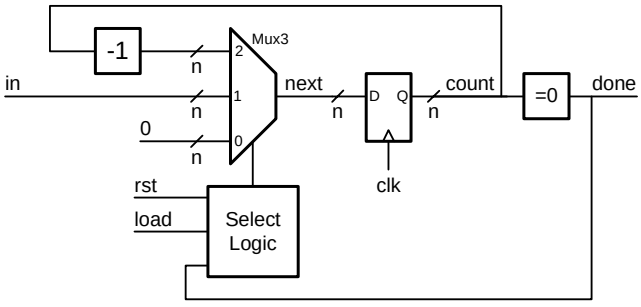


Figure 16.10: Block diagram of a timer FSM.

```
//------------------------------------------------------------------------
// Timer module
// Reset sets count to zero
// Load sets count to in
// Otherwise count decrements and saturates at zero (doesn't wrap)
// Done is asserted when count is zero
//------------------------------------------------------------------------
module Timer(clk, rst, load, in, done) ;
  parameter n=4 ;
  input clk, rst, load ;
  input [n-1:0] in ;
  output done ;
  wire [n-1:0] count, next_count ;

  DFF #(n) cnt(clk, next_count, count) ;
  Mux3 #(n) mux({n{1'b0}}, in, count-1'b1,
                {!rst & !load & !done,
                 load & !rst,
                 rst | (done & !load)},
                next_count) ;
  wire done = !(|count) ;
endmodule
//------------------------------------------------------------------------
```

Figure 16.11: Verilog description of a timer.

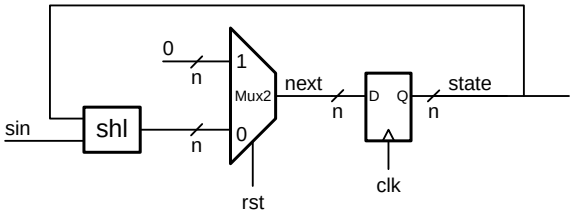


Figure 16.12: Block diagram of a simple shift register.

16.2 Shift Registers

In addition to incrementing, decrementing, and comparing, another popular datapath function is shifting. Shifters are particularly useful in serializers and deserializers that convert data from parallel to serial form and back again.

16.2.1 A Simple Shift Register

A block diagram of a simple shift register is shown in Figure 16.12 and a Verilog description of this module is given in Figure 16.13. Unless the shift register is reset, the next state is the current state shifted one-bit to the left with the `sin` input providing the rightmost bit (LSB). In the Verilog implementation, the 2:1 multiplexer is performed using the select "`? :` " construct, and the left shift is performed by the expression:

```
{out[n-2:0], sin}
```

Here the concatenate operation is used to concatenate the rightmost $n-1$ bits of `out` with `sin`, effectively shifting the bits of out one position to the left and inserting `sin` into the LSB.

This simple shift register could be used for example as a deserializer. A serial input is received on input `sin` and every n clocks the parallel output is read from `out`. A framing protocol of some type is necessary to determine where each parallel symbol starts and stops - i.e., during which clock the parallel output should be read.

16.2.2 Left/Right/Load (LRL) Shift Register

Analogous to our complex counter, we can make a complex shift register that can be loaded shifts in either direction. The block diagram of this left/right/load (LRL) shift register is shown in Figure 16.14 and the Verilog description is shown in Figure **??**. The module uses a five-input multiplexer to select between zero, a left shift (done with concatenation), a right shift (also done with concatenation), the input, and the output. Note that the shift expressions:


```
//------------------------------------------------------------------------
// Basic shift register
// rst - sets out to zero, otherwise out shifts left - sin becomes lsb
//------------------------------------------------------------------------
module Shift_Register1(clk, rst, sin, out) ;
  parameter n = 4 ;
  input clk, rst, sin ;
  output [n-1:0] out ;
  wire [n-1:0] out, next ;

  DFF #(n) cnt(clk, next, out) ;
  assign next = rst ? {n{1'b0}} : {out[n-2:0], sin} ;
endmodule
```

Figure 16.13: Verilog description of simple shift register. If `rst` is asserted the register is set to all 0s, otherwise it shifts left with input `sin` filling the LSB.

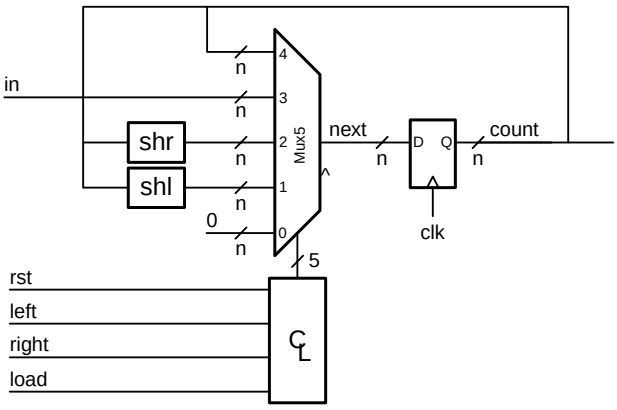


Figure 16.14: Block diagram of a left-right-load shift register.

```
//-----------------------------------------------------------------------
// Left/Right SR with Load
//-----------------------------------------------------------------------
module LRL_Shift_Register(clk, rst, left, right, load, sin, in, out) ;
  parameter n = 4 ;
  input clk, rst, left, right, load, sin ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  wire [n-1:0] out, next ;

  DFF #(n) cnt(clk, next, out) ;
  Mux5 #(n) mux({n{1'b0}}, {out[n-2:0],sin}, {sin, out[n-1:1]}, in, out,
                {(!rst & !left & !right & !load),
                 (!rst & !left & !right & load),
                 (!rst & !left & right),
                 (!rst & left),
                 rst}, next) ;
endmodule
```

Figure 16.15: Verilog description of a left/right/load (LRL) shift register.

```
{out[n-2:0],sin}
{sin, out[n-1:1]}
```

do not actually generate any logic. This shift operation is just wiring - from `sin` and the selected bits of `out` to the appropriate input bits of the multiplexer.

16.2.3 A Universal Shifter/Counter

If we made the LRL shift register module also increment, decrement, and check for zero, it could serve in place of any of the modules we have discussed so far in this section. This idea is not as far-fetched as it may seem. At first you may think that using a full featured module when all we need is a simple counter is wasteful. However, most synthesis systems given constant inputs will eliminate logic that is never used (i.e., if `up` and `down` are always zero the incrementer/decrementer will be eliminated). Thus, the unused logic shouldn't cost us anything in practice.[2] The advantage of the universal shifter/counter is that its a single module to remember and maintain.

Verilog for a universal shifter/counter module is shown in Figure 16.16. This code largely combines the code of the individual modules above. It uses a seven-input multiplexer to select between input, current state, increment, decrement, left shift, right shift, and zero. The increment/decrement is done with an assign

[2] Check that this is true with your synthesis system before using this approach.

(as with the UDL counter) and the shifts are done with concatenation (as with the LRL shifter).

One thing that is new with the universal module is the use of an arbiter `RArb` to make sure that no two select inputs to the multiplexer are asserted at the same time. This arbitration was explicitly coded in the modules above. With seven select inputs its easier to instantiate the arbiter module to perform this logic. In many uses of the universal shifter/counter (or any of our other datapath modules) we can be certain that two command inputs will not be asserted at the same time. In these cases, we don't need the arbitration (whether done with a module or explicitly). However, to verify that the command inputs are in fact one-hot, it is useful to code an *assertion* into the module. The assertion is a logical expression that generates no gates but flags an error during simulation if the expression is violated.

16.3 Control and Data Partitioning

A common theme in digital design is the separation of a module into a control finite-state machine and a datapath as shown in Figure 16.17. The data path computes its next state via multiplexers and function units - like the counters and shift registers in this chapter. The control FSM, on the other hand, computes its next state via a state table. The inputs to the module are separated into control inputs - that affect the state of the control FSM, and data inputs, that supply values to the datapath. Module outputs are partitioned in a similar manner. The control FSM controls the operation of the datapath via a set of command signals. The datapath communicates back to the control FSM via a set of status signals.

The counter and shifter examples we have looked at so far in this chapter are degenerate examples of this organization. They each consist of a datapath - with multiplexers and function units (shifters, incrementers, and/or adders) - and a control unit. However, their control units have been strictly combinational. The datapath commands (e.g, multiplexer selects and add/subtract controls) and control outputs have been functions soley of the current control inputs and datapath status. In this section we will examine two examples of modules where the control section includes internal state.

16.3.1 Example: Vending Machine FSM

Consider the problem of designing the controller for a soft-drink vending machine. The specification is as follows: The vending machine accepts nickels, dimes, and quarters. Whenever a coin is deposited into the coin slot, a pulse appears for one clock cycle on one of three lines indicating the type of coin: `nickel`, `dime`, or `quarter`). The price of the item is set on an n-bit switch internal to the machine (in units of nickels) and is input to the controller on the n-bit signal `price`. When sufficient coins have been deposited to purchase a soft drink, an status signal `enough` is asserted. Any time `enough` is asserted and the

```
//-----------------------------------------------------------------------
// Universal Shifter/Counter
// inputs take priority in order listed
// rst - resets state to zero
// left - shifts state to the left, sin fills LSB
// right - shifts state to the right, sin fills MSB
// up - increments state
// down - decrements state - will not decrement through zero.
// load - load from in
//
// Output done indicates when state is all zeros.
//-----------------------------------------------------------------------
module UnivShCnt(clk, rst, left, right, up, down, load, sin, in, out, done) ;
  parameter n = 4 ;
  input clk, rst, left, right, up, down, load, sin ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  output done ;
  wire [6:0] sel ; // multiplexer select, formed by arbiting the commands
  wire [n-1:0] out, next, outpm1 ;

  assign outpm1 = out + {{n-1{down}},1'b1} ; // incr or decr out
  DFF #(n) cnt(clk, next, out) ;
  RArb #(7) arb({rst, left, right, up, down & ~done, load, 1'b1}, sel) ;
  Mux7 #(n) mux(out, in, outpm1, outpm1,
                {sin,out[n-1:1]},{out[n-2:0],sin},{n{1'b0}}, sel, next) ;
  wire done = !(|out) ;
endmodule
```

Figure 16.16: Verilog description of an up/down/load (UDL) counter using a shared incrementer/decrementer and an explicit multiplexer.

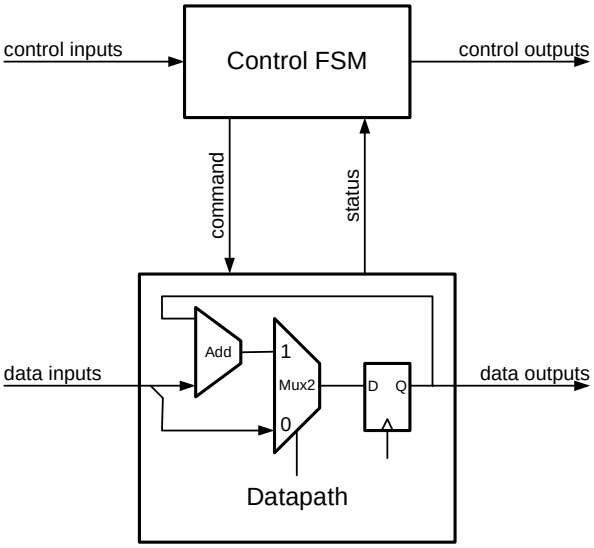


Figure 16.17: Systems are often partitioned into datapath, where the next state is determined by function units, and control, where the next state is determined by state tables.

user presses a `dispense` button, signal `serve` is asserted for exactly one cycle to serve the soft-drink. After asserting `serve`, the FSM must wait until signal `done` is asserted indicating that the mechanism has finished serving the soft drink. After serving is done, the machine returns change (if any) to the user. It does this one nickel at a time, asserting the signal `change`, for exactly one cycle and waiting for signal `done` to indicate that a nickel has been dispensed before dispensing the next nickel or returning to its original state. Any time the signal `done` is asserted, we must wait for `done` to go low before proceeding.

To design this machine, we start by considering the control part of the machine. First, lets look at the inputs and outputs. All of the inputs except `price` (`rst`, `nickel`, `dime`, `quarter`, `dispense`, and `done`) are control inputs, and all of the outputs `serve` and `change` are control outputs. Status that we need from the datapath includes a signal `enough` that indicates that enough money has been deposited, and a signal `zero` that indicates that no more change is owed. Commands to the datapath we will consider below when we look at the data state.

Now lets consider the states of the control portion of the machine. As illustrated in the state diagram of Figure 16.18 operates in three main phases. [3] First, during the deposit phase (state deposit), the user deposits coins and then

[3]In this diagram, edges from a state to itself are omitted. If the conditions on all edges leading out of the current state are not satisfied, the FSM stays in that state.

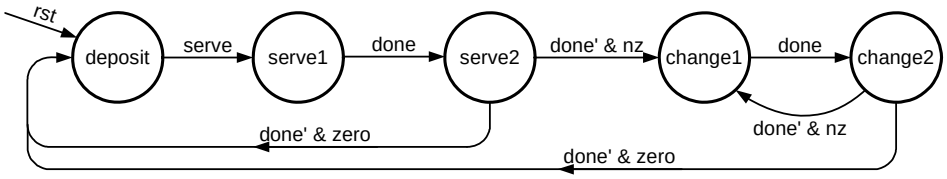


Figure 16.18: State diagram for control part of a vending machine controller. Edges from a state to itself are omitted for clarity. If none of the logic conditions on edges out of a state are true, the machine will remain in that state.

presses `dispense`. When signal `dispense` and `enough` are both asserted the machine advances to the serving phase (states serve1 and serve2). In state serve1 it waits for signal `done` to indicate that the soft drink has been served, and in state serve2, it waits for `done` to be deasserted. We assert the output `serve` during the first cycle only of state serve1. If there is no change to be dispensed, `zero` is true, the FSM returns to the deposit state from serve2. However, if there is change to be dispensed, the FSM enters the change phase (states change1 and change2). The machine cycles through these states, asserting `change` and waiting for `done` in change1 and waiting for `done` to go low in `change2`. We assert output `change` during the first cycle of each visit to state change1. Only when all change has been dispensed do we return to the deposit state.

Now that we have the control states defined, we turn our attention to the data state. This FSM has a single piece of data state - the amount of money the machine currently owes the user - in units of nickels. We'll call this state variable `amount`. The different actions that affect `amount` are:

Reset: amount $\leftarrow$ 0.

Deposit a coin: amount $\leftarrow$ amount + value, where value = 1, 2, or 5 for a nickel, dime, or quarter respectively.

Serve a drink: amount $\leftarrow$ amount - price.

Return one nickel of change: amount $\leftarrow$ amount - 1.

Otherwise: No change in amount.

We can now design a datapath that supports these operations. State variable `amount` can either be zeroed, added to or subtracted from, or hold its value. From these *register transfers* we see that we need the datapath of Figure 16.19. The next state for amount `next` is selected by a 3:1 multiplexer that selects between 0, `amount`, or `sum`, the output of an add/subtract unit that adds or subtracts `value` from `amount`. The `value` is selected by a 4:1 multiplexer to be 1, 2, 5, or `price`. We see from the figure that the command signals needed to

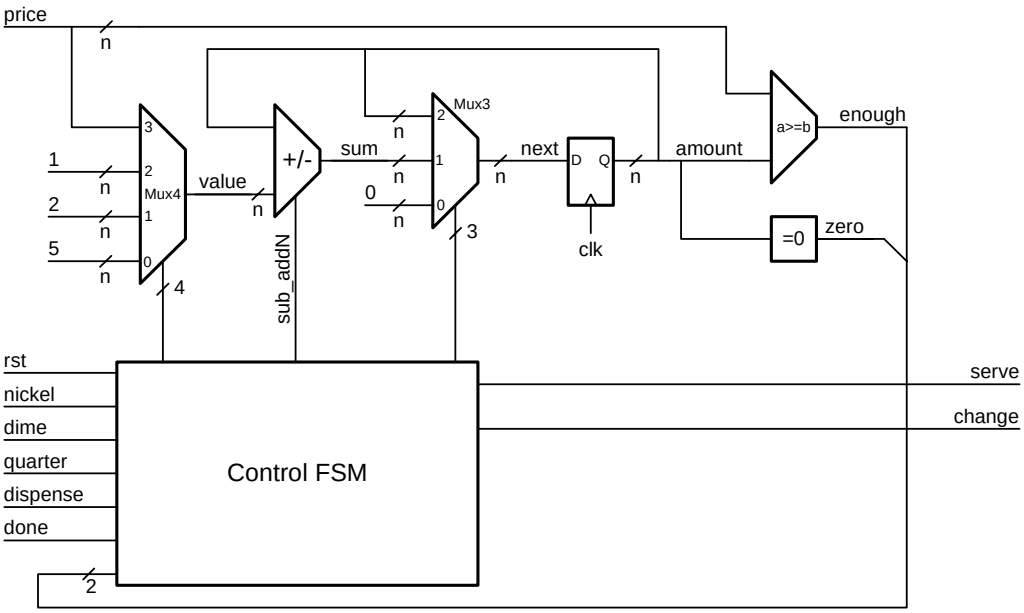


Figure 16.19: Block diagram of a vending machine controller.

```
//-------------------------------------------------------------------------
// VendingMachine - Top level module
// Just hooks together control and datapath
//-------------------------------------------------------------------------
module VendingMachine(clk, rst, nickel, dime, quarter, dispense, done, price,
                      serve, change) ;
  parameter n = `DWIDTH ;
  input clk, rst, nickel, dime, quarter, dispense, done ;
  input [n-1:0] price ;
  output serve, change ;

  wire enough, zero, sub ;
  wire [3:0] selval ;
  wire [2:0] selnext ;

  VendingMachineControl vmc(clk, rst, nickel, dime, quarter, dispense, done,
  enough, zero, serve, change, selval, selnext, sub) ;

  VendingMachineData #(n) vmd(clk, selval, selnext, sub, price, enough, zero) ;
endmodule
```

Figure 16.20: Verilog top-level module for vending-machine controller just declares the control and data modules and connects them together.

control the datapath are the two multiplexer select signals and the add/subtract control.

Verilog code for our vending machine controller is shown in Figures 16.20 through 16.23. The top-level module is shown in Figure 16.20. This module just instantiates the control (`VendingMachineControl`) and data (`VendingMachineData`) modules and connects them up. The command signals `sub`, `selval` and `selnext` are declared at this level as are the status signals `enough` and `zero`.

The Verilog for the control module of our vending machine controller is split over two figures. Figure 16.21 shows the first half of the module which includes the logic for generating the output and command variables. A variable `first` is defined which is used to distinguish the first cycle of states serve1 and change1. After one cycle in either of these states, `first` goes low. This variable enables us to only assert the outputs for one cycle on each visit to these states and to only decrement amount once on each visit to the change1 state. Without the `first` variable, we would need to expand both serve1 and change1 into two states. The outputs `serve` and `change` are generated by ANDing `first` with signals that are true in states serve1 and change1 respectively.

The datapath control signals are determined from input and state variables. The two select signals are one-hot variables. Each bit is determined by a logic expression. Signal `selval` (which selects the value input to the add/subtract unit) selects price when in the deposit state (`dep` true) and dispense is pressed. A 1 is selected if a nickel is input in the deposit state or if we are in the change state. Inputs of 2 and 5 are selected in the deposit state if a dime or a quarter are input respectively. Signal `selnext` selects the next state for the amount variable. Zero is selected if `rst` is true, and the output of the add/subtract unit is selected if variable `selv` is true. Otherwise the current value of amount is selected. Variable `selv` is true in the deposit state if a coin is entered or if dispense and enough are both true and in the change state if first is true. Note that we can select `price` to be the value whenever `dispense` is asserted in the deposit state because we only select the add/subtract output if `enough` is also asserted. Finally, the add/subtract control signal is set to subtract on `dispense` and `change` actions. Otherwise the adder adds.

The second half of the control module in Figure 16.22 shows the next state logic. This logic is implemented as a `casex` statement where the case is on the concatenation of four input bits and the current state. Most transitions encode to a single case. For example, when the machine is in the deposit state, and `dispense` and `enough` are both true, the first case `4'b11xx,`DEPOSIT` is active. It takes the next two cases to encode when the machine stays in the deposit state. A separate assignment statement is used reset the FSM to the deposit state.

A Verilog description of the vending machine controller datapath is shown in Figure 16.23. This code follows the datapath part of Figure **??** very closely. A state register holds the current amount. A three-input multiplexer feeds the state register with zero, the `sum` output of the add/subtract unit, or the current value of amount. An add subtract unit adds or subtracts `value` to or from `amount`. A four-input multiplexer selects the `value` to be added or subtracted.

```
module VendingMachineControl(clk, rst, nickel, dime, quarter, dispense, done,
  enough, zero, serve, change, selval, selnext, sub) ;
  input clk, rst, nickel, dime, quarter, dispense, done, enough, zero ;
  output serve, change, sub ;
  output [3:0] selval ;
  output [2:0] selnext ;
  wire [`SWIDTH-1:0] state, next ; // current and next state
  reg [`SWIDTH-1:0] next1 ;        // next state w/o reset

  // outputs
  wire first ; // true during first cycle of serve1 or change1
  wire serve1 = (state == `SERVE1) ;
  wire change1 = (state == `CHANGE1) ;
  wire serve = serve1 & first ;
  wire change = change1 & first ;

  // datapath controls
  wire dep = (state == `DEPOSIT) ;
  // price, 1, 2, 5
  wire [3:0] selval = {(dep & dispense),
                       ((dep & nickel)| change),
                       (dep & dime),
                       (dep & quarter)} ;
  // amount, sum, 0
  wire selv = (dep & (nickel | dime | quarter | (dispense & enough))) |
              (change & first) ;
  wire [2:0] selnext = {!(selv | rst),selv,rst} ;

  // subtract
  wire sub = (dep & dispense) | change ;

  // only do actions on first cycle of serve1 or change1
  wire nfirst = !(serve1 | change1) ;
  DFF #(1) first_reg(clk, nfirst, first) ;
```

Figure 16.21: Verilog description of control module for vending machine controller (part 1 of 2). This first half of the control module shows the output and command variables implemented with assign statements.

```
  // state register
  DFF #(`SWIDTH) state_reg(clk, next, state) ;

  // next state logic
  always @(state or zero or dispense or done or enough) begin
    casex({dispense, enough, done, zero, state})
      {4'b11xx,`DEPOSIT}:  next1 = `SERVE1 ;  // dispense & enough
      {4'b0xxx,`DEPOSIT}:  next1 = `DEPOSIT ;
      {4'bx0xx,`DEPOSIT}:  next1 = `DEPOSIT ;
      {4'bxx1x,`SERVE1}:   next1 = `SERVE2 ;  // done
      {4'bxx0x,`SERVE1}:   next1 = `SERVE1 ;
      {4'bxx01,`SERVE2}:   next1 = `DEPOSIT ;  // ~done & zero
      {4'bxx00,`SERVE2}:   next1 = `CHANGE1 ;  // ~done & ~zero
      {4'bxx1x,`SERVE2}:   next1 = `SERVE2  ;  // done
      {4'bxx1x,`CHANGE1}:  next1 = `CHANGE2 ;  // done
      {4'bxx0x,`CHANGE1}:  next1 = `CHANGE1 ;  // done
      {4'bxx00,`CHANGE2}:  next1 = `CHANGE1 ;  // ~done & ~zero
      {4'bxx01,`CHANGE2}:  next1 = `DEPOSIT ;  // ~done & zero
      {4'bxx1x,`CHANGE2}:  next1 = `CHANGE2 ;  // ~done & zero
    endcase
  end

  // reset next state
  assign next = rst ? `DEPOSIT : next1 ;
endmodule
```

Figure 16.22: Verilog description of control module for vending machine controller (part 2 of 2). This second half of the control module shows the next-state function implemented with a `casex` statement.

```
module VendingMachineData(clk, selval, selnext, sub, price, enough, zero) ;
  parameter n = 6 ;
  input clk, sub ;
  input [3:0] selval ;  // price, 1, 2, 5
  input [2:0] selnext ; // amount, sum, 0
  input [n-1:0] price ; // price of soft drink - in nickels
  output enough ;       // amount > price
  output zero ;         // amount = zero

  wire [n-1:0] sum ;  // output of add/subtract unit
  wire [n-1:0] amount ; // current amount
  wire [n-1:0] next ;   // next amount
  wire [n-1:0] value ;  // value to add or subtract from amount
  wire ovf ;            // overflow - ignore for now

  // state register holds current amount
  DFF #(n) amt(clk, next, amount) ;

  // select next state from 0, sum, or hold
  Mux3 #(n) nsmux({n{1'b0}}, sum, amount, selnext, next) ;

  // add or subtract a value from current amount
  AddSub #(n) add(amount, value, sub, sum, ovf) ;

  // select the value to add or subtract
  Mux4 #(n) vmux(`QUARTER, `DIME, `NICKEL, price, selval, value) ;

  // comparators
  wire enough = (amount >= price) ;
  wire zero = (amount == 0) ;
endmodule
```

Figure 16.23: Datapath for vending machine controller.

Finally, two assign statements generate the status signals `enough` and `zero`.

A testbench for the vending machine controller is shown in Figure 16.24 and waveforms from simulating the controller with this test bench are shown in Figure 16.25. The test begins by resetting the machine. A nickel is then deposited followed by a dime - bringing `amount` to 3 (15 cents). At this point, we go one cycle with no input to make sure `amount` stays at 3. We then assert `dispense` to make sure that trying to dispense a soft drink before enough money has been deposited doesn't work. Next we deposit two quarters in back-to-back cycles bringing `amount` to 8 and then 13. When `amount` reaches 13 signal `enough` goes high since we have exceeded the price (which is 11). After an idle cycle, we again assert `dispense`. This time it works. The state advances to 001 (serve1) and `amount` is reduced to 2 (price of 11 deducted).

In the first cycle of the serve1 state (state = 001) `serve` is asserted. The machine remains in this state for one more cycle waiting for `done` to go high. It spends just one cycle in state serve2 (state = 011) since `done` is already low and continues into state change1 (state = 010). On the first cycle of change1, output `change` is asserted (to return an nickel to the user) and `amount` is decremented to 1. The machine remains in the change1 state for one more cycle waiting for `done`. It then spends just one cycle in state change2 (state = 100) before returning to change1 - because zero is not true. Again `change` is asserted and `amount` is decremented on the first cycle of change1. This time, however `amount` is decremented to zero and signal `zero` is asserted. After waiting a second cycle in the change1 state, the machine transitions through change2 and back to the deposit state - since `zero` is set - ready to start again.

16.3.2 Example: Combination Lock

As our second control and datapath example, consider an electronic combination lock that accepts input from a decimal keypad. The user must enters the code as a sequence of decimal digits and then presses an `enter` key. If the user entered the correct sequence, the `unlock` output is asserted (presumably to actuate a large bolt that unlocks a door). To relock the machine, the user presses `enter` a second time. If the user entered an incorrect sequence, a `busy` output is asserted and a timer is activated to wait a predefined period before allowing the user to try again. The busy light should not come on until the user had entered the entire sequence and pressed `enter`. If the light were to come on at the first incorrect keypress, this gives the user information that can be used to discover the code one digit at a time.

There are three inputs to our lock system: `key`, `key_valid`, and `enter`. A four-bit code `key` indicates the current key being pressed and is accompanied by a signal `key_valid` that indicates when a key is valid. The keyboard is preprocessed so that every key press asserts key and valid for exactly one cycle. In a similar manner, the `enter` signal is preprocessed so that it is for exactly one cycle each time the user presses the enter key to unlock or relock the bolt. The length of the sequence is set by an internal variable `length`, and the sequence itself is stored in an internal memory.

```
module testVend ;
  reg clk, rst, nickel, dime, quarter, dispense, done ;
  reg [3:0] price ;
  wire serve, change ;

  VendingMachine #(4) vm(clk, rst, nickel, dime, quarter, dispense, done, price,
                         serve, change) ;

  // clock with period of 10 units
  initial begin
    clk = 1 ; #5 clk = 0 ;
    forever
      begin
        $display("%b %h %h %b %b",
        {nickel,dime,quarter,dispense}, vm.vmc.state, vm.vmd.amount, serve, change) ;
        #5 clk = 1 ; #5 clk = 0 ;
      end
    end

  // give prompt feedback
  always @(posedge clk) begin
    done = (serve | change) ;
  end

  initial begin
    rst = 1 ; {nickel, dime, quarter, dispense} = 4'b0 ; price = `PRICE ;
    #25 rst = 0 ;
    #10 {nickel, dime, quarter, dispense} = 4'b1000 ; // nickel 1
    #10 {nickel, dime, quarter, dispense} = 4'b0100 ; // dime 3
    #10 {nickel, dime, quarter, dispense} = 4'b0000 ; // nothing
    #10 {nickel, dime, quarter, dispense} = 4'b0001 ; // try to dispense early
    #10 {nickel, dime, quarter, dispense} = 4'b0010 ; // quarter 8
    #10 {nickel, dime, quarter, dispense} = 4'b0010 ; // quarter 13
    #10 {nickel, dime, quarter, dispense} = 4'b0000 ; // nothing
    #10 {nickel, dime, quarter, dispense} = 4'b0001 ; // dispense 2
    #10 dispense = 0 ;
    #100 $stop ;
  end
endmodule
```

Figure 16.24: Verilog testbench for vending machine controller.

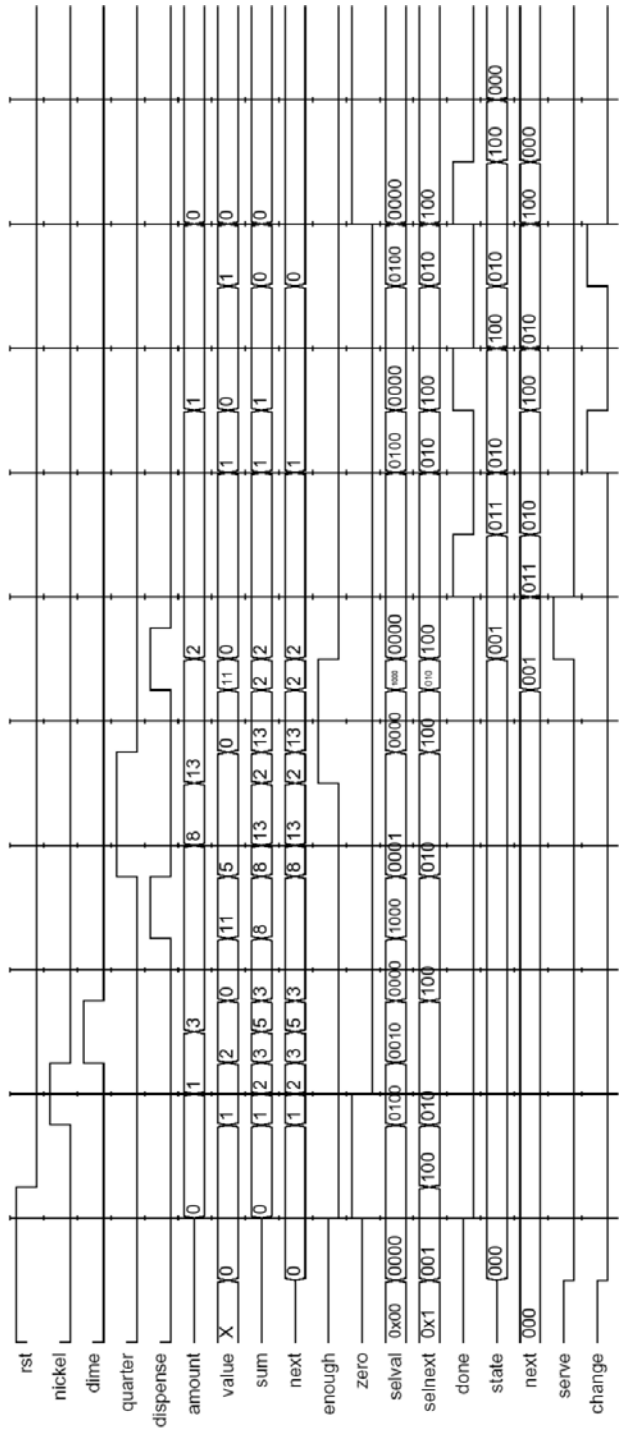


Figure 16.25: Waveforms from simulating the vending machine controller with the test bench of Figure 16.24

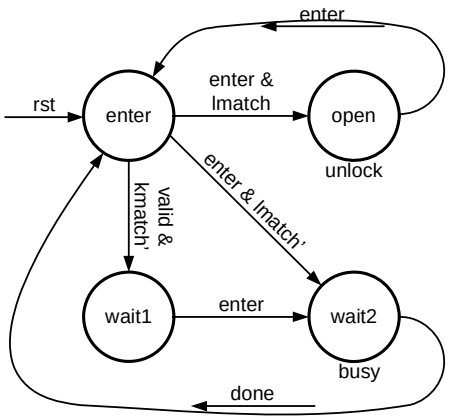


Figure 16.26: State diagram for control portion of combination lock.

A state diagram describing the control portion of our combination lock machine is shown in Figure 16.26. The machine resets to the enter state. In this state the machine accepts input. Each time a key is pressed, it is checked against the expected digit. If its correct (kmatch) the machine stays in the enter state, if not (`valid` $\wedge$ $\overline{\texttt{kmatch}}$) the machine transitions to the wait1 state. The machine waits in the wait1 state until `enter` is pressed, and then enters the wait2 state. The machine starts a timer in the wait2 state and remains in this state, asserting `busy` until the timer signals `done`.

When the entire code has been entered correctly, the machine will still be in the enter state - an incorrect digit would have taken it to wait1 - and `lmatch` will be true - the length of the code entered matches the internal `length` variable. If `enter` is pressed at this point, the machine transitions to the open state (`enter` $\wedge$ `lmatch`) and the bolt is unlocked. A second `enter` in the open state returns to the reset. If in the enter state, the `enter` key is pressed when the length does not match (either too few or too many digits in the codeword) (`enter` $\wedge\overline{\texttt{lmatch}}$) the machine goes to the wait2 state.

A block diagram of the datapath portion of our combination lock is shown in Figure 16.27. The datapath has two distinct sections. The upper section compares the length and value of the code entered. The lower section times the wait period when an incorrect code is entered. This section consists of a single timer module that loads the timer interval `twait` when control signal `load` is asserted. The timer then counts down. When it reaches zero, status signal `done` is asserted.

The upper half of the datapath consists of a counter, a ROM, and two comparators. The counter keeps track of which digit of the code we are on. It is reset to zero before entering the enter state and then counts up as each key is pressed. The counter output, index, selects the digit to be compared next.

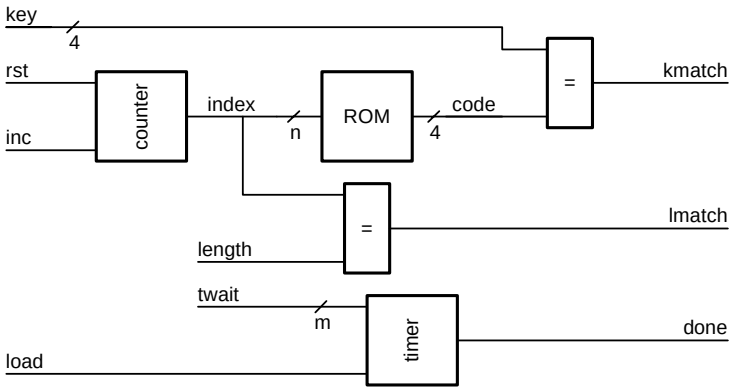


Figure 16.27: Block diagram of datapath portion of combination lock.

Signal index is compared to length to generate status signal `lmatch`. It is also used as an address for the ROM containing the code. The current digit of the code is read from the ROM, signal code, and compared to the key entered by the user to generate status signal `kmatch`.

Figures 16.28 and 16.29 show a Verilog implementation of our combination lock. Here we have put both the control and datapath in a single module. This eliminates the code that is otherwise required to wire together the control and data sections, but for a large module can result in an unweildy piece of code.

The datapath portion of the module is shown in Figure 16.28. The Verilog follows directly from the block diagram of Figure 16.27. An up/down/load counter (Section 16.1.2) is used for the counter. Only the `rst` and `up` control inputs are used. The `load` and `down` are zero. The synthesizer will take advantage of these zero control inputs when synthesizing the counter and eliminate the unused logic. The timer module of Section 16.1.3 is used to count down in the wait2 state. The two comparators are implemented with assignment statements.

Figure 16.29 shows the control portion of the Verilog description of the combination lock module. The top portion of the code generates the output and next state variables. Outputs `busy` and `unlock` are generated by decoding the state variable `state` since these outputs are true whenever the machine is in the wait2 and open states respectively. The enter and wait1 states are decoded as well for use in the command equations. The `rstctr` command signal is set to reset the digit counter on reset or in states that transition to the enter state (wait2 and open) so the counter is zeroed and ready to sequence digits in the enter state. Similarly the `load` signal loads the timer in states that transition to the wait2 state (enter and wait1) so it can count down in wait2. The `inc` signal increments the counter in the enter state each time a key is pressed.

The next-state function is implemented with a `casex` statement with a sep-

```
//--------------------------------------------------------------------
// CombLock
// Inputs:
//   key - (4-bit) accepts a code digit each time key_valid is true
//   key_valid - signals when a new code digits is on key
//   enter - signals when entire code has been entered
// Outputs:
//   busy - asserted after incorrect code word entered during timeout
//   unlock - asserted after correct codeword is entered until enter
//            is pressed again.
//--------------------------------------------------------------------
module CombLock(clk, rst, key, key_valid, enter, busy, unlock) ;
  parameter n = 4 ; // bits of code length
  parameter m = 4 ; // bits of timer
  input clk, rst, key_valid, enter ;
  input [3:0] key ;
  output busy, unlock ;

  //--- datapath ----------------------------------------
  wire rstctr ; // reset the digit counter
  wire inc ;    // increment the digit counter
  wire load ;   // load the timer
  wire done ;   // timer done
  wire [n-1:0] index ;
  wire [3:0] code ;
  UDL_Count #(n) ctr(clk, rstctr, inc, 1'b0, 1'b0, 4'b0, index) ; // counter
  ROM #(n,4) rom(index, code) ; // ROM storing the code
  Timer #(m) tim(clk, rst, load, `TWAIT, done) ; // wait timer
  wire kmatch = (code == key) ;       // key comparator
  wire lmatch = (index == `LENGTH) ;  // length comparator
```

Figure 16.28: Verilog description of the combination lock (part 1 of 2). This section describes the datapath and closely follows Figure 16.27.

```
  //--- control ----------------------------------------
  wire [`SWIDTH-1:0] state, next ;       // current and next state
  reg  [`SWIDTH-1:0] next1 ;             // for reset
  wire senter = (state == `ENTER) ;      // decode state
  wire unlock = (state == `OPEN) ;
  wire busy = (state == `WAIT2) ;
  wire swait1 = (state == `WAIT1) ;
  assign rstctr = rst | unlock | busy ;  // reset before returning to enter
  assign inc = senter & key_valid ;      // increment on each key entry
  assign load = senter | swait1 ;        // load before entering wait2

  DFF #(`SWIDTH) sr(clk, next, state) ; // state register

  always @(enter or lmatch or key_valid or kmatch or done or state) begin
    casex({enter, lmatch, key_valid, kmatch, done,state})
      {5'bxx10x,`ENTER}: next1 = `WAIT1 ;  // valid & ~kmatch
      {5'b0x11x,`ENTER}: next1 = `ENTER  ; // valid & kmatch
      {5'b110xx,`ENTER}: next1 = `OPEN   ;  // enter & lmatch
      {5'b10xxx,`ENTER}: next1 = `WAIT2  ; // enter & ~lmatch
      {5'b0x0xx,`ENTER}: next1 = `ENTER  ; // ~enter & ~valid

      {5'b1xxxx,`OPEN}: next1 = `ENTER  ; // enter
      {5'b0xxxx,`OPEN}: next1 = `OPEN   ; // ~enter

      {5'b1xxxx,`WAIT1}: next1 = `WAIT2   ; // enter
      {5'b0xxxx,`WAIT1}: next1 = `WAIT1   ; // ~enter

      {5'bxxxx1,`WAIT2}: next1 = `ENTER   ; // done
      {5'bxxxx0,`WAIT2}: next1 = `WAIT2   ; // ~done
    endcase
  end
  assign next = rst ? `ENTER : next1 ; // reset
endmodule
```

Figure 16.29: Verilog description of the combination lock (part 2 of 2). This section describes the control. Assign statements generate the command and output signals. A `casex` statement computes the next-state function.

arate `assign` statement to reset to the enter state. The case variable is a concatenation of the five input and status signals that affect the next state function with the current state.

Waveforms from simulating the combination lock module on a sequence of test inputs are shown in Figure 16.30. The test visits all states and traverses all edges of the state diagram of Figure 16.26. This is done with three attempts to unlock the lock. One correct attempt and two failures. After reset, the test first enters the correct sequence with a pause after the 1 and after the 7. On the cycle after entering the final 8, `enter` is pressed and the next cycle `unlock` goes high and the machine enters the open state (`state = 1`. After one cycle low, `enter` goes high again returning the machine to the enter state.

The second attempt to unlock the lock involves entering an invalid code. As soon as the first key is entered incorrectly (7 instead of 8) the machine transits to the wait1 state (`state` = 2). It stays in this state until `enter` is pressed at which time it goes to the wait2 state (`state` = 3). In the wait2 state the `busy` output is asserted and the timer starts counting down from 4.[4]

The third attempt to unlock the lock involves entering a code of the wrong length. After the first digit is entered correctly, `enter` is asserted. Because the correct length code has not yet been entered (`lmatch` =0), the machine transits to the wait2 state and starts counting down the timer.

16.4 Bibliographic Notes

16.5 Exercises

16–1 - design a counter that can count up, count down, load and that saturates counting up (at a programmable max count) and counting down (at zero).

16–2 - design a datapath circuit to compute Fibonacci numbers. Each cycle the circuit should output the next Fibonacci number (starting with 0 on the first cycle). The circuit should have parameterized width and should signal when the next number is too large to be represented at this width.

16–3 - redesign the fibonacci number circuit of Exercise 16–2 to be 32-bits wide but use only an 8-bit adder.

16–4 - vending machine with extended coin pulse.

16–5 - vending machine with saturating add

16–6 - combination lock with multiple users - first digit selects one of 8 codes.

16–7 - combination lock that forgives one error. optional, calculate probability of guessing

[4] In practice we would use a much longer timeout. However using a short timeout greatly reduces simulation time and results in an easier to read set of waveforms.

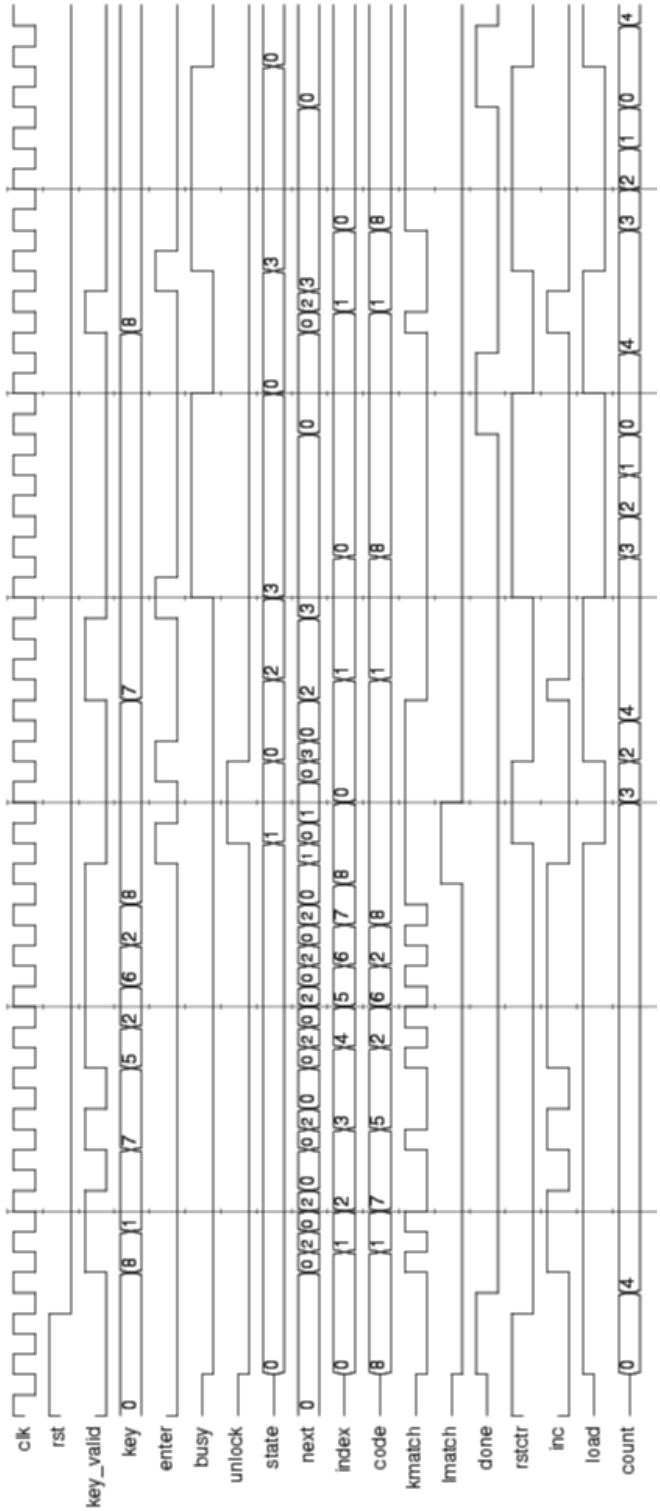


Figure 16.30: Waveforms from simulating the combination lock module of Figures 16.28 and 16.29.

Chapter 17

Factoring Finite State Machines

Factoring a state machine is the process of splitting the machine into two or more simpler machines. Factoring can greatly simplify the design of a state machine by separating orthogonal aspects of the machine into separate FSMs where they can be handled independently. The separate FSMs communicate via logic signals. One FSM provides input control signals to another FSM and senses its output status signals. Such factoring, if done properly, makes the machine simpler and also makes it easier to understand and maintain — by separating issues.

In a factored FSM, the state of each sub-machine represents one dimension of a multi-dimensional state space. Collectively the states of all of the sub-machines define the state of the overall machine — a single point in this state space. The combined machine has a number of states that is equal to the product of the number of states of the individual sub-machines — the number of points in the state space.[1] With individual sub-machines having a few 10s of states, it is not unusual for the overall machine to have thousands to millions of states. It would be impractical to handle such a large number of states without factoring.

We have already seen one form of factoring in Section 16.3 where we developed a state machine with a data path component and a control component. In effect we factored the total state of the machine into a datapath portion and a control portion. Here we generalize this concept by showing how the control portion itself can be factored.

In this chapter we illustrate factoring by working two examples. In the first example, we start with a flat FSM and factor it into multiple simpler FSMs. In the second example we derive a factored FSM directly from the specification, without bothering with the flat FSM. Most real FSMs are designed using the latter method. A factoring is usually a natural outgrowth of the specification

[1] For most machines, not all points in the state space are reachable.

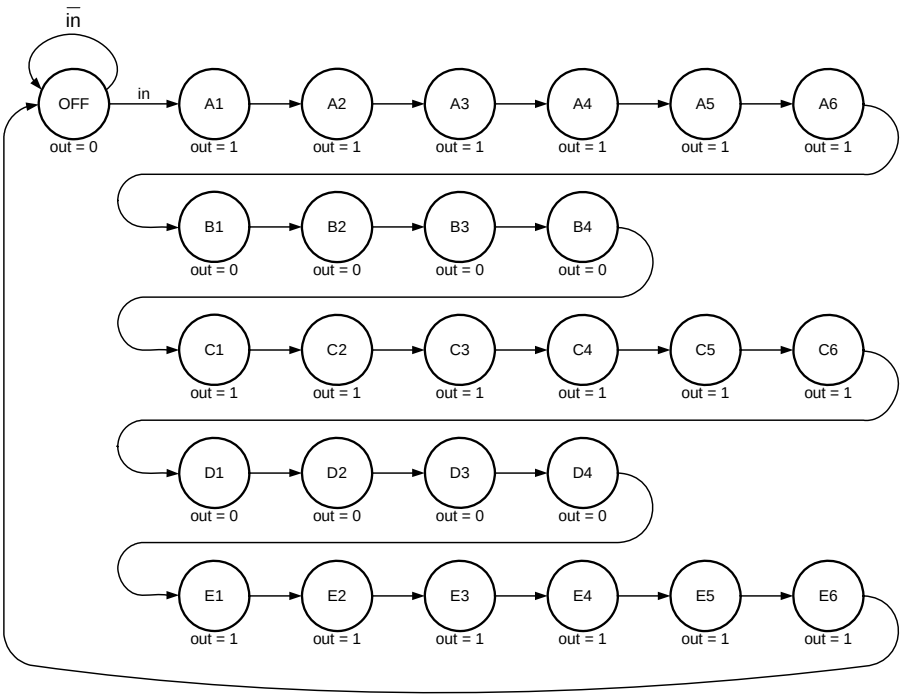


Figure 17.1: State diagram for the light flasher.

of a machine. It is rarely applied to an already flat machine.

17.1 A Light Flasher

Suppose you have been asked to design a light *flasher*. The flasher has a single input `in` and a single output `out`. When `in` goes high (for one cycle) it initiates the flashing sequence. During this sequence output `out`, which drives a light-emitting diode (LED) *flashes* three times. For each flash `out` goes high (LED on) for six cycles. Output `out` goes low for four cycles between flashes. After the third flash your FSM returns to the *off* state awaiting the next pulse on `in`.

A state diagram for this light flasher is shown in Figure 17.1. The FSM contains 27 states six for each of the three flashes, four for each of the two periods between flashes, and one `OFF` state. We can implement this FSM with a big `case` statement, with 27 cases. However, suppose the specifications change to require 12 cycles for each flash, 4 flashes, and 7 cycles between flashes. Because this machine is *flat* any one of these changes would require completely changing the `case` statement.

We will illustrate the process of factoring with our light flasher. This machine

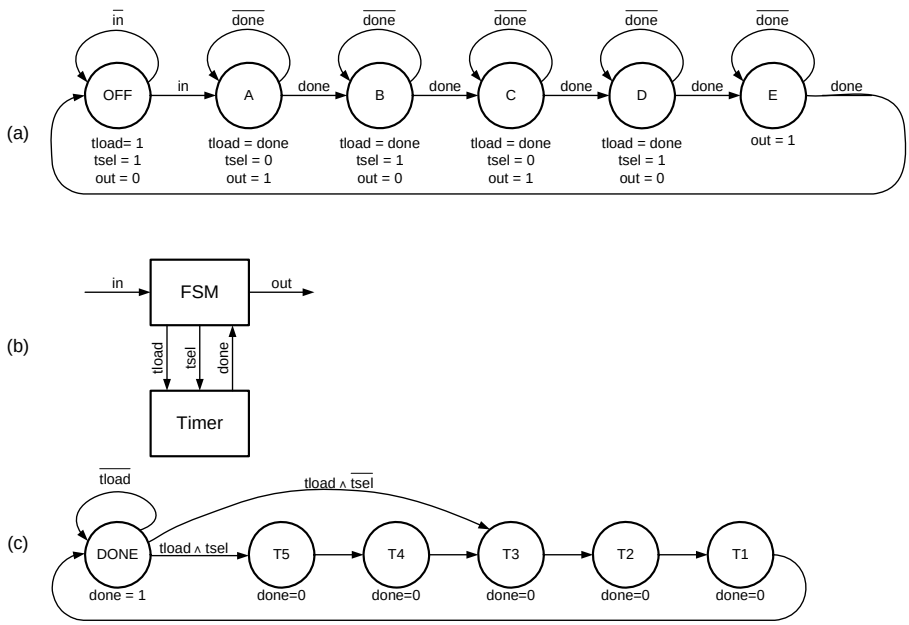


Figure 17.2: State diagram for the light flasher. With on and off timing factored out into a timer. (a) State diagram of *master* FSM. (b) Block diagram showing connections between master FSM and timer FSM. (c) State diagram of timer FSM.

can be factored in two ways. First, we can factor out the counting of on and off intervals into a timer. This allows each of the six or four state sequences for a given flash or interval to be reduced to a single state. This factoring of timing both simplifies the machine and makes it considerably simpler to change the on and off intervals. Second, we can factor out the three flashes (even though the last is slightly different). This again simplifies the machine and also allows us to change the number of flashes.

Figure 17.2 shows how the flasher is factored to separate the process of counting time intervals into a separate machine. As shown in Figure 17.2(b), the factored machine consists of a *master* FSM that accepts `in` and genreates `out` and a timer FSM. The timer FSM receives two control signals from the master and returns a single status signal. The `tload` control signal instructs the timer to load a value into a count-down timer (i.e., to start the timer), and `tsel` selects the value to be loaded. When `tsel` is high, the timer is set to count six cycles. The counter counts four cycles when `tsel` is low. When the counter has completed its count down the `done` signal is asserted and remains asserted until the counter is reloaded. A state diagram for this timer is shown

in Figure 17.2(c).[2] The timer is implemented as a datapath FSM as discussed in Section 16.1.3.

A state diagram of the master FSM is shown in Figure 17.2(a). The states correspond exactly to the states of Figure 17.1 except that each sequence of repeated states (e.g., A1 to A6) is replaced by a single state (e.g., A). The machine starts in the `OFF` state. This state repeatedly loads the timer with the count-down value for the *on* sequence (`tsel = 1`). When `in` goes true, the FSM enters state A, `out` goes high, and the timer begins counting down. The master FSM stays in state A waiting for the timer to finish counting and signal `done`. It will remain in this state for six cycles. During the last cycle in state A, `done` is true, so `tload` is aserted (`tload = done`) and the counter is loaded with the count-down value for the *off* sequence (`tsel = 0`). Note that `tload` is asserted only during this final cycle of state A, when `done` is true. If it was asserted during each cycle of state A the timer would continually reset and never reach done. Since `done` is true in this final cycle, the machine enters state B on the next cycle and `out` goes low. The process repeats in states B through E with each state waiting for `done` before moving to the next state. Each of these states (except E) loads the counter for the next state during its last cycle by specifying `tload = done`.

If we compare the FSM of Figure 17.2 to the flat FSM of Figure 17.1 we realize that the state of the flat machine has been separated. The portion of the state that represents the cycle count during the current flash or space (the numeric part of the state name, or the horizontal position in Figure 17.1) is held in the counter, while the portion of state that reflects which flash or space we are currently on (the alpha part of the state name, or the vertical position in Figure 17.1) is held in the master FSM. By decomposing the state into horizontal and vertical in this manner, we are able to represent all 27 states of the original machine with just two six-state machines.

Verilog code for the master FSM of Figure 17.2(a) is shown in Figure 17.3. The `Flash` module instantiates a state register and a timer (see Figure **??**) and then uses a case statement to describe the combinational next state and output logic. For each of the six states a single concatenated assignment is used to set the output signal, `out`, the two timer controls, `tload` and `tsel`, and the next state `next1`. Note that the values assigned to timer control `tload` depend on the value of the timer status `done`. This is an example of a state machine where an input `done` directly affects an output `tload` with no delay. In each state, the next state is determined with a `? :` statement that waits for either `in` or `done` before advancing to the next state. A final assign statement resets the machine to the `OFF` state.

For completeness the verilog code for the timer is shown in Figure 17.4. The approach taken is similar to that described in Section 16.1.3.

The waveform display from a simulation of the factored flasher of Figure 17.2 is shown in Figure 17.5. The output, the fourth line from the top, shows the

[2] This state diagram only shows `tload` being active in state `DONE`. In fact, our timer can be loaded in any state. The additional edges are omitted for clarity.

```
// define states for flash1
`define SWIDTH 3
`define S_OFF 3'b000  // off state
`define S_A   3'b001  // first flash
`define S_B   3'b010  // first space
`define S_C   3'b011  // second flash
`define S_D   3'b100  // second space
`define S_E   3'b101  // third and final flash

// Flash - flashes out three times 6 cycles on, 4 cycles off
//         each time in is asserted.
module Flash(clk, rst, in, out) ;
  input clk, rst, in ; // in triggers start of flash sequence
  output out ;        // out drives LED
  reg  out ;                        // output
  wire [`SWIDTH-1:0] state, next ; // current state
  reg  [`SWIDTH-1:0] next1  ;      // next state without reset
  reg  tload, tsel ;               // timer inputs
  wire done ;                      // timer output

  // instantiate state register
  DFF #(`SWIDTH) state_reg(clk, next, state) ;

  // instantiate timer
  Timer1 timer(clk, rst, tload, tsel, done) ;

  // next state and output logic
  always @(state or rst or in or done) begin
    case(state)
      `S_OFF: {out, tload, tsel, next1} =
              {1'b0, 1'b1, 1'b1, in ? `S_A : `S_OFF } ;
      `S_A:   {out, tload, tsel, next1} =
              {1'b1, done, 1'b0, done ? `S_B : `S_A } ;
      `S_B:   {out, tload, tsel, next1} =
              {1'b0, done, 1'b1, done ? `S_C : `S_B } ;
      `S_C:   {out, tload, tsel, next1} =
              {1'b1, done, 1'b0, done ? `S_D : `S_C } ;
      `S_D:   {out, tload, tsel, next1} =
              {1'b0, done, 1'b1, done ? `S_E : `S_D } ;
      `S_E:   {out, tload, tsel, next1} =
              {1'b1, done, 1'b1, done ? `S_OFF : `S_E } ;
      default:{out, tload, tsel, next1} =
              {1'b1, done, 1'b1, done ? `S_OFF : `S_E } ;
    endcase
  end

  // reset to off state
  assign next = rst ? `S_OFF : next1 ;
endmodule
```

Figure 17.3: Verilog description of the master FSM from Figure 17.2(a).

```
// define time intervals
// load 5 for 6-cycle interval 5 to 0.
`define T_WIDTH 3
`define T_ON  3'd5
`define T_OFF 3'd3

// Timer 1 - reset to done state.  Load time when tload is asserted
//   Load with T_ON if tsel, otherwise T_OFF.  If not being loaded or
//   reset, timer counts down each cycle.  Done is asserted and timing
//   stops when counter reaches 0.
module Timer1(clk, rst, tload, tsel, done) ;
  parameter n=`T_WIDTH ;
  input clk, rst, tload, tsel ;
  output done ;
  wire [n-1:0] count ;
  reg  [n-1:0] next_count ;
  wire done ;

  // state register
  DFF #(n) state(clk, next_count, count) ;

  // signal done
  assign done = !(|count) ;

  // next count logic
  always@(rst or tload or tsel or done or count) begin
    casex({rst, tload, tsel, done})
      4'b1xxx: next_count = `T_WIDTH'b0 ;
      4'b011x: next_count = `T_ON ;
      4'b010x: next_count = `T_OFF ;
      4'b00x0: next_count = count - 1'b1 ;
      4'b00x1: next_count = count ;
      default: next_count = count ;
    endcase
  end
endmodule
```

Figure 17.4: Verilog description for the timer FSM used by the light flasher of Figure 17.2.

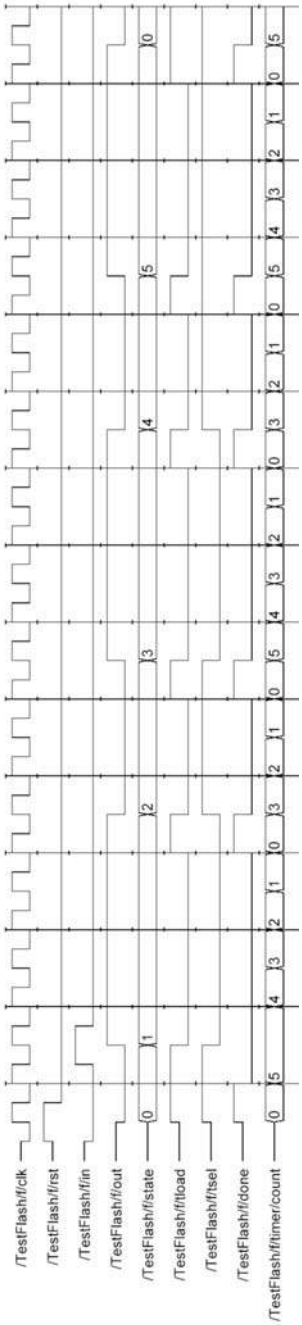


Figure 17.5: A waveform display showing a simulation of the factored light flasher of Figure 17.2.

desired three pulses with each pulse six clocks wide and spaces four clocks wide. The state of the master FSM is directly below the output, and the state of the timer is on the bottom line. The waveforms show how the master FSM stays in one state while the timer counts down — from 5 to 0 for flashes, and from 3 to 0 for spaces. The timer control lines (directly above the timer state) show how in states A-E (1-5) `tload` follows `done`.

We can factor our flasher further by recognizing that states A, C, and E are repeating the same function. The only difference is the number of remaining flashes. We can factor the number of remaing flashes out into a second counter as shown in Figure 17.6. Here the master FSM has only three states that determine whether the machine is off, in a flash, or in a space. The state that determines the position within a flash or space is held in a timer (just as in Figure 17.2. Finally, the state that determines the number of remaining flashes is held in a counter. Collectively the three FSMs, the master, the timer, and the counter, determine the total state of the factored machine. Each of the three sub-machines determines the state along one axis of a three-dimensional state space.

The state diagram of the master FSM for the doubly factored machine is shown in Figure 17.6(b). The machine has only three states. It starts in the `OFF` state. In the off state, both the timer and the counter are loaded. The timer is loaded with the count-down value for a flash. The counter is loaded with one less than the number of flashes required (i.e. the counter is loaded with a 3 for 4 flashes). Having input `in` go high causes a transition to the `FLASH` state. In the `FLASH` state the output `out` is true, the timer counts down, and the counter is idle. During the last cycle of the `FLASH` state the timer has reached its zero state and `tdone` is true. During this cycle the timer is reloaded with the count-down value for a space. With `tdone` true, the FSM proceeds from the `FLASH` state to the `SPACE` state if the counter is not done (`cdone` false). Otherwise, if this was the last flash (`cdone` true) the machine returns to the `OFF` state. In the `SPACE` state, the output is false, the timer counts down, and the counter is idle. In the final cycle of the `SPACE` state, `tdone` is true. This causes the counter to decrement, reducing the count of the number of remaining flashes, and the timer to reload with the count-down value for a flash.

The Verilog code for the doubly factored flasher of Figure 17.6 is shown in Figure 17.7 and the Verilog description of the counter module is shown in Figure 17.8. The next-state and output function for the master FSM again uses a concatenated assignment to set the next state, the output, and the four control signals with a single assignment. A nested `? :` statement is used to compute the next state for the `FLASH` state to test both `tdone` and `cdone`. In several states, status signal `tdone` is passed directly through to control signals `tload` and `cdec`. The counter module is nearly identical to the timer module but with slightly different control because the counter only decrements when `cdec` is asserted while the timer always decrements. With some generalization a single parameterized module could be used for both functions.

The waveforms from a simulation of the doubly factored light flasher are shown in Figure 17.9. For this simulation, the counter was initialized with a

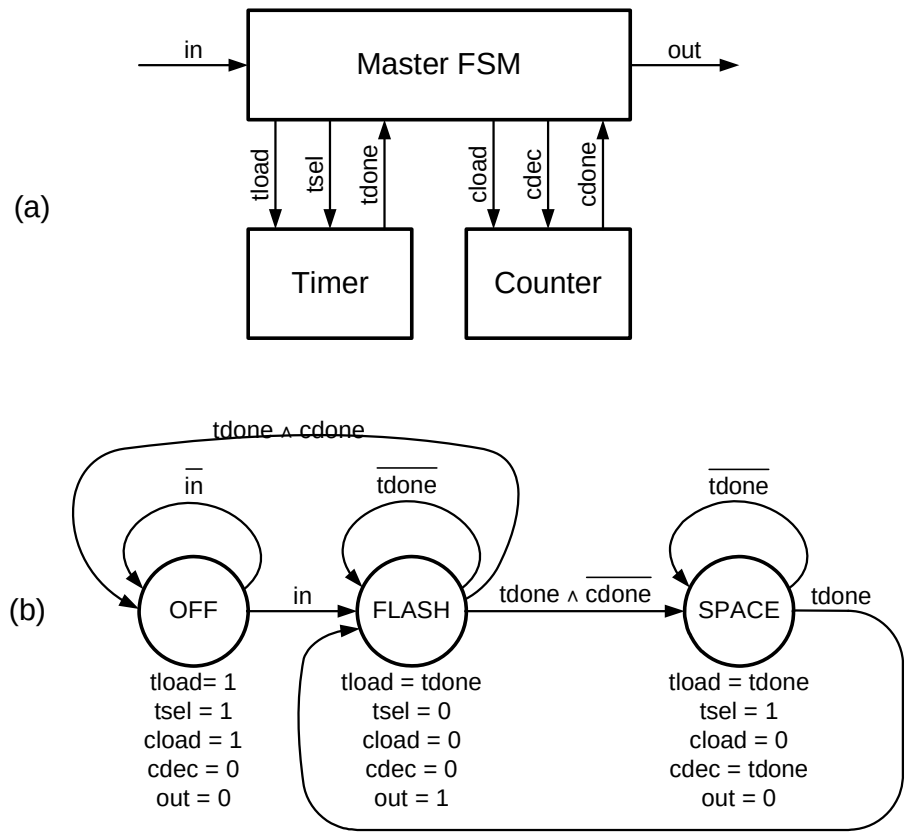


Figure 17.6: The light flasher of Figure 17.1 factored twice. The position within the current flash is held in a timer. The number of flashes remaining is held in a counter. Finally, whether we are off, in a flash, or a space is determined by the master FSM. (a) Block diagram of the twice factored machine. (b) State diagram of the master FSM.

```
// defines for doubly factored states
‘define XWIDTH 2
‘define X_OFF   2’b00
‘define X_FLASH 2’b01
‘define X_SPACE 2’b10

module Flash2(clk, rst, in, out) ;
  input clk, rst, in ; // in triggers start of flash sequence
  output out ;        // out drives LED
  reg  out ;                        // output
  wire [‘XWIDTH-1:0] state, next ; // current state
  reg  [‘XWIDTH-1:0] next1  ;      // next state without reset
  reg  tload, tsel, cload, cdec ;  // timer and countr inputs
  wire tdone, cdone ;              // timer and counter outputs

  // instantiate state register
  DFF #(‘XWIDTH) state_reg(clk, next, state) ;

  // instantiate timer and counter
  Timer1   timer(clk, rst, tload, tsel, tdone) ;
  Counter1 counter(clk, rst, cload, cdec, cdone) ;

  always @(state or rst or in or tdone or cdone) begin
    case(state)
      ‘X_OFF:  {out, tload, tsel, cload, cdec, next1} =
               {1’b0, 1’b1, 1’b1, 1’b1, 1’b0,
                in ? ‘X_FLASH : ‘X_OFF } ;
      ‘X_FLASH:{out, tload, tsel, cload, cdec, next1} =
               {1’b1, tdone, 1’b0, 1’b0, 1’b0,
                tdone ? (cdone ? ‘X_OFF : ‘X_SPACE) : ‘X_FLASH } ;
      ‘X_SPACE:{out, tload, tsel, cload, cdec, next1} =
               {1’b0, tdone, 1’b1, 1’b0, tdone,
                tdone ? ‘X_FLASH : ‘X_SPACE } ;
      default:{out, tload, tsel, cload, cdec, next1} =
               {1’b0, tdone, 1’b1, 1’b0, tdone,
                tdone ? ‘X_FLASH : ‘X_SPACE } ;
    endcase
  end

  assign next = rst ? ‘X_OFF : next1 ;
endmodule
```

Figure 17.7: Verilog description of the master FSM from Figure 17.6.

```
// defines for pulse counter
// load with 3 for four pulses
`define C_WIDTH 2
`define C_COUNT 3

// Counter1 - pulse counter
//   cload - loads counter with C_COUNT
//   cdec  - decrements counter by one if not already zero
//   cdone - signals when count has reached zero

module Counter1(clk, rst, cload, cdec, cdone) ;
  parameter n=`C_WIDTH ;
  input clk, rst, cload, cdec ;
  output cdone ;
  wire [n-1:0] count ;
  reg  [n-1:0] next_count ;
  wire cdone ;

  // state register
  DFF #(n) state(clk, next_count, count) ;

  // signal done
  assign cdone = !(|count) ;

  // next count logic
  always@(rst or cload or cdec or cdone or count) begin
    casex({rst, cload, cdec, cdone})
      4'b1xxx: next_count = `C_WIDTH'b0 ;
      4'b01xx: next_count = `C_COUNT ;
      4'b0010: next_count = count - 1'b1 ;
      4'b00x1: next_count = count ;
      default: next_count = count ;
    endcase
  end
endmodule
```

Figure 17.8: Verilog description of the counter from Figure 17.6.

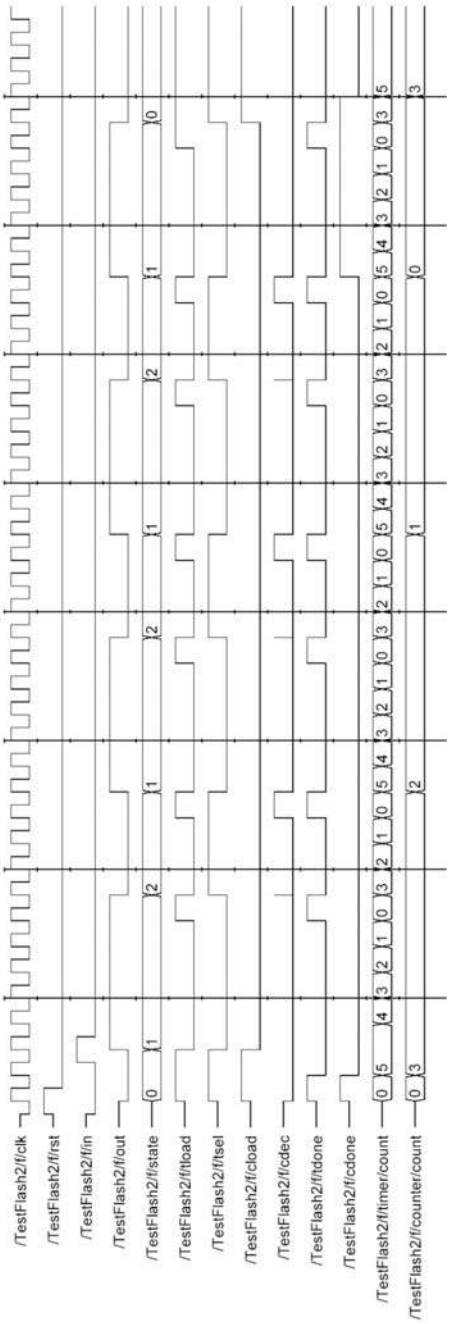


Figure 17.9: A waveform display showing a simulation of the doubly factored light flasher of Figure 17.6.

3 so that the master FSM produces 4 flashes. The different time scales of the three state variables are clearly visible. The counter (bottom line) moves most slowly, counting down from 3 to 0 over the four-flash sequence. It decrements after the last cycle of each space. At each point in time, it represents the number of flashes left to do after the current flash is complete. The master FSM state moves the next most slowly. After starting on the 0 (`OFF`) state, it alternates between the 1 (`FLASH`) state and the 2 (`SPACE`) state until the four flashes are complete. Finally, the timer state moves most rapidly counting down from 5 to 0 for flashes, or 3 to 0 for spaces.

17.2 Traffic Light Controller

As a second example of factoring, we consider a more sophisticated version of the traffic-light controller we introduced in Section 14.3. This machine has two inputs `car_ew` and `car_lt` that indicate that cars are waiting on the east-west road (`ew`) and that cars are waiting in a left-turn lane (`lt`). The machine has nine output lines that drive three sets of three lights each one set each for the north-south road, the east-west road, and the left-turn lane (from the north-south road). Each set of lights consists of a red light, a yellow light, and a green light.

Normally the light will be green for the north-south road. However, if a car is detected on the east-west road or the left-turn lane, we wish to switch the lights so that the east-west or left-turn light is green (with priority going to the left-turn lane). Once the lights are switched to east-west or left-turn, we leave them switched until either no more cars are detected in that direction, or until a timer expires. The lights then return to green in the north-south direction.

Each time we switch the lights we switch the lights in the active direction from green to yellow. Then, after a time interval, we switch them to red. Then, after a second time interval, we switch them to green. The lights are not allowed to change again until they have been green for a third time interval.

Given this specification we decide to factor the finite-state machine that implements this traffic-light controller into five modules as shown in Figure 17.10. A master FSM accepts the inputs and decides which direction should be green `dir`. To time when it is time to force the lights back to north-south, it uses a timer, `Timer1`. It also receives a signal `ok` back from the combiner that indicates that the sequence from the last direction change is complete and the direction is allowed to change again.

The combiner module maintains a current direction state, and combines this current direction state with the `light` signal from the light FSM to generate the 9 light outputs `lights`. The combiner also receives direction requests from the master FSM on the `dir` signal and sequences the light FSM in response to these requests. When a new direction request occurs, the combiner deasserts `on` (sets it low) to ask the light FSM to switch the lights to red. Once the lights are red, the current direction is set equal to the direction request and the `on` signal is asserted to request that the lights be sequenced to green. Only after

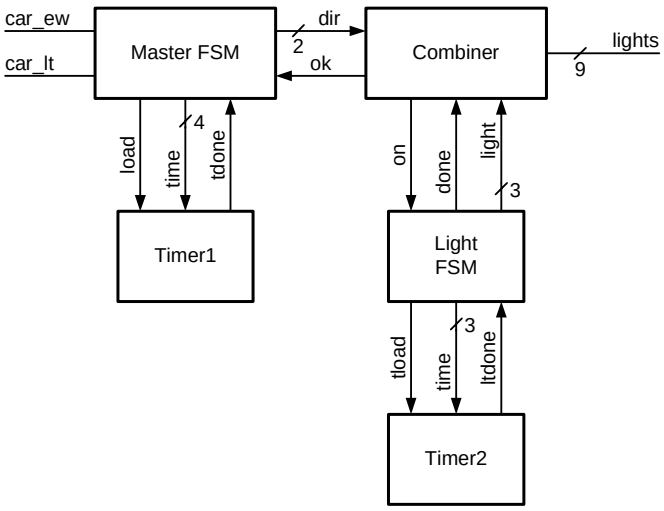


Figure 17.10: Block diagram of a factored traffic-light controller.

the `done` signal from the light FSM is asserted, signaling that the sequence is complete and the lights have been green for the required time interval is the `ok` signal asserted to allow another direction change.

The modules here illustrate two types of relationships. The master FSM and combiner modules form a *pipeline*. Requests flow down this pipeline from left to right. A request is input to the Master FSM in the form of a transition on the `car_ew` and `car_lt` inputs. The Master processes this request and in turn issues a request to the combiner on the `dir` lines. The combiner in turn processes the request and outputs the appropriate sequence on the `lights` output in response. We will discuss pipelines in more depth in Chapter 21.

The `ok` signal here is an example of a *flow control* signal. It provides back-pressure on the master FSM to prevent it from getting ahead of the combiner and light FSMs. The master FSM makes a request and it is not allowed to make another request until the `ok` signal indicates that the rest of the circuit is finished processing the first request.

The other relationships in Figure 17.10 are *master-slave* relationships. The Master FSM acts as a master to Timer1, giving it commands, and Timer1 acts as a slave, receiving the commands and carrying them out. In a similar manner the combiner is the master of the Light FSM which in turn is the master of Timer2.

The light FSM sequences the traffic lights from green to red and then back to green. It receives requests from the combiner on the `on` signal and responds to these requests with the `done` signal. It also generates the 3-bit `light` signal which indicates which light in the current direction should be illuminated. When

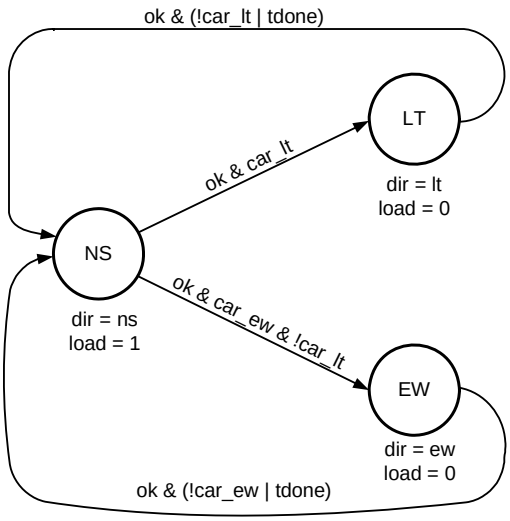


Figure 17.11: State diagram for the master FSM of Figure 17.10.

the `on` signal is set high, it requests that the light signal switch the `light` signal to green. When this is completed, and the minimum green time interval has elapsed, the light FSM responds by asserting `done`. When the `on` signal is set low, it requests that the light FSM sequence the `light` signal to red — via a yellow state and observing the required time intervals. When this is completed `done` is set low. The light FSM uses its own timer (Timer2) to count out the time intervals required for light sequencing.

For the inteface between the light FSM and the combiner, the `done` signal provides flow control. The combiner can only toggle `on` high when `done` is low and can only toggle `on` low when `done` is high. After toggling `on`, it must wait for `done` to switch to the same state as `on` before switching `on` again.

A state diagram for the master FSM is shown in Figure 17.11. The machine starts in the `NS` state. In this state Timer1 is loaded so it can count down in the `LT` and `EW` states, and the requested direction is `NS`. State `NS` is exited when the `ok` signal indicates that a new direction can be requested and when one of the `car` signals indicates that there is a car waiting in another direction. In the `EW` and `LT` states, the new direction is requested and the state is exited with `ok` is true and either there is no longer a car in that direction or the timer has expired, as signaled by `tdone`.

The light FSM is a simple light sequencer similar to our light flasher of Section 17.1. A state diagram for the light FSM is shown in Figure 17.12. Like the light flasher, during the last cycle in each state the timer is loaded with the timeout for the next state. The machine starts in the `RED` state. It transitions to `GREEN` when the timer is done and the `on` signal from the combiner indicates

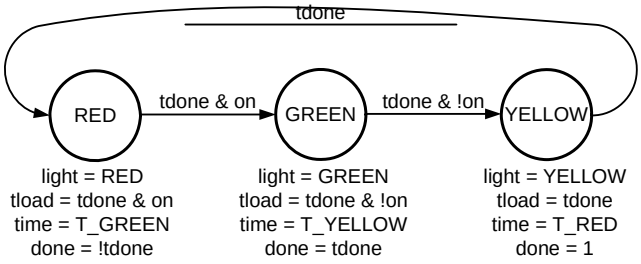


Figure 17.12: State diagram for the light FSM of Figure 17.10.

that a green light is requested. The `done` signal is asserted after the timer in the `GREEN` state has completed its count down. The transition to `YELLOW` is triggered by the timer being complete and the `on` signal being low. The transition from `YELLOW` to `RED` occurs on the timer alone. The `done` signal is held high during the `YELLOW` state. It is allowed to go low, signaling that the transition to `RED` is complete, only after the timer has completed its countdown in the `RED` state.

A Verilog description of the master FSM for the factored traffic-light controller is shown in Figure 17.13. This module instantiates a timer and then uses a `case` statement to realize a three-state FSM that exactly follows the state diagram of Figure 17.11. Verilog select, "`?  :`", statements are used for the next-state logic. A triply-nested select statement is used in state `M_NS` to test in turn signals `ok`, `car_lt`, and `car_ew`.

The master FSM does not instantiate the combiner. Both the master and combiner are instantiated as *peer* modules at the top level.

A Verilog description of the combiner module is shown in Figure 17.14. This module accepts a `dir` input from the master FSM, responds to the master FSM with flow-control signal `ok`, and generates the `lights` output. The key piece of state in the combiner module is the current direction register `cur_dir`. This holds the direction that the light FSM is currently sequencing. It is updated with the requested direction, `dir`, when `on` and `done` are both low. This occurs when the combiner has requested that the lights be sequence to red (`on` low), and the light FSM has completed this requested action (`done` low).

The `on` command, which requests the light FSM to sequence the lights to green, is asserted any time the current direction and the requested direction match. When the master FSM requests a new direction, this causes `on` to go low - requesting the light FSM to sequence the lights red in preparation for the new direction.

The `ok` response to the master FSM is asserted when `on` and `done` are both true. This occurs when the light FSM has completed sequencing the requested direction to green.

The `lights` output is computed by a `case` statement with the current direction as the case variable. This case statement insertes the `light` output from

```
// master FSM states
// these also serve as direction values
`define MWIDTH 2
`define M_NS   2'b00
`define M_EW   2'b01
`define M_LT   2'b10

//Master FSM
//  car_ew - car waiting on east-west road
//  car_lt - car waiting in left-turn lane
//  ok     - signal that it is ok to request a new direction
//  dir    - output signaling new requested direction

module TLC_Master(clk, rst, car_ew, car_lt, ok, dir) ;
  input clk, rst, car_ew, car_lt, ok ;
  output [1:0] dir ;

  wire [`MWIDTH-1:0] state, next ; // current state and next state
  reg  [`MWIDTH-1:0] next1 ;       // next state without reset
  reg  tload ;                     // timer load
  reg  [1:0] dir ;                 // direction output
  wire tdone ;                     // timer completion

  // instantiate state register
  DFF #(`MWIDTH) state_reg(clk, next, state) ;

  // instantiate timer
  Timer #(`TWIDTH) timer(clk, rst, tload, `T_EXP, tdone) ;

  always @(state or rst or car_ew or car_lt or ok or tdone) begin
    case(state)
      `M_NS: {dir, tload, next1} =
             {`M_NS, 1'b1, ok ? (car_lt ? `M_LT
                                        : (car_ew ? `M_EW : `M_NS))
                           : `M_NS} ;
      `M_EW: {dir, tload, next1} =
             {`M_EW, 1'b0, (ok & (!car_ew | tdone)) ? `M_NS : `M_EW} ;
      `M_LT: {dir, tload, next1} =
             {`M_LT, 1'b0, (ok & (!car_ew | tdone)) ? `M_NS : `M_LT} ;
      default: {dir, tload, next1} =
             {`M_NS, 1'b0, `M_NS} ;
    endcase
  end
  assign next = rst ? `M_NS : next1 ;
endmodule
```

Figure 17.13: Verilog description of the master FSM for the traffic-light controller.

```
//-----------------------------------------------------------------------
// Combiner -
//   dir - direction request from master FSM
//   ok  - acknowledge to master FSM
//   lights - 9-bits to control traffic lights {NS,EW,LT}
//-----------------------------------------------------------------------
module TLC_Combiner(clk, rst, dir, ok , lights) ;
  input clk, rst ;
  input [1:0] dir ;
  output ok ;
  output [8:0] lights ;
  wire done ;
  wire [2:0] light ;
  reg [8:0] lights ;
  wire [1:0] cur_dir ;

  // request green from light FSM until direction changes
  wire on = (cur_dir == dir) ;

  // update direction when light FSM has made lights red
  wire [1:0] next_dir = rst ? 2'b0 : ((!on & !done) ? dir : cur_dir) ;

  // ok to take another change when light FSM is done
  wire ok = on & done ;

  // current direction register
  DFF #(2) dir_reg(clk, next_dir, cur_dir) ;

  // combine cur_dir and light to get lights
  always @(cur_dir or light) begin
    case(cur_dir)
      `M_NS: lights = {light, `RED, `RED} ;
      `M_EW: lights = {`RED, light, `RED} ;
      `M_LT: lights = {`RED, `RED, light} ;
      default: lights = {`RED, `RED, `RED} ;
    endcase
  end

  // light FSM
  TLC_Light lt(clk, rst, on, done, light) ;
endmodule
```

Figure 17.14: Verilog description of the combiner for the traffic-light controller.

the light FSM into the position corresponding to the current direction and sets the other positions to be red.

A Verilog description of the light FSM is shown in Figure 17.15. The module instantiates a timer and then uses a `case` statement to implement a FSM with state transitions as shown in Figure 17.12. Verilog select statements are used for the next-state logic.

Waveforms from a simulation of the factored traffic-light controller are shown in Figure 17.16. The machine is initally reset with the Master FSM in the NS state (00). Output `dir` is also NS (00). The `ok` line is initially low because the light FSM has not yet finished its sequencing to make the lights green in the NS direction. The lights are initially all red (444) but change after one cycle to be green in the north-south direction (144).

The light FSM is initialized to the RED (00) state and advances to the green state because `on` and `tdone` are both asserted. In the green state it starts a timer and waits until `tdone` is again asserted before signaling `done` to the combiner which in turn causes the combiner to signal `ok` to the Master FSM.

Because `car_ew` is asserted, once the `ok` signal goes high, the Master FSM requests a change in direction to east-west by setting `dir` to 01. The combiner responds by setting `on` low to request the light FSM to sequence the current direction's light to red. The light FSM in turn responds by transitiong to the YELLLOW state (10) which causes `light` to go to yellow (2) and `lights` to go to 244 - yellow in the north south direction. When the light timer completes its countdown, the light machine enters the RED state (00), `light` becomes 4 (RED), and `lights` becomes 444 - red in all directions. Once the light timer counts down the minimum time in the all-red state, the light FSM sets `done` low signaling that it has completed the transition.

With `on` and `done` both low, the combiner updates `cur_dir` to east-west (01) and sets `on` high to request the light FSM to sequence the lights green in the east-west direction. When the light FSM completes this action - including timing the green state - it sets `done` true. This in turn causes the combiner to set `ok` true, signaling the Master FSM that it is ready to accept a new direction.

When `ok` is asserted the second time, the Mater FSM requests the north-south direction again (`dir` = 00). This decision is driven by the `car_ew` line being low and the master timer being done. This new direction causes `on` to go low, sequencing the lights to the all red state. Then, after `cur_dir` is updated, setting `on` high to sequence them back to the green state. When this is all complete, `ok` is asserted again.

On this third assertion of `ok`, signal `car_lt` is true, so a left-turn is requested (`dir` = 10) and the lights are sequenced to red and back to green again. When the sequencing is completed, `ok` is asserted for a fourth time. This time `car_lt` is still asserted, but the master timer is done, so a north-south direction is again requested.

```
//-----------------------------------------------------------------------
// Light FSM
//-----------------------------------------------------------------------
module TLC_Light(clk, rst, on, done, light) ;
  input clk, rst, on ;
  output done ;
  output [2:0] light ;
  reg  [2:0] light ;
  reg  done ;
  wire [`LWIDTH-1:0] state, next ; // current state, next state
  reg  [`LWIDTH-1:0] next1 ;       // next state w/o reset
  reg  tload ;
  reg  [`TWIDTH-1:0] tin ;
  wire tdone ;

  // instantiate state register
  DFF #(`LWIDTH) state_reg(clk, next, state) ;

  // instantiate timer
  Timer timer(clk, rst, tload, tin, tdone) ;

  always @(state or rst or on or tdone) begin
    case(state)
      `L_RED: {tload, tin, light, done, next1} =
              {tdone & on, `T_GREEN, `RED, !tdone,
               (tdone & on) ? `L_GREEN : `L_RED} ;
      `L_GREEN: {tload, tin, light, done, next1} =
              {tdone & !on, `T_YELLOW, `GREEN, tdone,
               (tdone & !on) ? `L_YELLOW : `L_GREEN} ;
      `L_YELLOW: {tload, tin, light, done, next1} =
              {tdone, `T_RED, `YELLOW, 1'b1, tdone ? `L_RED : `L_YELLOW} ;
      default: {tload, tin, light, done, next1} =
              {tdone, `T_RED, `YELLOW, 1'b1, tdone ? `L_RED : `L_YELLOW} ;
    endcase
  end

  assign next = rst ? `L_RED : next1 ;
endmodule
```

Figure 17.15: Verilog description of the light FSM for the traffic-light controller.

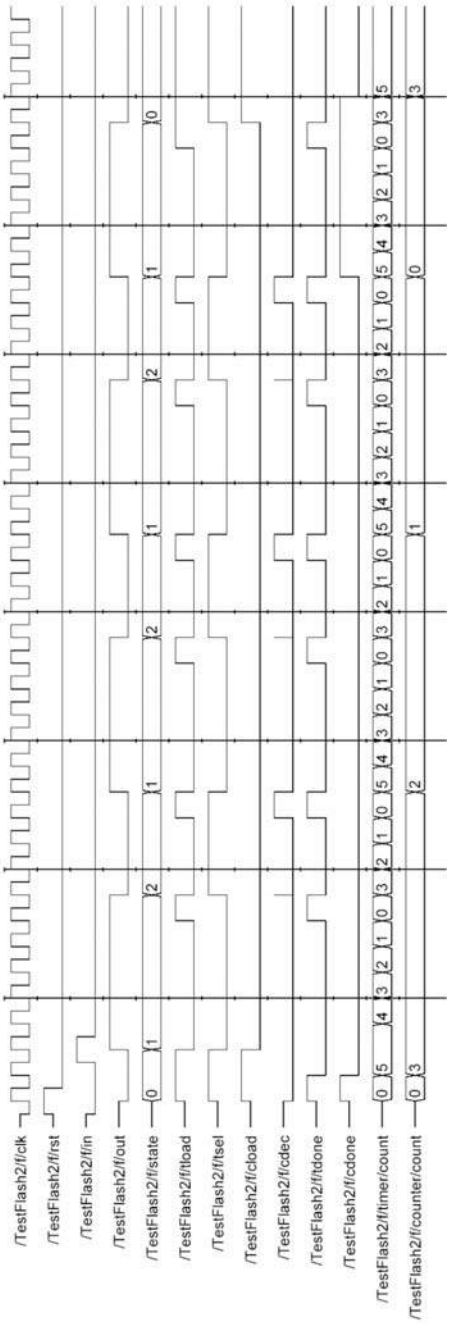


Figure 17.16: A waveform display showing a simulation of the factored traffic-light controller.

17.3 Exercises

17–1 *Flasher.* Modify the flasher FSM to flash an SOS sequence - three short flashes (one clock each) followed by three long flashes (four clocks each) followed by three short flashes again.

17–2 *Traffic Light Controller.* Modify the traffic light controller so that north-south and east-west have equal priority. Add an additional input, `car_ns` that indicates when a car is waiting in the north-south direction. In either the NS or EW states, change to the other if there is a car waiting in the other direction **and** the master timer is done.

17–3 *Traffic Light Controller.* Modify the traffic light controller so that a switch from a red light to a green light is preceeded by both red and yellow being on for three clocks.

Chapter 18

Microcode

Realizing the next-state and output logic of a finite-state machine using a memory array gives a flexible way of realizing a FSM. The function of the FSM can be altered by changing the contents of the memory. We refer to the contents of the memory array as *microcode* and a machine realized in this manner is called a *microcoded* FSM. Each word of the memory array determines the behavior of the machine for a particular state and input combination and is referred to as a *microinstruction*.

We can reduce the required size of a microcode memory by augmenting the memory with special logic to compute the next state, and by selectively updating infrequently changing outputs. A single microinstruction can be provided for each state, rather than one for each state $\times$ input combination, by providing an instruction sequencer and a *branch* microinstruction to cause changes in control flow. Bits of the microinstruction can be shared by different functions by defining different microinstruction types for control, output, and other functions.

Microcode was originated by Maurice Wilkes at Cambridge University in 1951 to implement the control logic for the EDSAC computer [**?**]. It has been widely used since that time in many different types of digital systems.

18.1 A Simple Microcoded FSM

Figure 18.1 shows a block diagram of a simple microcoded FSM. A memory array holds the next-state and output functions. Each word of the array holds the next state and output for a particular combination of input and next state. The array is addressed by the concatenation of the current state and the inputs. A pair of registers holds the current state and current output.

In practice the memory could be realized as a RAM or EEPROM allowing software to reprogram the microcode. Alternatively the memory could be a ROM. With a ROM, a new mask set is required to reprogram the microcode. However, this is still advantageous as changing the program of the ROM does not otherwise alter the layout of the chip. Some ROM designs even allow the

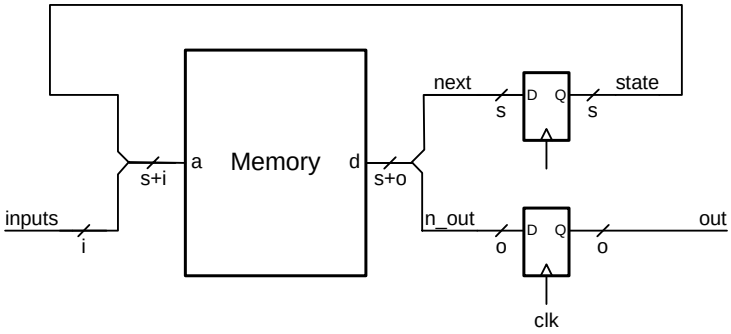


Figure 18.1: Block diagram of a simple microcoded FSM.

program to be changed by altering only a single metal-level mask — reducing the cost of the change. Some designs take a hybrid approach putting most of the microcode into ROM (to reduce cost) but keeping a small portion of microcode in RAM. A method is provided to redirect an arbitrary state sequence into the RAM portion of the microcode to allow any state to be *patched* using the RAM.

A verilog description of this FSM is shown in Figure 18.2. It follows the schematic exactly with the addition of some logic to reset the state when `rst` is asserted. The module `ROM` is a read-only-memory that takes an address `{state, in}` and returns a microinstruction `uinst`. The microinstruction is then split into its next-state and output components. The FSM of Figures 18.1 and 18.2 is very simple because it has no function until the ROM is programmed.

To see how to program the ROM to realize a finite state machine, consider our simple traffic-light controller from Section 14.3. The state diagram of this controller is repeated in Figure 18.3. To fill our microcode ROM, we simply write down the next state and output for each current state/input combination as shown in Table 18.1. Consider the first line of the table. Address `0000` corresponds to state GNS (green north-south) with input `car_ew` = 0. For this state, the output is 100001 (green north-south, red east-west) and the next state is GNS (000). Thus, the contents of ROM location 0000 is the 000100001, the concatenation of the next state 000 with the output. For the second line of the table, address 0001 corresponds to GNS with `car_ew` = 1. The output here is the same as for the first line, but the next state is now YNS (001) hence the contents of this ROM location is 001100001. The remaining rows of the table are derived in a similar manner. The ROM itself is loaded with the contents of the column labeled Data.

The results of simulating the microcoded FSM of Figure 18.2 using the ROM contents from Table 18.1 are shown in Figure 18.4. The output, state, and microcode ROM address, and microcode ROM data (microinstruction) (the bottom four signals) are displayed in octal (base 8). The system intitializes to state 0 (GNS) with output 41 (green (4) north-south, red (1) east-west). Then

```
module ucode1(clk,rst,in,out) ;
  parameter n = 1 ; // input width
  parameter m = 6 ; // output width
  parameter k = 3 ; // bits of state

  input  clk, rst ;
  input  [n-1:0] in ;
  output [m-1:0] out ;

  wire   [k-1:0] next, state ;
  wire [k+m-1:0] uinst ;

  DFF #(k) state_reg(clk, next, state) ;  // state register
  DFF #(m) out_reg(clk, uinst[m-1:0], out) ; // output register
  ROM #(n+k,m+k) uc({state, in}, uinst) ; // microcode store
  assign next = rst ? {k{1'b0}} : uinst[m+k-1:m] ; // reset state
endmodule
```

Figure 18.2: Verilog description of a simple microcoded FSM

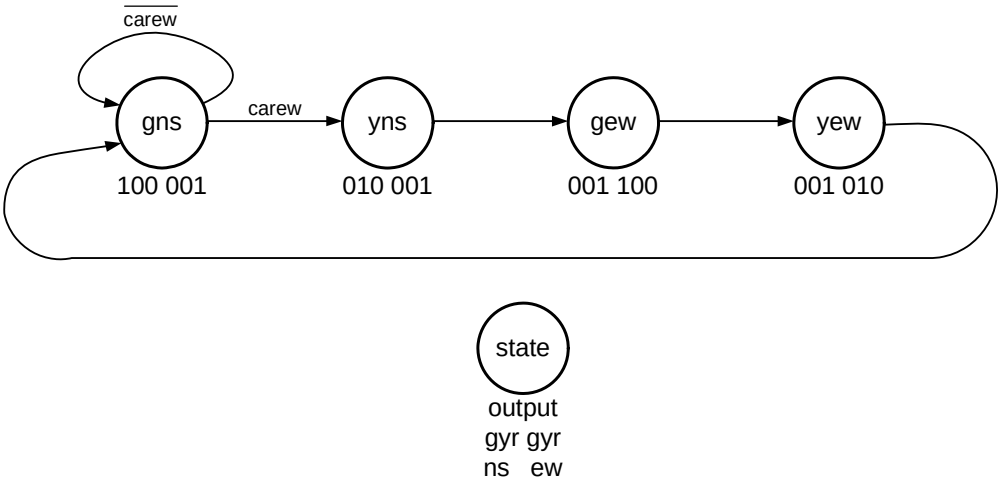


Figure 18.3: State diagram of simple traffic-light controller.

Address	State	car_ew	Next State	Output	Data
0000	GNS (000)	0	GNS (000)	100001	000100001
0001	GNS (000)	1	YNS (001)	100001	001100001
0010	YNS (001)	0	GEW (010)	010001	010010001
0011	YNS (001)	1	GEW (010)	010001	010010001
0100	GEW (010)	0	YEW (011)	001100	011001100
0101	GEW (010)	1	YEW (011)	001100	011001100
0110	YEW (011)	0	GNS (000)	001010	000001010
0111	YEW (011)	1	GNS (000)	001010	000001010

Table 18.1: State table for simple microcoded traffic-light controller

the input line (`car_ew`) goes high switching the ROM address from 00 to 01. This causes the microinstruction to switch from 041 to 141 which selects a next state of 1 (YNS) on the next clock. The machine then proceeds through states 2 (GEW) and 3 (YEW) before returning to state 0.

The beauty of microcode is that we can change the function of our FSM by changing only the contents of the ROM. Suppose for example that we would like to modify the FSM so that:

1. The light stays green in the east-west direction as long as `car_ew` is true.
2. The light stays green in the north south direction for a minimum of three cycles (states GNS1, GNS2, and GNS3).
3. After a yellow light, the lights should go red in both directions for one cycle before turning the new light green.

Table 18.2 shows the state table that implements these modifications. We split state GNS into three states and add two new states (RNS and REW). Note that the GEW state now tests `car_ew` and stays in GEW as long as it is true. The results of simulating our FSM of Figure 18.2 with this new microcode is shown in the waveform display of Figure 18.5.

18.2 Instruction Sequencing

A microcoded FSM can be made considerably more efficient by using a *sequencer* to generate the address of the next instruction. Performing instruction sequencing has two big advantages. First, for microinstructions that simply proceed to the next instruction, this instruction address can be generated with a counter, eliminating the need to store the address in the microcode memory. Second, by using logic to select or combine the different inputs, the microcode store can store a single microinstruction for each state, rather than having to store separate (and nearly identical) instructions for each possible combination of inputs.

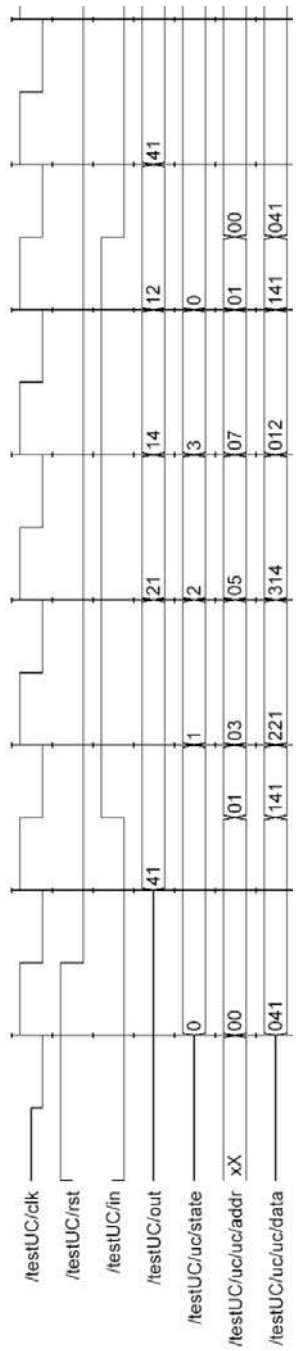


Figure 18.4: A waveform display showing a simulation of the microcoded FSM of Figure 18.2 using the microcode from Table 18.1.

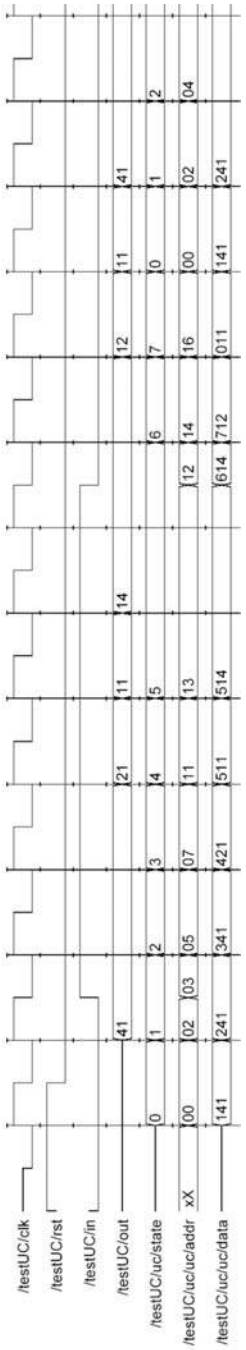


Figure 18.5: A waveform display showing a simulation of the microcoded FSM of Figure 18.2 using the microcode from Table 18.2.

Address	State	`car_ew`	Next State	Output	Data
0000	GNS1(000)	0	GNS2(001)	100001	001100001
0001	GNS1(000)	1	GNS2(001)	100001	001100001
0010	GNS2(001)	0	GNS3(010)	100001	010100001
0011	GNS2(001)	1	GNS3(010)	100001	010100001
0100	GNS3(010)	0	GNS3(010)	100001	010100001
0101	GNS3(010)	1	YNS (011)	100001	011100001
0110	YNS (011)	0	RNS (100)	010001	100010001
0111	YNS (011)	1	RNS (100)	010001	100010001
1000	RNS (100)	0	GEW (101)	001001	101001001
1001	RNS (100)	1	GEW (101)	001001	101001001
1010	GEW (101)	0	YEW (110)	001100	110001100
1011	GEW (101)	1	GEW (101)	001100	101001100
1100	YEW (110)	0	REW (111)	001010	111001010
1101	YEW (110)	1	REW (111)	001010	111001010
1110	REW (111)	0	GNS (000)	001001	000001001
1111	REW (111)	1	GNS (000)	001001	000001001

Table 18.2: State table for simple microcoded traffic-light controller

A review of the microcode from Table 18.2 shows the redundancy a sequencer can eliminate. Each instruction includes an explicit next-state field while all instructions either select themselves or the next instruction in sequence for the next state. Also, all instructions are duplicated for both input states, with only two having a small difference in the next state field. This overhead would be even higher if there were more than one input signal.

Adding a sequencer to our microcoded FSM is the first step from an FSM (where the next state is determined by a logic function) to a stored-program computer, where the next instruction is determined by interpreting the current instruction. With a sequencer, like a stored program computer, our microcoded machine *executes* microinstructions in sequence until a branch instruction redirects execution to a new address.

Figure 18.6 shows a microcoded FSM that uses an instruction sequencer. The state register here is replaced with a microprogram counter (μPC or uPC) register. At any point in time this register represents the current state by selecting the current microinstruction. With this design we have reduced the number of microinstructions in the microcode memory from 2^{s+i} to 2^s, that is from 2^i per state to 1 per state. The cost of this reduction is increasing the width of each microinstruction from $s+o$ bits to $s+o+b$ bits. Each microinstruction consists of three fields as shown in Figure 18.7: an o-bit field that specifies the current output, an s-bit field that specifies the address to branch to (the branch target), and a b-bit field that specifies a branch instruction.

With the instruction sequencer, inputs are tested by the branch logic as directed by a branch instruction from the current microinstruction. As a result

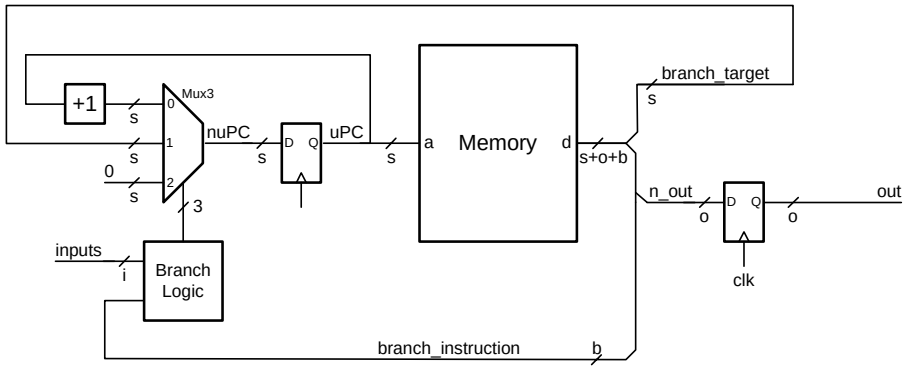


Figure 18.6: A microcoded FSM using an instruction sequencer. A multiplexer and incrementer compute the next microinstruction address (next state) based on a branch instruction and the input conditions.

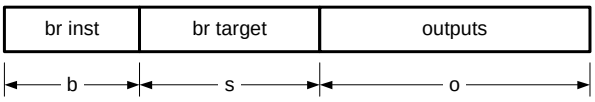


Figure 18.7: Microinstruction format for the microcoded FSM with instruction sequencer of Figure 18.6.

Encoding	OpCode	Description
000	NOP	Never branch, always proceed to uPC+1
001	B0	Branch on input 0. If `in[0]` branch to `br_upc` otherwise continue to uPC+1
010	B1	Branch on input 1.
011	BA	Branch any. Branch if either input is true.
100	BR	Always branch. Select `br_upc` as the next uPC regardless of the inputs.
101	BN0	Branch on not input 0. If `in[0]` is false, branch, otherwise continue to uPC+1.
110	BN1	Branch on not input 1.
111	BNA	Branch only if inputs 0 and 1 are both false.

Table 18.3: Branch instruction encodings.

of this test, the sequencer either branches (by selecting the branch target field as the next uPC) or doesn't (by selecting uPC+1 as the next UPC).

Consider an example with a two-bit input field. We can define a three-bit branch instruction `brinst` as follows.

```
branch = (brinst[0] & in[0] | brinst[1] & in[1]) ^ brinst[2] ;
```

Branch instruction bits 0 and 1 select whether we test input bits 0 or 1 (or either). Branch instruction bit 2 controls the polarity of the test. If `brinst[2]` is low, we branch if the selected bit(s) is high. Otherwise we branch if the selected bit(s) is low. Using this encoding of the branch instruction, we can perform the branches shown in Table 18.3.

Other encodings of the branch instruction are possible. A common n-bit encoding uses $n-1$-bits to select one of 2^{n-1} inputs to test and the remaining bit to select whether to branch on the selected input high or low. One of the inputs is set always high to allow the NOP and BR instructions to be created. For this encoding the branch signal would be:

```
branch = brinst[n-1] ^ in[brinst[n-2:0]] ;
```

The branch instructions created by this alternate encoding (for a 3-bit `brinst` and three inputs are listed in Table 18.4. To provide the NOP and BR instructions, we use a constant 1 for the fourth input. Here each branch instruction tests exactly one input, while in the encoding of Table 18.3 the instructions may test zero, one, or two inputs. Many other possible encodings beyond to the two presented here are possible.

A Verilog description of the microcoded FSM with an instruction sequencer is shown in Figure 18.8. The Verilog follows the block diagram of Figure 18.6 closely. One assign statement calculates signal `branch` which is true if the sequencer is to branch on the next cycle. A second asssign statement then calculates the next micro program counter (`nupc`) based on `branch` and `rst`.

Encoding	OpCode	Description
000	B0	Branch on input 0.
001	B1	Branch on input 1.
010	B2	Branch on input 2.
011	BR	Always branch. (Input 3 is the constant "1").
100	BN0	Branch on not input 0.
101	BN1	Branch on not input 1.
110	BN2	Branch on not input 2.
111	NOP	Never branch.

Table 18.4: Alternate branch instruction encodings.

```
module ucode2(clk,rst,in,out) ;
  parameter n = 2 ; // input width
  parameter m = 9 ; // output width
  parameter k = 4 ; // bits of state
  parameter j = 3 ; // bits of instruction

  input  clk, rst ;
  input  [n-1:0] in ;
  output [m-1:0] out ;

  wire   [k-1:0] nupc, upc ; // microprogram counter
  wire [j+k+m-1:0] uinst ;   // microinstruction word

  // split off fields of microinstruction
  wire [m-1:0] nxt_out ; // = uinst[m-1:0] ;
  wire [k-1:0] br_upc  ; // = uinst[m+k-1:m] ;
  wire [j-1:0] brinst  ; // = uinst[m+j+k-1:m+k] ;
  assign {brinst, br_upc, nxt_out} = uinst ;

  DFF #(k) upc_reg(clk, nupc, upc) ;  // microprogram counter
  DFF #(m) out_reg(clk, nxt_out, out) ; // output register
  ROM #(k,m+k+j) uc(upc, uinst) ; // microcode store

  // branch instruction decode
  wire branch = (brinst[0] & in[0] | brinst[1] & in[1]) ^ brinst[2] ;

  // sequencer
  assign nupc = rst ? {k{1'b0}} : branch ? br_upc : upc + 1'b1 ;
endmodule
```

Figure 18.8: Verilog description of a microcoded FSM with an instruction sequencer

Address	State	Br Inst	Target	NS LT EW	Data
0000	NS1	BLT (001)	LT1(0101)	100001001	0010101100001001
0001	NS2	BNEW (110)	NS1(0000)	100001001	1100000100001001
0010	EW1	NOP (000)		010001001	0000000010001001
0011	EW2	BEW (010)	EW2(0011)	001001100	0100011001001100
0100	EW3	BR (100)	NS1(0000)	001001010	1000000001001010
0101	LT1	NOP (000)		010001001	0000000010001001
0110	LT2	BLT (001)	LT2(0110)	001100001	0010110001100001
0111	LT3	BR (100)	NS1(0000)	001010001	1000000001010001

Table 18.5: Microcode for traffic light controller with sequencer of Figure 18.6.

To make the code more readable we use an assign statement to split out the three fields of the microinstruction into the output (`nxt_out`), the branch target (`br_upc`), and the branch instruction (`brinst`). Using these mnemonic names, rather than indexing fields of `uinst` makes the remainder of the code easier to follow.

Consider a slightly more involved version of our traffic-light controller that includes a left-turn signal as well as north-south and east-west signals. Table 18.5 shows the microcode . Here input 0 is `car_lt` and input 1 is `car_ew` so we rename our branches BLT (branch if `car_lt`), BNEW (branch if not `car_ew`) and so on.

The microcode of Table 18.5 starts in state NS1 where the light is green in the north-south direction. In this state the left-turn sensor is checked with a BLT to LT1. If `car_lt` is true, control transfers to LT1. Otherwise the uPC proceeds to the next state, NS2. In NS2 the microcode branches back to NS1 if `car_ew` is false (BNEW NS1). Otherwise control falls through to EW1 where the north-south light goes yellow. EW1 is always followed by EW2 where the east-west light is green. A BEW EW2 keeps the uPC in state EW2 as long as `car_ew` is true. When `car_ew` goes false, the uPC proceeds to state EW3 where the east-west light is yellow and a BR NS1 transfers control back to NS1. The left turn sequence (LT1, LT2, LT3) operates in a similar manner.

Waveforms from a simulation of the microcoded sequencer of Figure 18.8 running the microcode of Table 18.5 is shown in Figure 18.9. The fifth row from the top shows the microprogram counter `upc`. The machine is reset to `upc = 0` (NS1) advances to 1 (NS2), then back to 0 (NS1) before branching to 5 (LT1). It proceeds from 5 (LT1) to 6 (LT2) and remains in 6 until `car_lt` goes low. At that point it advances to 7 (LT3) and back to 0 (NS1). At this point `car_ew`=1, and the sequence followed is 0, 1, 2, 3 (NS1, NS2, EW1, EW2). The machine stays in 3 (EW2) until `car_ew` goes low and then proceeds to 4 (EW3) and back to 0 (NS1). The machine cycles between NS1 and NS2 a few times until both `car_ew` and `car_lt` go high at the same time. As the machine is in NS1 when this happens, `car_lt` is checked first and the uPC is directed to LT1.

Because the microcoded FSM of Figure 18.6 can only branch one way in each microinstruction, it takes two states (NS1 and NS2) to perform a three-

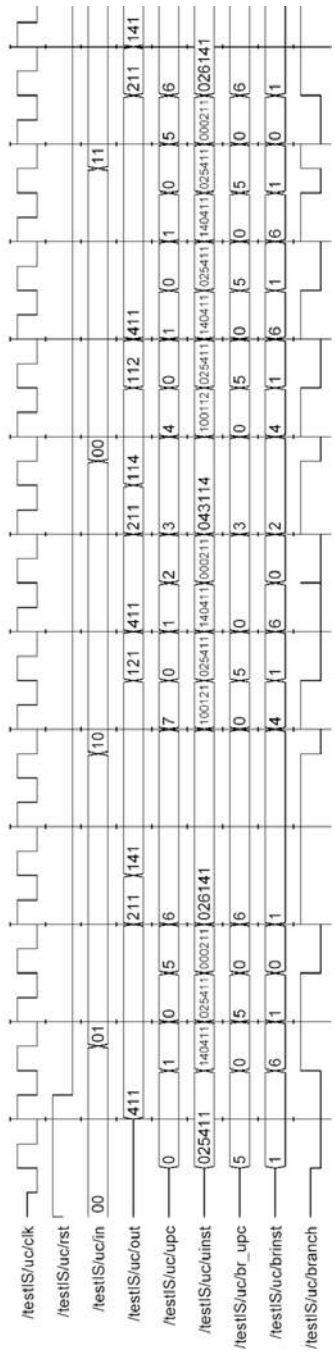


Figure 18.9: A waveform display showing a simulation of the microcoded FSM of Figure 18.2 using the microcode from Table 18.2.

Address	State	Br Inst	Target	NS LT EW	Data
0000	NS1	BNA (111)	NS1(0000)	100001001	1110000100001001
0001	NS2	BLT (001)	LT1(0100)	010001001	0010100010001001
0010	EW1	BEW (010)	EW1(0010)	001001100	0100010001001100
0011	EW2	BR (100)	NS1(0000)	001001010	1000000001001010
0100	LT1	BLT (001)	LT1(0100)	001100001	0010100001100001
0101	LT2	BR (100)	NS1(0000)	001010001	1000000001010001

Table 18.6: Alternate microcode for traffic light controller with sequencer of Figure 18.6.

way branch between staying with north-south, going to east-west, or going to left-turn. This results in two states with the lights green in the north-south direction (NS1 and NS2) and two states with the lights yellow in the north-south direction (EW1 and LT1). The real solution to this problem is to support a multi-way branch (which we will discuss below). However we can partially solve the problem in software by using the alternate microcode shown in Table 18.6.

In the alternate microcode of Table 18.6, the uPC stays in state NS1 as long as `car_ew` and `car_lt` are both false by using a BNA NS1 (branch on not any inputs to NS1). NS1 is now the only state with the lights green in the north-south direction. If any inputs are true, the uPC proceeds to state NS2 which is the single state with the lights yellow in the north-south direction. State NS2 tests the `car_lt` input and branches to state LT1 if true (BLT LT1). If `car_lt` is false, the uPC proceeds to EW1. The remainder of the machine is similar to that of Table 18.5 except that the EW and LT states have been renumbered.

Simulation waveforms for this alternate microcode are shown in Figure 18.10.

18.3 Multi-way Branches

As we saw in the previous section, using an instruction sequencer greatly reduces the size of our microcode memory but at the expense of restricting each state to have at most two next states (`upc+1` and `br_upc`). This restriction can be a problem if we have an FSM that has a large number of exits from a particular state. For example, in a microcoded processor, it is typical to branch to one of 10s to 100s of next states based on the *opcode* of the current instruction. Another multi-way branch is then needed on the *addressing mode* of the instruction. Implementing such a multi-way dispatch using the sequencer of Section 18.2 would result in very poor efficiency as n cycles would be required to test for n different opcodes.

We can overcome this limitation of two-way branches by using an instruction sequencer that supports multi-way branches as shown in Figure 18.11. This sequencer is similar to that of Figure 18.6 except that the branch target, `br_upc`, is generated from the branch instruction , `brinst`, and inputs rather than being provided directly from the microinstruction. With this approach, we can branch

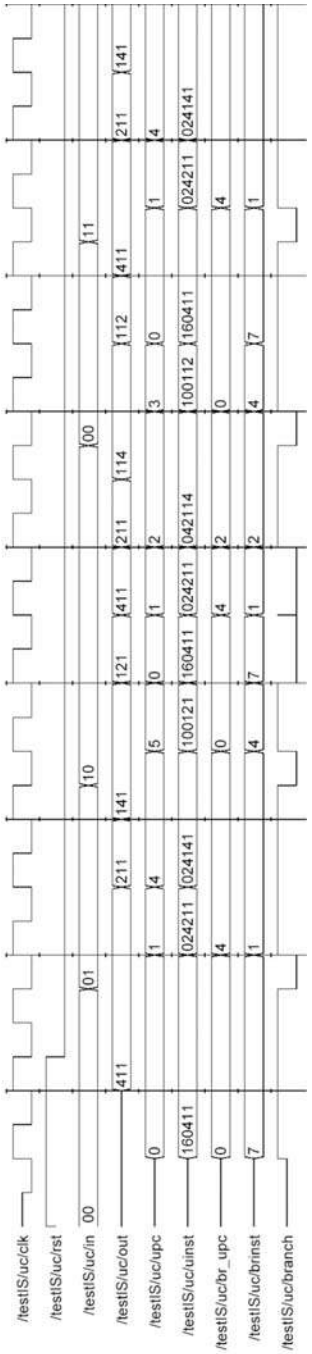


Figure 18.10: A waveform display showing a simulation of the microcoded FSM of Figure 18.2 using the microcode from Table 18.2.

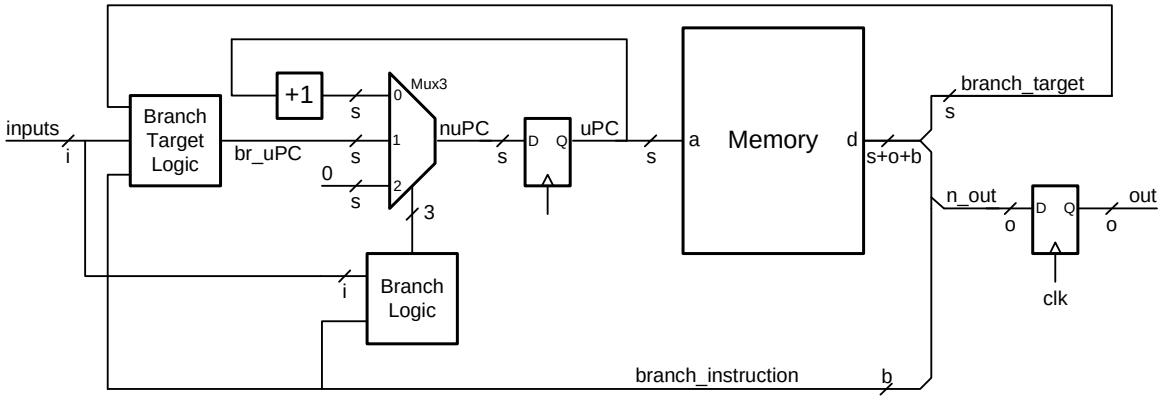


Figure 18.11: A microcoded FSM with an instruction sequencer that supports multi-way branches.

Encoding	Opcode	Description
BRx	00xx	Branch on condition x. (As in Table 18.4 includes BR)
BRNx	01xx	Branch on not condition x. (As in Table 18.4 includes NOP)
BR4	1000	Branch four ways. `nupc = br_upc + in`.

Table 18.7: Branch instructions for microcoded FSM supporting multi-way branches.

to up to 2^i next states (one for each input combination) from each state.

The branch instruction encodes not just the condition to test, but also how to determine the branch target. Table 18.7 shows one possible method of encoding multi-way branch instructions. The BRx and BRNx instructions are two-way branch instructions identical to those specified in Table 18.4. The BR4 instruction is a four-way branch that selects one of four adjacent states (from `br_upc` to `br_upc+3`) depending on the input.

To use the BR4 instruction requires some care in mapping states to microinstruction addresses and may require that some states are duplicated. Consider, for example, the state diagram of Figure 18.12. A mapping of this state diagram to microcode addresses for a machine with a four-way branch instruction that adds the input to the branch target is shown in Table 18.8. The four-way branch from X targets address 000, so we must lay out branch targets A1, B1, C1, and X in locations 000, 001, 010, and 011 respectively. In a similar manner we locate states so that the four-way branch from C1 targets address 100. Thus we must place states C2, C3, X, and C1 at locations 100, 101, 110, and 111 respectively. To make this work we need two copies of X, one at 011 and one at

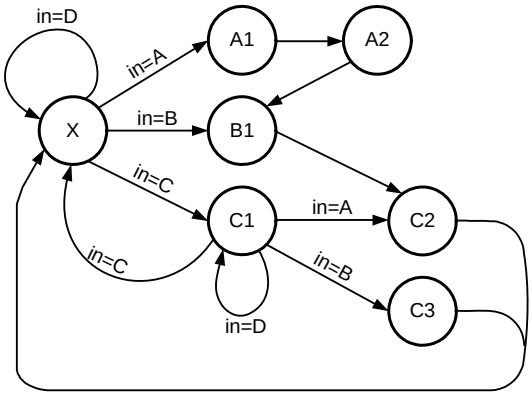


Figure 18.12: State diagram with two four-way branches.

Address	State	Branch Inst	Branch Target
000	A1	BR	A2
001	B1	BR	C2
010	C1	BR4	C2
011	X	BR4	A1
100	C2	BR	X
101	C3	BR	X
110	X'	BR4	A1
111	C1'	BR4	C2

Table 18.8: Mapping of state diagram of Figure 18.12 onto microcode addresses. States X and C1 are duplicated because they each appear in two four-way branches.

110 and two copies of C1 (at 010 and 111). When we duplicate a state in this manner we simply arrange for the two copies (e.g., X and X') to have identical behavior.

18.4 Multiple Instruction Types

So far we have considerd microcoded FSMs that update all output bits in every microinstruction. In general, most FSMs need to update only a subset of the outputs in a given state. Our traffic-light controller FSMs, for example, change at most one light on each state change. We can save bits of the microinstruction by modifying our FSM to update only a single output register in any given state. We waste other microinstruction bits by specifying a branch instruction and branch target in each microinstruction even though many microinstructions

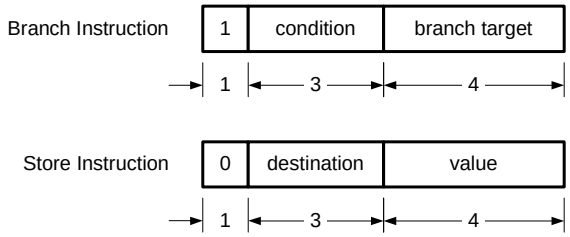


Figure 18.13: Instruction formats for a microcoded FSM with separate output and branch instructions.

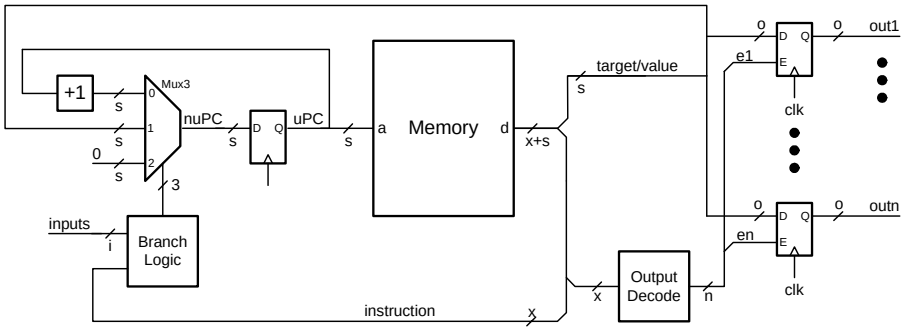


Figure 18.14: Block diagram of a microcoded FSM with output instructions.

always proceed to the next state and don't branch. We can save these redundant branch bits by only branching in some instructions and updating outputs in other instructions.

Figure 18.13 shows the instruction formats for a microcoded FSM with two microinstruction types: a branch instruction and a store (output) instruction. Each microinstruction is one or the other. A branch instruction, identified by a 1 in the left-most bit, specifies a branch condition and a branch target. When the FSM encounters a branch microinstruction, it branches (or doesn't) as specified by the condition and target. No outputs are updated. A store instruction, identified by a 0 in the left-most bit, specifies an output register and a value. When the FSM encounters a store microinstruction it stores the value to the specified output register and then proceeds to the next microinstruction in sequence. No branching takes place with a store instruction.

Figure 18.14 shows a block diagram of a microcoded FSM that supports the two instruction types of Figure 18.13. Each microinstruction is split into an x-bit instruction field and a s-bit value field. The instruction field holds the operation-code (opcode) bit (the left-most bit that distinguishes between branch and store) and either the condition (for a branch) or the destination (for

a store). The value field holds either the branch target (for a branch) or the new output value (for a store).

The instruction sequencer of Figure 18.14 is identical to that of Figure 18.6 except that the branch logic always selects the next microinstruction `upc+1` when the current microinstruction is a store instruction.[1] The major difference is with the output logic. Here a decoder enables at most one output register to receive the value field on a store instruction.

A Verilog description for the microcode engine with two instruction types of Figure 18.14 is shown in Figure 18.15. Here the microinstruction is split into an opcode (0=store, 1=branch), an instruction (destination for store, condition for branch), and a value. For a store, the destination is decoded into a one-hot enable vector, `e`, that is used to enable the value to be stored into one of the three output registers (ns=0, ew=1, lt=2) or to load the timer (dest = 3). For a branch, `inst[2]` determines the polarity of the branch, and the low two bits, `inst[1:0]`, determine the condition to be tested (lt=0, ew=1, lt|ew=2, timer=3).

Table 18.9 shows the microcode for a more sophisticated traffic-light controller programmed on the microcode engine of Figure 18.15. The first three states load the three output registers with RED in the east-west and left-turn registers and GREEN in the north-south register. Next states NS1 and NS2 wait for eight cycles by loading the timer and waiting for it to signal done. State NS4 then waits for either input. The north-south light is set to YELLOW in NS5. NS6 and NS7 then set the timer and wait for it to be done before advancing to NS8 where the north-south light is set to RED. If the left-turn input is true NS9 branches to LT1 to sequence the left-turn light. Otherwise the east-west lights are sequenced in states EW1 through EW9. Waveforms from a simulation of this microcode are shown in Figure 18.16.

18.5 Microcode Subroutines

The state sequences in Table 18.9 are very repetitive. The NS, EW, and LT sequences perform largely the same actions. The only salient difference is the output register being written. Just as we shared common state sequences by factoring FSMs in Chapter 17, we can share common state sequences in a microcoded FSM by supporting *subroutines*. A subroutine is a sequence of instructions that can be called from several different points and after exiting returns control to the point from which it was called.

Figure 18.17 shows the block diagram of a microcode engine that supports one level of subroutines. This machine is identical to that of Figure 18.14 except for two differences: (a) a return uPC register, `rupc` and associated logic have been added to the sequencer, and (b) a select register and associated logic have been added to the output section.

[1] One can just as easily add multiple instruction types and output registers to a FSM that supports multi-way branches (Figure 18.11).

```
module ucodeMI(clk,rst,in,out) ;
  parameter n = 2 ; // input width
  parameter m = 9 ; // output width
  parameter o = 3 ; // output sub-width
  parameter k = 5 ; // bits of state
  parameter j = 4 ; // bits of instruction

  input  clk, rst ;
  input  [n-1:0] in ;
  output [m-1:0] out ;

  wire   [k-1:0] nupc, upc ; // microprogram counter
  wire [j+k-1:0] uinst ;   // microinstruction word
  wire done ; // timer done signal

  // split off fields of microinstruction
  wire opcode ; // opcode bit
  wire [j-2:0] inst ;  // condition for branch, dest for store
  wire [k-1:0] value ; // target for branch, value for store
  assign {opcode, inst, value} = uinst ;

  DFF #(k) upc_reg(clk, nupc, upc) ;  // microprogram counter
  ROM #(k,k+j) uc(upc, uinst) ; // microcode store

  // output registers and timer
  DFFE #(o) or0(clk, e[0], value[o-1:0], out[o-1:0]) ;     // NS
  DFFE #(o) or1(clk, e[1], value[o-1:0], out[2*o-1:o]) ;   // EW
  DFFE #(o) or2(clk, e[2], value[o-1:0], out[3*o-1:2*o]) ; // LT
  Timer #(k) tim(clk, rst, e[3], value, done) ;            // timer

  // enable for output registers and timer
  wire [3:0] e = opcode ? 4'b0 : 1<<inst ;

  // branch instruction decode
  wire branch = opcode ? (inst[2] ^ (((inst[1:0] == 0) & in[0]) | // BLT
     ((inst[1:0] == 1) & in[1]) | // BEW
     ((inst[1:0] == 2) & (in[0]|in[1])) | //BLE
     ((inst[1:0] == 3) & done))) // BTD
                       : 1'b0 ; // for a store opcode

  // microprogram counter
  assign nupc = rst ? {k{1'b0}} : branch ? value : upc + 1'b1 ;
endmodule
```

Figure 18.15: Verilog description of a microcoded FSM with two instruction types.

Address	State	Instruction	Value	Data
00000	RST1	SLT (0010)	RED 001	001000001
00001	RST2	SEW (0001)	RED 001	000100001
00010	NS1	SNS (0000)	GREEN 100	000000100
00011	NS2	STIM(0011)	TGRN 01000	001101000
00100	NS3	BNTD(1111)	NS3 00100	111100100
00101	NS4	BNLE(1110)	NS4 00101	111000101
00110	NS5	SNS (0000)	YELLOW 010	000000010
00111	NS6	STIM(0011)	TYEL 00011	001100011
01000	NS7	BNTD(1111)	NS7 01000	111101000
01001	NS8	SNS (0000)	RED 001	000000001
01010	NS9	BLT (1000)	LT1 10100	100010100
01011	EW1	STIM(0011)	TRED 00010	001100010
01100	EW2	BNTD(1111)	EW2 01100	111101100
01101	EW3	SEW (0001)	GREEN 100	000100100
01110	EW4	STIM(0011)	TGRN 01000	001101000
01111	EW5	BNTD(1111)	EW5 01111	111101111
10000	EW6	SEW (0001)	YELLOW 010	000100010
10001	EW7	STIM(0011)	TYEL 00011	001100011
10010	EW8	BNTD(1111)	EW8 10010	111110010
10011	EW9	BTD (1011)	RST2 00001	101100001
10100	LT1	STIM(0011)	TRED 00010	001100010
10101	LT2	BNTD(1111)	LT2 10101	111110101
10110	LT3	SLT (0010)	GREEN 100	001000100
10111	LT4	STIM(0011)	TGRN 01000	001101000
11000	LT5	BNTD(1111)	LT5 11000	111111000
11001	LT6	SLT (0010)	YELLOW 010	001000010
11010	LT7	STIM(0011)	TYEL 00011	001100011
11011	LT8	BNTD(1111)	LT8 10010	111111011
11100	LT9	BTD (1011)	RST1 00000	101100000

Table 18.9: Microcode to implement traffic-light controller on FSM with two instruction types.

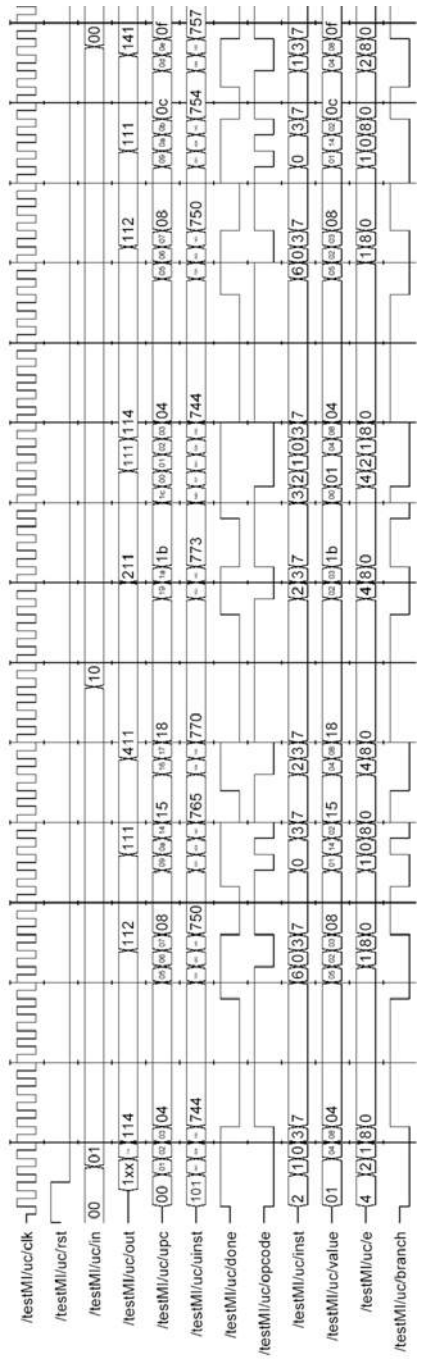


Figure 18.16: A waveform display showing a simulation of the microcoded FSM of Figure 18.15 using the microcode from Table 18.9.

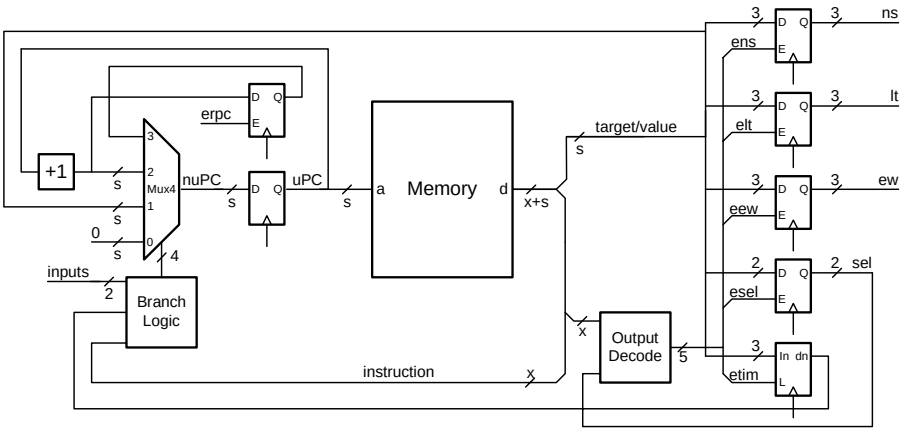


Figure 18.17: Microcoded FSM with support for one level of subroutines.

The `rupc` register is used to hold the `upc` to which a subroutine should branch when it completes. When a subroutine is called, the branch target is selected as the next `upc` and `upc+1`, the address of the next instruction in sequence is saved in the `rupc` register. A special branch instruction CALL is used to cause the enable line to the `rupc` register, `erpc`, to be asserted. When the subroutine is complete, it returns control to the saved location using another special branch instruction RET to select the `rupc` as the source of the next `upc`.

The select register is used to allow the same state sequence to write to different output registers when called from different places. A two-bit register identifier (ns=0, ew=1, lt=2) can be stored in the select register. A special store instruction SSEL can then be used to store to the register specified by the select register (rather than by the destination bits of the instruction). Thus, the main program can store 0 (ns) into the select register and then call a subroutine to sequence the north-south lights on and off. The program can then store 1 (ew) into the select register and call the same subroutine to sequence the east-west lights on and off. The same subroutine can sequence different lights because it performs all of its output using the SSEL instruction.

EXAMPLE CODE HERE

18.6 A Simple Computer

18.7 Bibliographic Notes

18.8 Exercises

microcode sequence detector

Chapter 19

Sequential Examples

This chapter gives some additional examples of sequential circuits. We start with a simple FSM that reduces the number of 1s on its input by a factor of 3 to review how to draw a state diagram from a specification and how to implement a simple FSM in Verilog. We then implement an SOS detector to review factoring of state machines. Next we revisit our Tic-Tac-Toe game from Section 9.4 and build a datapath sequential circuit that plays a game against itself useing the combinational move generator we previously developed. We illustrate the use of table-driven sequential circuits and composing circuits from sequential building blocks like counters and shift registers by building a Huffman encoder and decoder. The encoder uses table lookup along with a counter and shift register while the decoder traverses a tree data structure stored in a table.

19.1 A Divide-by-Three Counter

In this section we will design a finite state machine that outputs a high signal on the output for one cycle for each three cycles the input has been high. More specifically, our FSM has a single input `in` and a single output `out`. When `in` is detected high for the third cycle (and sixth, ninth, etc...) `out` will go high for exactly one cycle. This FSM divides the *number* of pulses on the input by three. It does not divide the binary number represented by the input by three.

A state diagram for this machine is shown in Figure 19.1. At first it may seem that we can implement this machine with three states. Four, however are required. We need states A to D to distinguish having seen the input high for 0, 1, 2, or 3 cycles so far. The machine resets to state A. It sits in this state until the input is high on a rising clock edge at which time it advances to state B. The second high input takes the machine to C, and the third high input takes the machine to D where the output goes high for one cycle. We can't simply have this third hign input take us back to A because we need to distinguish having seen three cycles of high input — in which case the output goes high —

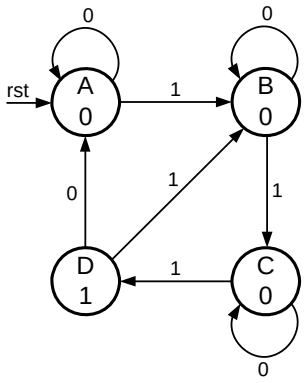


Figure 19.1: State diagram for a divie-by-three counter FSM. The four states represent having seen the input high for 0, 1, 2, and 3 cycles so far.

from having seen zero cycles of high input.[1]

The FSM always exits state D after one cycle. The input during this cycle determines the next state. If the input is low, the machine advances to state A to wait for three more high inputs before the next output. If the input is high, this counts as one of the three high inputs, so the machine advances to state B to wait for two more.

A verilog description of this divide-by-three FSM is shown in Figure 19.2. A *case* statement is used to implement the next-state function, including reset. A single assign statement implements the output function, making the output high in state D. The define statements used to define the states and their width are not shown. Waveforms from simulating this verilog model are shown in Figure 19.3.

19.2 An SOS Detector

Morse code, once widely used for telegraph and radio communication, encodes the alphabet, numbers, and a few punctuation marks into an on/off signal as patterns of dots and dashes. Spaces are used to separate symbols. A *dot* is a short period of *on*, a *dash* is a long period of *on*. Dots and dashes within a symbol are separated by short periods of *off*, and a space is a long period of *off*. The universal distress code, SOS, in Morse code is three dots (S), a space, three dashes (O), a space, and three dots again (the second S).

Consider the task of building a finite-state machine to detect when an SOS is received on the input. We assume that a *dot* is represented by the input being high for exactly one cycle, a *dash* is represented by the input being high

[1] See Exercise 19–3 for an approach that does require only three states.

```
//------------------------------------------------------------------------
//Divide by 3 FSM
//  in - increments state when high
//  out - goes high one cycle for every three cycles in is high
//    it goes high for the first time on the cycle after the third cycle
//    in is high.
//------------------------------------------------------------------------
module Div3FSM(clk, rst, in, out) ;
  input clk, rst, in ;
  output out ;

  wire out ;
  wire [`AWIDTH-1:0] state ; // current state
  reg  [`AWIDTH-1:0] next ;  // next state

  // instantiate state register
  DFF #(`AWIDTH) state_reg(clk, next, state) ;

  // next state function
  always @(state or rst or in) begin
    case(state)
      `A: next = rst ? `A : (in ? `B : `A) ;
      `B: next = rst ? `A : (in ? `C : `B) ;
      `C: next = rst ? `A : (in ? `D : `C) ;
      `D: next = rst ? `A : (in ? `B : `A) ;
      default: next = `A ;
    endcase
  end

  // output function
  assign out = (state == `D) ;
endmodule
```

Figure 19.2: Verilog description of the divide-by-three counter.

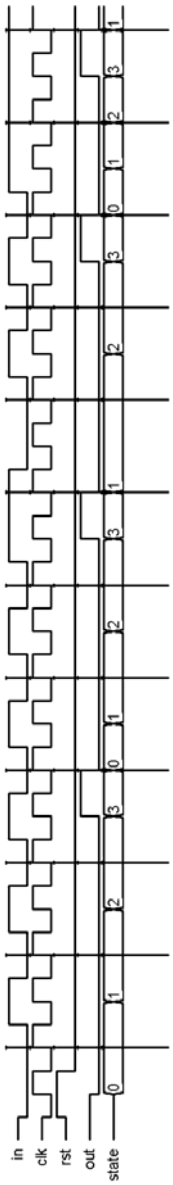


Figure 19.3: Waveforms from simulating the divide-by-three counter.

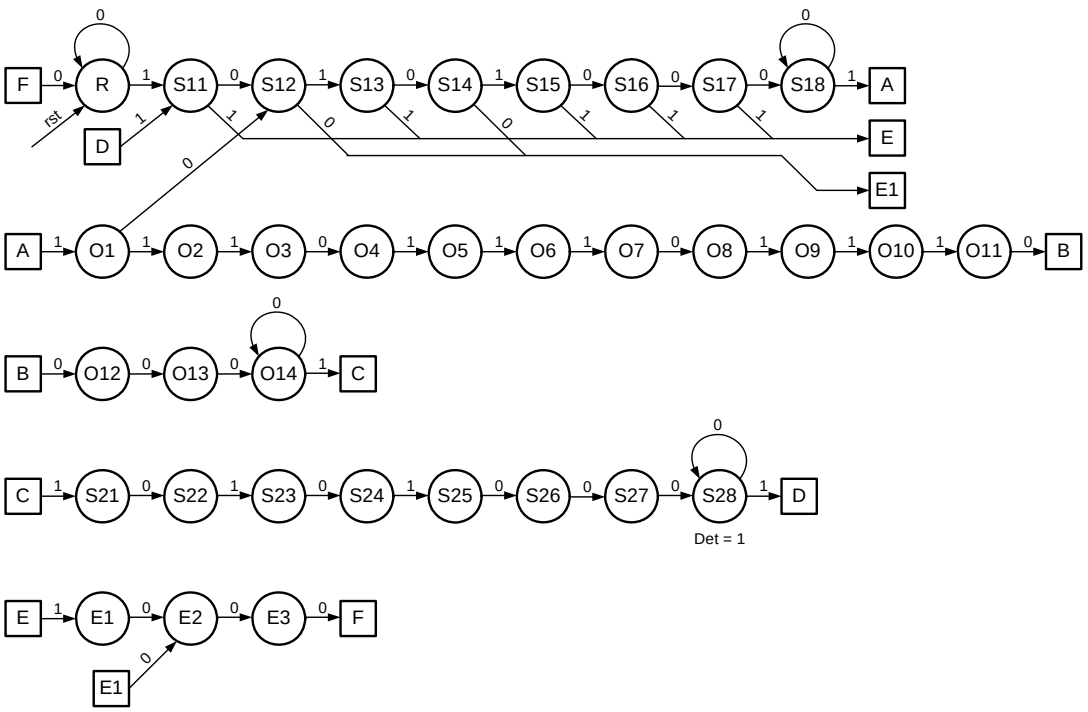


Figure 19.4: A state diagram for an SOS detector realized as a single, flat FSM. The square boxes indicate connections.

for exaclty three cycles, that dots and dashes within a symbol are separated by the input being low for exactly one cycle, and that a *space* is represented by the input being low for three or more cycles. Note that the input going either high or low for exactly two cycles is an illegal condition. With this set of definitions, one legal SOS string is 10101000111011101110001010100 0.

We can build an SOS detector as a single, flat state machine as shown with the state diagram of Figure 19.4. The FSM resets to state R. States S11 to S18 detect the first "S" and the associated space. States O1 to O11 detect the "O" and states O12 to O14 detect the space following the "O". Finally states S21 to S28 detect the second "S" and the subsequent space. State 28 outputs a "1" to indicate that SOS has been detected.

For clarity, many transitions are omitted from Figure 19.4. The transitions along the horizontal path from state R through state S28 represent the transitions that occur when an SOS is detected. If at any point along this path a 1 is detected when a 0 is expected the machine transitions to state E1. Similarly, if a 0 is detected when we expect a 1, the machine transitions to state E2. These transitions are shown for the first row (via boxes E and E1) and then omitted

to avoid cluttering the figure. States E1 to E3 are error handling states that wait for a space after an error condition and then restart the detection.

The transition from O1 to S12 handles the case where the input includes the string SSOS. After detecting the first S, we are expecting an O but instead receive a second S. If we were to transition to state E2 on receiving a 0 in O1, we would miss this second S and hence the SOS. Instead we must recognize the *dot* and go to state S12.

The transition from S28 to S11 (via the box labeled D) is needed to allow back-to-back SOSs with minimum sized spaces to be detected. After detecting SOS, including the subsequent space, in state S28, the next 1 may be the first dot of the next SOS and must be recognized by going to state S11.

While the flat FSM of Figure 19.4 works, it is not a very good solution for a number of reasons. First, it is not modular. If we were to change the definition of a dot to be the input going high for 1 or 2 cycles, the flat machine woudl need to be changed in eight places (every place a dot is recognized). Similar global change would be needed to accomodate a change to the definition of a dash or a space. Also, the machine would need to be complete reworked if the sequence we are detecting is different than SOS, say ABC. Second, the machine is large, 34 states, and would become even larger with more flexible definitions of dots and dashes. Finally, some aspects of the machine, like the transition from O1 to S12 are subtle.

The SOS machine is a perfect candidate for *factoring*. We can build FSMs to detect dots, dashes, and spaces, and then use the outputs of these FSMs to build FSMs that detect S and O. Finally, a simple top-level FSM detects SOS. A block diagram for a factored version of our SOS detection FSM is shown in Figure 19.5. The input bit stream (*sequence*) is input to three element detecting FSMs - *Dot*, *Dash*, and *Space*. Each of these FSMs has two outputs, one that indicates when the element has been detected, and one that indicates that the current input sequence *could be* part of that element. For example, the *Dot* FSM outputs *isDot* when a dot is detected, and *cbDot* when the current sequence could be a dot, but we need additional input before deciding.

The six signals out of the three element detectors feed a pair of character detectors, one each for S and O. Like the element detectors, each character detector also has an *is* and a *could be* output. The four signals out of the two character detectors are input to a top-level SOS FSM that indicates when SOS is detected.

Figure 19.6 shows the three FSMs that detect the elements Dot, Dash, and Space. The Dot FSM resets to state 0. Upon detecting a 1 on the input it indicates that the current sequence *could be* a dot by asserting output *cb* and transitions to state *Dot*. In state *Dot*, a 0 on the input results in a dot being detected with the *is* output asserted and returns the machine to state 0. Note that when the *is* output is asserted, the *cb* output is also asserted. If a 1 is detected in state *Dot*, the machine goes to state 1 to wait for a zero. The Dash and Space machines operate in a similar manner.

The character FSM for the character S is shown in Figure 19.7. The machine resets to state OTH (other) which is also the target of the default (def) tran-

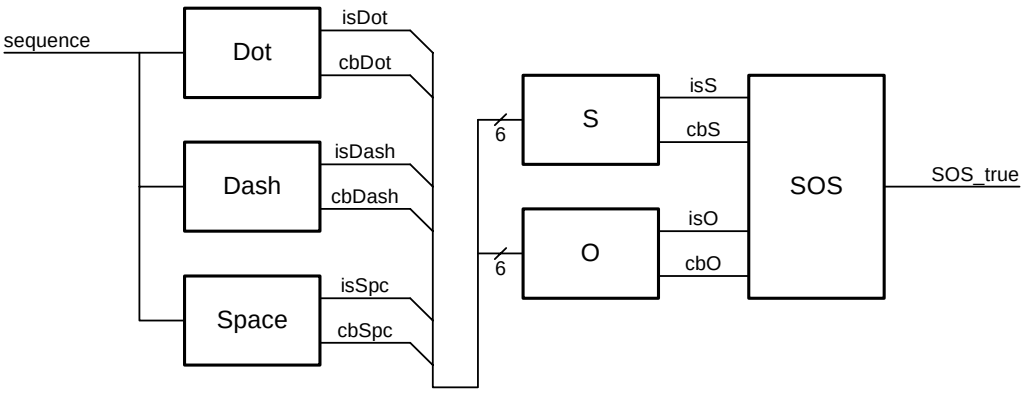


Figure 19.5: A block diagram of a factored SOS detector. The first rank of FSMs detects dots, dashes, and spaces. The second rank detects Ss and Os. The final SOS FSM detects the sequence SOS. Each sub-machine has two outputs: one indicates when the desired symbol has been detected (e.g., *isS*). The other indicates when the current sequence *could be* a prefix of the desired symbol (e.g., *cbS*).

sition which covers unexpected inputs. Upon detecting a space, the machine enters state SPC. Detecting the first dot moves the machine to state D1, and subsequent dots move the machine to states D2, and D3. Detecting a space in state D3 returns the machine to state SPC and asserts *is* to signal that an S has been detected.

If at any point during the sequence from SPC through D1, D2, and D3 and back to SPC the input could not be the element being waiting for (e.g., if *cbDot* is false in state D1), then the machine returns to state OTH. This is why we need the *could-be* outputs on the element detectors. They allow us to detect illegal elements between the elements we are looking for. Consider, for example the input sequence 00010110101000. The machine detects the space 000, and the first dot 10, but then returns to state OTH on the illegal element 110 because *cbDot* falls when the second 1 is detected. If we just wait for *isDot* we would erroneously determine that this sequence is an S because it has three dots. Without monitoring *cbDot* we wouldn't see that the three dots are not contiguous, and hence not an S.

The main SOS detecting FSM, shown in Figure 19.8, contains only three states. It waits in state ST (start) until an S is detected when it moves to state S1. From state S1 if an O is detected it moves to state O. However, if at any point in S1 input *cbO* goes false, indicating an illegal sequence between the S and the O, the machine returns to ST. If the machine detects a second S while in state O, it asserts its *is* output, detecting an SOS, and returns to ST. If input *cbS* goes false while instate O, detecting an illegal sequence between the O and

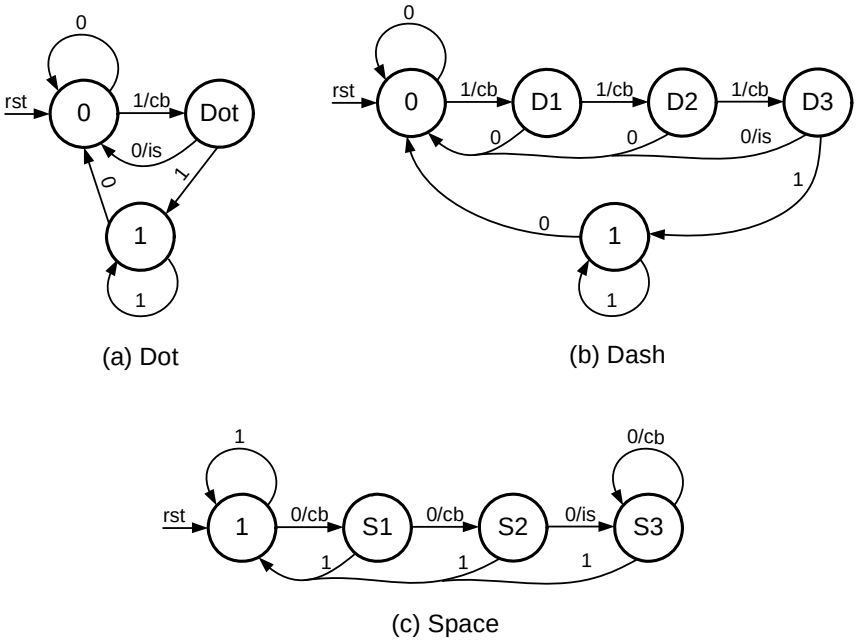


Figure 19.6: Element finite state machines. (a) Dot, (b) Dash, and (c) Space. Each outputs when the current sequence *could be* (cb) the respective element and when the element has been detected (is).

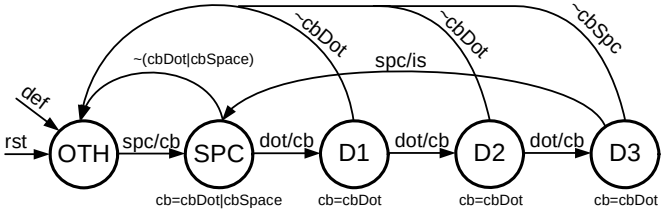


Figure 19.7: State diagram for S-detecting FSM.

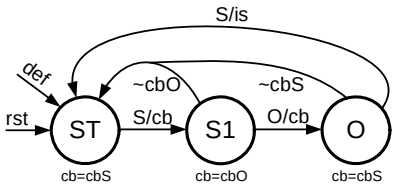


Figure 19.8: State diagram main SOS detecting FSM.

the S, the machine returns to ST without detecting SOS.

Waveforms showing operation of the factored SOS detector are shown in Figure 19.9. The waveforms show two good SOS detections with a SOT (T is a single dash) between them. Note the *cb* and *is* waveforms for each element, for the characters S and O, and for SOS itself. On the second 1 of the T's dash, *cbDot* falls causing *cbS* and *cbSOS* to fall in turn (combinationally).

Factoring the SOS detector gives us a much simpler system that is far easier to modify and maintain. Instead of 34 states in one brittle, monolithic state machine, we have a total of 20 states divided over six small, simple FSMs. The largest single FSM in the factored machine has a total of five states. Should we modify our specification to change the definition of a dot to be one or two 1s in a row, we would make one simple change to the Dot FSM.[2]

19.3 A Tic-Tac-Toe Game

In Section 9.4 we designed a combinational module that generated moves for the game of Tic-Tac-Toe. In this section we will use this module as a component in a sequential system that plays a game of Tic-Tac-Toe against itself.

A block diagram of the system is shown in Figure 19.10. The state in the system resides in the three registers on the left side of the figure. The 9-bit `Xreg` and `Oreg` hold bit maps that reflect the current positions of Xs and Os respectively. The 1-bit `xplays` register is true if X plays next and false if Y plays next. Reset is not shown in the figure. `Xreg` and `Oreg` reset to all zeros. `xplays` resets to true.

When it is X's turn to play (`xplays` = 1) the multiplexers direct `Xreg` to the `xin` input of the move generator and `Oreg` to the `oin` input. The move generator generates the next move on `xout` and this is ORed with the current X position to generate the new x position that is stored back in `Xreg` at the end of the cycle. The write to `Xreg` is enabled by `xplays`. When `xplays` is false, the multiplexers switch the move generator inputs to generate a move for O and this move is written back to `Oreg` at the end of the cycle.

A Verilog description of the tic-tac-toe playing system is shown in Figure 19.11. After the declaration of the three state registers an assignment

[2]For some practice modifying this factored SOS detector, see Exercises 19–4 and 19–5.

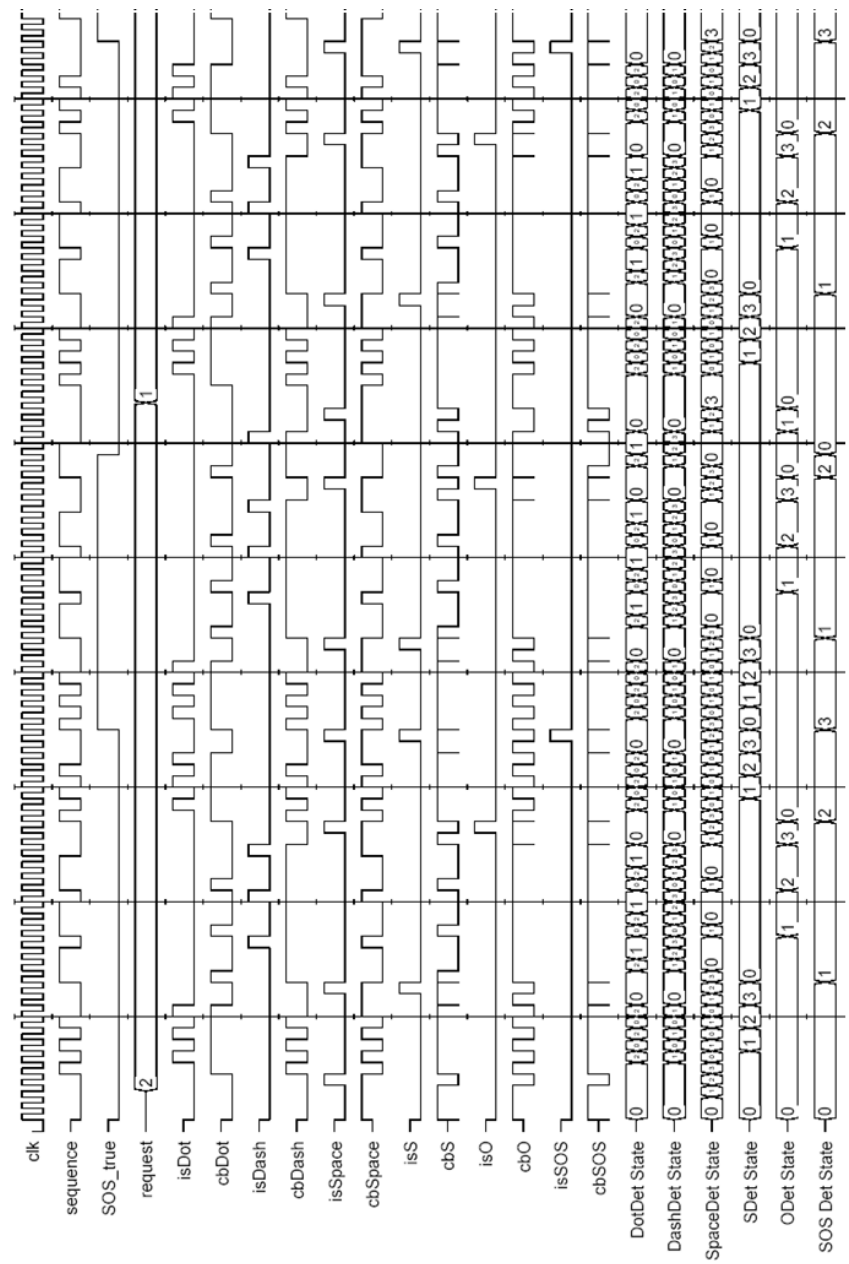


Figure 19.9: Waveforms showing operation of factored SOS detector.

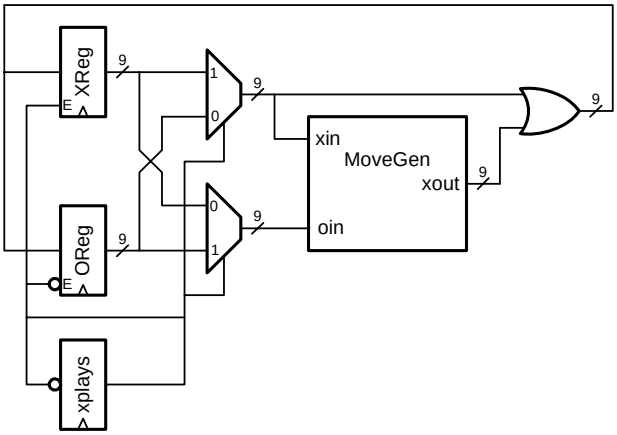


Figure 19.10: Block diagram of a tic-tac-toe playing system using the move generation module of Section 9.4.

statement toggles `xplays` each cycle. The input multiplexers are implemented as select statement in the argument list for the move generator. Two assign statements compute the next state for `Xreg` and `Oreg`. The OR of the move with the previous state is included in these statements.

19.4 A Huffman Encoder/Decoder

A Huffman code is an *entropy* code that encodes each symbol of an alphabet with a bit string. Frequently used symbols are encoded with short bit strings while infrequently used symbols are encoded with longer bit strings. To be able to distinguish short bit strings from the first parts of longer bit strings, each short bit string must not be used as a prefix for any longer bit string. The net result is *data compression*; that is, a typical sequence of symbols is encoded into fewer bits than would be required if all symbols were encoded with the same number of bits.

19.4.1 Huffman Encoder

For this example we will build a Huffman encoder and decoder for the letters of the alphabet, A-Z. The input to our encoder is a 5-bit code where A = 1 and Z = 26.[3] To prevent input characters from arriving faster than our encoder can handle them, our encoder generates an input ready *(irdy)* signal that indicates when the encoder is ready for the next input character. The output is a serial

[3]This corresponds to the low 5-bits of the ASCII code for both upper-case and lower-case letters.

```
//------------------------------------------------------------------------------
// Sequential Tic-Tac-Toe game
//   Plays a game against itself
//------------------------------------------------------------------------------
module SeqTic(clk, rst, xreg, oreg, xplays) ;
  input clk, rst ;
  output [8:0] xreg, oreg ;
  output xplays ;

  wire [8:0] nxreg, noreg, move ; // next state
  wire nxplays ; // next state

  // state
  DFF #(9) x(clk, nxreg, xreg) ;
  DFF #(9) o(clk, noreg, oreg) ;
  DFF xp(clk, nxplays, xplays) ;

  // x plays first, then alternate
  assign nxplays = rst ? 1 : ~xplays ;

  // move generator - mux inputs so current player is x
  TicTacToe movGen(xplays ? xreg : oreg, xplays ? oreg : xreg, move) ;

  // update current player
  assign nxreg = rst ? 0 : (xreg | (xplays ? move : 0)) ;
  assign noreg = rst ? 0 : (oreg | (xplays ? 0 : move)) ;
endmodule
```

Figure 19.11: Verilog description of the Tic-Tac-Toe playing system.

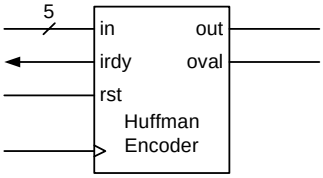


Figure 19.12: Block diagram symbol for our Huffman Encoder. The encoder accepts a 5-bit character on *in* each time *ready* goes high and generates a bit serial output stream on *out* when *valid* is high.

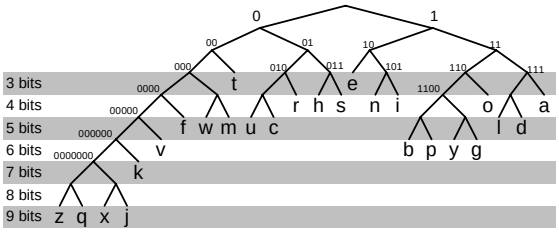


Figure 19.13: Huffman code for the alphabet shown as a binary tree. Starting at the root, each branch to the left denotes a 0 and each branch to the right a 1. Hence the letter W which is reached by a sequence of left, left, left, right, left is encoded as 00010.

bit stream of the encoded characters. To make it easy for the decoder to find the start of the bit stream, our encoder also generates an output valid *(oval)* signal that signals when the bits in the output stream are valid. A block diagram symbol for our encoder showing inputs and outputs is shown in Figure 19.12.

Figure 19.13 shows the code we will use for our example in tree form. The path from the root of the tree to a character gives the code for that character. The letter E, for example, is reached by going right, left, left and hence is represented by the three-bit string 100. The letter J is reached by going left 7 times and then right twice is represented by the nine-bit string 000000011. Very frequently occuring characters like T and E are represented with just three bits. Very infrequently occurring characters like Z, Q, X, and J are represented with nine bits. Representing the code as a tree makes it clear that a short string used to represent one symbol is not a prefix of a longer string used to represent another symbol since each leaf node of the tree terminates the path used to reach that leaf.

A block diagram of a Huffman encoder is shown in Figure 19.14. A five-bit input register holds the current symbol and is loaded with a new symbol each time *irdy* is asserted. The symbol is used to address a ROM that stores the string and string length associated with each symbol. For example, the ROM

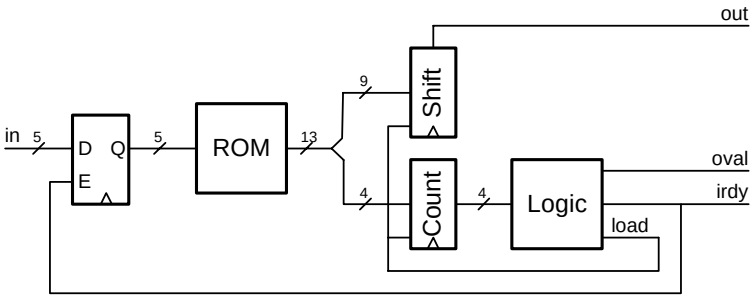


Figure 19.14: Block diagram of Huffman encoder. A ROM stores the length and string for each character. A counter counts down the length while a shift shifts out the string.

stores string 0011 001000000 for the symbol T. This indicates that the string representing T is three bits in length and the three bits are 001. Because the maximum length string is nine bits, we use four bits to represent the length and nine bits to represent the string. Strings shorter than nine bits are left-aligned in the nine-bit field so they can be shifted out to the left.

One cycle after a new symbol is loaded into the input register by *irdy*, signal *load* is asserted to load the length and string associated with that symbol into a counter and a shift register. The counter then counts down while the shift register shifts bits onto the output. When the counter reaches a count of 2 (second to last bit), *irdy* is asserted to load the next symbol into the input register, and when the counter reaches a count of 1 (last bit of this symbol), *load* is asserted to load the length and string for the next symbol into the counter and shift register.

A verilog implementation of our Huffman encoder is shown in Figure 19.15. We use the up/down/load counter from Section 16.1.2 to implement our counter and the left/right/load shift register from Section 16.2.2 to implement our shifter. Note that while we don't use the up function of the counter or the right function of the shift register, this will still result in an efficient implementation as the synthesizer will optimize away the unused logic. The table is implemented in module `HuffmanEncTable` (not shown) which is coded as a large case statement.

The control logic for the Huffman encoder is straightforward. One line of code asserts *irdy* on a count of two or zero - the latter is needed to load the first symbol after a reset. A DFF then delays *irdy* one cycle to generate signal *dirdy* that is used to load the counter and shifter. One line of code and a DFF are used to keep *oval* low after reset until the first time *dirdy* is asserted.

Figure 19.16 shows the result of simulating the Huffman encoder on the input string "THE". The three symbols 14 (T), 08 (H), and 05 (E) in hexadecimal are shown on *in*. The resulting output is 001 (T), 0110 (H), and 100 (E) shifted

```
//------------------------------------------------------------------------
// Huffman Encoder
//   in - character 'a' to 'z' - must be ready
//   irdy - when high accepts the current input character
//   out - bit serial huffman output
//   oval - true when output holds valid bits
//
//   input character accesses a table with each entry having
//   length[4], bits[9]
//------------------------------------------------------------------------
module HuffmanEncoder(clk, rst, in, irdy, out, oval) ;
  input clk, rst ;
  input [4:0] in ;
  output irdy, out, oval ;

  wire [3:0] length, count ;
  wire [8:0] bits, obits ;
  wire [4:0] char ;
  wire dirdy ; // irdy delayed by one cycle - loads count and sr
  wire oval ;

  // control
  wire out  = obits[8] ; // MSB is output
  wire irdy = ~rst & ((count == 2) | (count == 0)) ; // 0 count for reset
  wire noval = ~rst & (dirdy | oval) ;  // output valid cycle after load

  // instantiate blocks
  UDL_Count2 #(4) cntr(.clk(clk), .rst(rst), .up(1'b0), .down(~dirdy),
.load(dirdy), .in(length), .out(count)) ;
  LRL_Shift_Register #(9) shift(.clk(clk), .rst(rst), .left(~dirdy),
        .right(1'b0), .load(dirdy), .sin(1'b0), .in(bits), .out(obits)) ;
  DFF #(5) in_reg(clk, irdy ? in : char, char) ;
  DFF #(1) irdy_reg(clk, irdy, dirdy) ;
  DFF #(1) ov_reg(clk, noval, oval) ;
  HuffmanEncTable tab(char, length, bits) ;

endmodule
```

Figure 19.15: Verilog description of the Huffman encoder.

out bit serially on *out* starting with the first cycle in which *oval* is asserted. The value in the counter can be seen counting down from the string length (3 or 4) to 1 for each symbol while the value in the shift register *obits* is shifting the string associated with each symbol left.

19.4.2 Huffman Decoder

Now that we have encoded a character string using a Huffman code we will look at building the corresponding decoder. To decode Huffman-encoded bit string we simply traverse the encoding tree of Figure 19.13 traversing one edge for each bit of the input bit stream - the left branch for each zero, and the right branch for each one. When we encounter a terminal node during this traversal, we emit the corresponding symbol on the output and restart our traversal at the root of the tree.

To facilitate storing the decoding tree in a table, we relabel the nodes of the tree as shown in Figure 19.17. Each node is assigned an integer that serves as its address in the table. Note that the root does not need to be stored in the table, so we start labeling nodes at 0 with the left child of the root. At each entry in the table we store a type and a value. The type indicates whether this node is an internal node (type=0) or a terminal node (type=1). For a terminal node, the value holds the symbol to emit. For an internal node, the value holds the address of the left child of this node (which will always be an even number). The address of the right child can be found by adding one to the value.

To see how we traverse the table representing the tree to decode a bit string, consider decoding the bit string 001. We start at the root of the tree which has a left child with address 0 and the first 0 of the string directs us to this child. We read the entry for address zero and find that it is an internal node with a value of 2. The second bit of the string is a 0, so we proceed to address 2 (if this bit were a 1 we would have gone to address 3). We read the entry for address 2 and find that it is an internal node with a value of 6. The third bit of the string is a 1, so we proceed to one more than this value, address 7. Reading the entry for address 7 we find that it is a terminal node with a value of "T" (hex 14). We emit this value and reset our machine to start again from the root.

A block diagram of the Huffman decoder is shown in Figure 19.18 and Verilog code for the decoder is shown in Figure 19.19. The address of the current table node is held in the *node* register. When *type* is asserted — indicating a terminal node — *node* is set to the value of the next input bit (which selects one of the two children of the root to restart the search), the value field from the table is enabled into the output register, and *oval* is asserted on the following cycle. This outputs the current symbol and starts the machine at one of the children of the root depending on the first bit of the next symbol. If *type* is not assereted — indicating an internal node — the input value is combined with the value field from the table to select the left or right child of the current node — traversing the tree. The input value provides the least significant bit of the node address while the remaining bits come from the value field of the table. This simple concatenation is possible because all left children in the table have

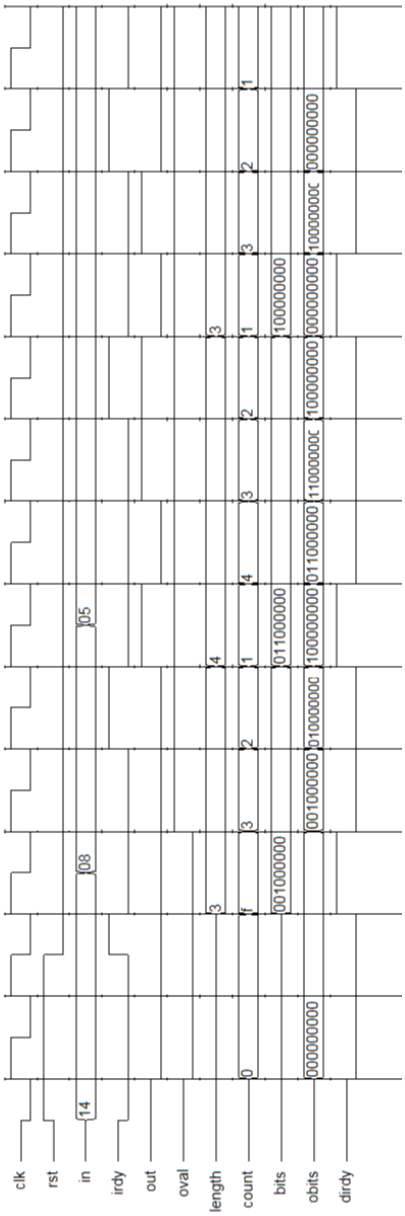


Figure 19.16: Waveforms from simulating the Huffman encoder on the string "THE".

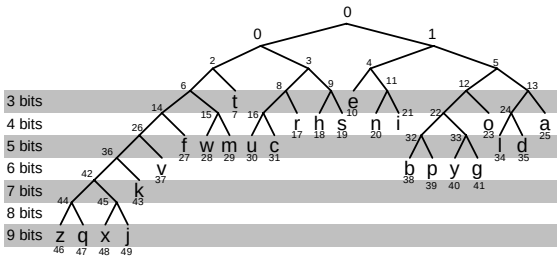


Figure 19.17: The Huffman code tree of Figure 19.13 relabeled to facilitate its storage in a decoding table. Each node of the tree is assigned a unique integer that serves as its address in the table.

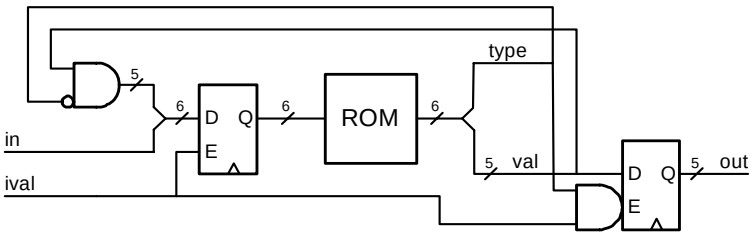


Figure 19.18: Block diagram of a Huffman decoder. The *node* register holds the address of the current tree node. The tree itself is stored in the ROM.

```
//-----------------------------------------------------------------------
// Huffman Decoder - decodes bit-stream generated by encoder
//   in - bit stream
//   ival - true when new valid bit present
//   out - output character
//   oval - true when new valid output present
//-----------------------------------------------------------------------
module HuffmanDecoder(clk, rst, in, ival, out, oval) ;
  input clk, rst, in, ival ;
  output [4:0] out ;
  output oval ;

  wire [5:0] node ; // pointers into table
  wire [4:0] value ; // output from table
  wire type ; // type from table
  wire ftype ; // fake a type on first ival cycle to prime pump

  wire [5:0] nnode = rst ? 6'd0
                         : (ival ? {(type|ftype) ? 5'b0 : value, in}
                                 : node) ;
  wire [4:0] nout = rst ? 5'd0
                        : ((ival & type) ? value : out) ;

  DFF #(6) node_reg(clk, nnode, node) ;
  HuffmanDecTable tab(node, {type, value}) ;
  DFF #(5) out_reg(clk, nout , out) ;
  DFF #(1) oval_reg(clk, ~rst & type & ival, oval) ;
  DFF #(1) ft_reg(clk, rst | (ftype & ~ival), ftype) ;
endmodule
```

Figure 19.19: Verilog description of the Huffman decoder.

even addresses. If *ival* goes low, the machine stalls, holding its present state until a valid input bit is available. Signal *ftype* in the Verilog model forces the machine to start from the root on the first valid input following reset.

Waveforms showing the combined operation of the Huffman encoder feeding the Huffman decoder are shown in Figure 19.20. The first 11 lines are the same as Figure 19.16 and represent the operation of the encoder encoding the symbol string "THE" into the bit string 0010110100. Signals `mid` and `mval` are output from the encoder and input (as `in` and `ival`) to the decoder. The state of the decoder is shown in the `node` variable and the `type` and `value` variables show what is read from the table at each node address. Note that each time `type` is asserted — indicating a leaf node — the search restarts on the next cycle with `node` at 0 or 1 (depending on `mid`). Also on the cycle following `type` the just

decoded symbol is output on `out` (values shown are hexadecimal) and `oval` is asserted to indicate a valid output.

19.5 A Video Display Controller

19.6 Exercises

19–1 *Divide-by-Four Counter:* Modify the counter from Section 19.1 to be a divide-by-four counter.

19–2 *Divide-by-Nine Counter:* Show how you can build a divide-by-nine counter using two divide-by-three counters. What happens to timing of the output pulse when you combine two counters?

19–3 *Divide-by-Three Mealy Machine:* Show how you can implement the divide-by-three counter of Section 19.1 using only three states if you allow the output to be a function of both the present state and the input. An FSM with a combinational path from input to output like this is called a *Mealy Machine* while an FSM where the output is a function only of the present state is called a *Moore Machine*.

19–4 *Modified SOS Machine:* Modify the factored SOS detector of Section 19.2 so that a dot is defined as one or two consecutive 1s and a dash is defined as 3 or 4 consecutive 1s.

19–5 *Further SOS Modifications:* Further alter the modified SOS machine from Exercise 19–4 so that the pauses between dots and dashes within a character may be 1 or 2 consecutive 0s, the spaces between characters are 3 or 4 consecutive 0s, and 5 or more consecutive 0s is an inter-word space. SOS should appear as a single word to be recognized.

19–6 *Huffman Encoder with Flow Control:* Modify the Huffman Encoder of Section 19.4.1 to accept an input valid signal *ival* that is true when a valid symbol is available on the input. A new symbol will be accepted only when both *ival* and *irdy* are asserted. Note that the output valid signal *oval* may need to go low after a string is shifted out if there is a wait for the next input signal.

19–7 *More Flow Control:* Take the Huffman encoder from Exercise 19–6 and extend it further to accept an output ready *ordy* signal that is true when the module connected to the output is ready to accept the next bit.

19–8 *One-Bit Strings:* Modify the Huffman encoder of Section 19.4.1 so it will work with a length of one. That is for codes where a symbol may be represented by a one-bit string.

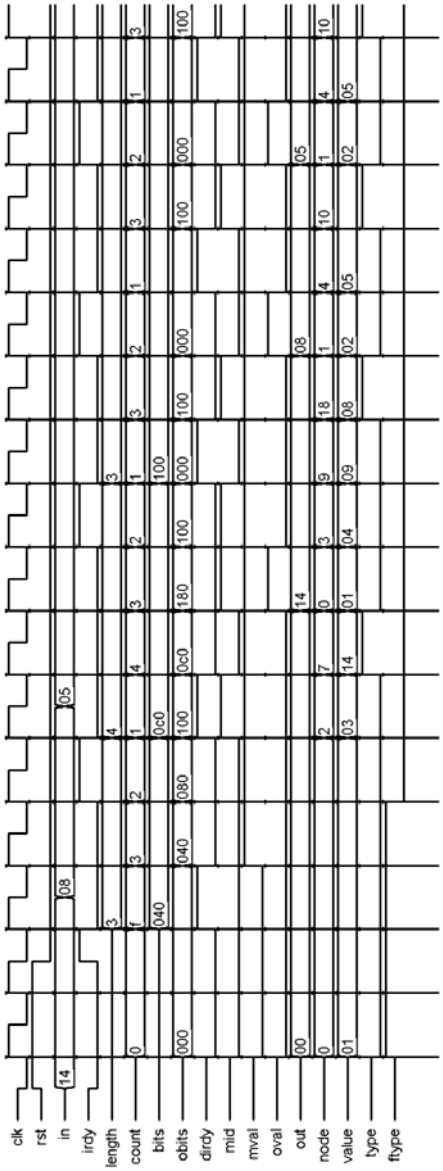


Figure 19.20: Waveforms of Huffman encoder and decoder, encoding the string "THE" into 0010110100 and then decoding this bit string back to "THE".

Chapter 20

System-Level Design

At this point in the course you now have the skills to design complex combinational and sequential logic modules. However, if someone were to ask you to design a DVD player, a computer system, or an internet router you would realize that each of these is not a single finite-state machine (or even a single datapath with associated finite-state controller). Rather, a typical system is a collection of modules each of which may include several datapaths and finite-state controllers. Once the system is decomposed to simple modules the design and analysis skills you have learned in the previous chapters can be applied. However, the problem remains to parition the system to this level where the design becomes manageable. This *system-level design* is one of the most interesting and challenging aspects of digital systems.

some idioms of system design

20.1 The System Design Process

The design of a system involves the following steps:

Specification: The most important step in designing any system is deciding - and clearly specifying in writing - what you are going to build. We discuss specifications in more detail in Section 20.2.

Partitioning: Once the system is specified, the main task in system design is dividing the system into manageable subsystems or modules. This is a process of divide and conquer. The overall system is divided into subsystems that can then be designed (conquered)separately. At each stage, the sybsystems should be specified to the same level of detail as the overall system was during our first step. As described in Section 20.3 we can partition a system by state or by task.

Interface Specification: It is particularly important that the interfaces between subsystems be described in detail. With good interface specifications, individual modules can be developed and verified independently.

When possible, interfaces should be independent of module internals — allowing modules to be modified without affecting the interface, or the design of neighboring modules.

Timing Design: Early in the design of a system, it is important to describe the timing and sequencing of operations. In particular, as work flows between modules, the sequencing of which module does a particular task on a particular cycle must be worked out to ensure that the right data comes together at the correct place and time. This timing design also drives the performance tuning step described below.

Module Design: Once the system is partitioned, modules and interfaces have been specified, and the system timing has been worked out, the individual modules can be designed and verified independently. Often the exact performance and timing (e.g., throughoput, latency, or pipeline depth) of a module is not known exactly until after the module design is complete. As these performance parameters are finalized, they may affect the system timing and require performance tuning to meet system performance specifications. The test of a good system design is if such independently designed modules can be assembled into a working system without rework.

Performance Tuning: Once the performance parameters of each module are known (or at least estimated), the system can be analyzed to see if it meets its performance specification. If a system falls short of a performance goal — or if the goal is to achieve the highest performance at a given cost — performance can be tuned by adding parallelism. This topic is treated in more detail in Chapter 21.

20.2 Specification

All too often people start desigining a system without a clear specification only to discover half-way (or further) through the design that they are building the wrong system. Much work is then discarded as they restart the design process. Another problem with vague specifications is that two designers may read the specification differently and design incompatible system parts.

A system design may start from an oral discussion of requirements. However, writing the specification down is a critical step to make sure that there are no misunderstandings about what is being designed. A written specification can also be used to validate that the right system is being designed by reviewing the specification with prospective customers and users of the system.

A good specification at a minimum must include a description of:

1. An overall description. What the system is, what it does, and how is it used.

2. All inputs and outputs: their formats, range of values, timing, and protocols.

Name	Direction	Width	Description
leftUp	input	1	when true moves the left paddle up.
leftDown	input	1	when true moves the left paddle down.
leftStart	input	1	when true starts the game or serves the ball from left to right.
rightUp	input	1	when true moves the right paddle up.
rightDown	input	1	when true moves the right paddle down.
rightStart	input	1	when true starts the game or serves the ball from right to left.
red	output	8	the intensity of the red color on the screen at the current pixel.
green	output	8	the intensity of the green color on the screen at the current pixel.
blue	output	8	the intensity of the blue color on the screen at the current pixel.
hsync	output	1	horizontal synchronization — when asserted starts a horizontal retrace of the screen.
vsync	output	1	vertical synchronization — when asserted starts a vertical retrace of the screen.

Table 20.1: Inputs and Outputs of Pong System

3. All user visible state. This includes configuration registers, mode bits, and internal memories.

4. All *modes* of operation.

5. All notable *features* of the system.

6. All interesting *edge cases*, i.e., how the system handles marginal cases.

The remainder of this section gives specifications for three example systems: a "pong" game, a DES cracker, and a music player. We will then perform the system-level design of these three examples in the remainder of the chapter.

20.2.1 Pong

Overall Description: Pong is a video game. It displays a ping-pong-like game on a VGA video screen. Users control the game using push-buttons to move the paddles and serve. Games are played to 11 points. The player winning the last point serves. The screen is considered to be a 64 by 64 grid for purposes of the game — point (0,0) is top left.

Inputs and Outputs: The inputs and outputs of our Pong system are specified in Table 20.1. Note that our digital module produces as output digital red, green, blue, and sync outputs for the display. A separate analog module combines these signals to produce the analog signals to drive the display.

Name	Width	Description
rightPadY	6	y-position of the top of the right paddle.
leftPadY	6	y-position of the top of the left paddle.
ballPosX	6	x-position of the ball.
ballPosY	6	y-position of the ball.
ballVelX	1	x velocity of the ball (0=left, 1=right).
ballVelY	2	y velocity of the ball (00=none, 01=up, 10= down).
rightScore	4	score of right player.
leftScore	4	score of left player.
mode	2	current mode - idle, rserve, lserve, play.

Table 20.2: User visible state of the Pong system

Name	Description
idle	score is zero, awaiting first serve. First start button pressed zeros score and starts game with serve from that direction. (e.g., lstart serves from left to right).
play	ball in play. Ball advances according to velocity. Hitting top or bottom of court reverses y velocity. Hitting paddle reverses x velocity. Missing left or right paddle enters rserve or lserve mode respectively and increments appropriate score.
lserve	waiting for left player to serve. When lstart is pressed ball is served from left to right.
rserve	waiting for right player to serve. When rstart is pressed ball is served from right to left.

Table 20.3: Modes of the Pong system

State: The visible state of the Pong system is shown in Table 20.2. Most of this state represents the positions of game elements on the video screen. This state is visible in the sense that it can be seen on the display. The user cannot directly read or write this state.

Modes: The modes of the pong system are shown in Table 20.3.

While we have given many details of the pong video game, this specification is by no means complete. Things left unspecified include the value of the ball position and velocity on a serve, how the y-velocity of the ball changes when hitting a paddle, and the height of a paddle. A complete specification should leave nothing to the imagination. In Exercise 20–1 the reader is given the task of completing the specification of the pong game. In practice, specifications are typically completed in an iterative manner with additional details specified on each iteration. Specification reviews are often held where a specification is presented to a group and reviewed critically to identify missing or incorrect items.

Name	Direction	Width	Description
cipherText	input	8	Cipher text to be cracked. This text is input one byte at a time. A byte is accepted on each clock for which cipherTextValid and cipherTextReady are asserted.
cipherTextValid	input	1	Asserted when cipherText has the next valid byte of cipher text to load.
cipherTextReady	output	1	Asserted when the system is able to accept a byte of cipher text.
start	input	1	Start key search. Asserted when loading of cipher text is complete to direct the system to start search of key space.
found	output	1	Asserted when the key is found.
key	output	56	When found is asserted the recovered key is output on this signal.

Table 20.4: Inputs and Outputs of the DES cracker system

20.2.2 DES Cracker

Overall Description: The DES cracker is a system that accepts a portion of ciphertext encrypted using the data encryption standard and searches the space of possible keys to find the key that was used to encrypt the ciphertext. The system assumes that the original plaintext is plain ASCII text that uses only capital letters and numbers.

Inputs and Outputs: The inputs and outputs of the DES cracker are shown in Table 20.4

State:

Modes:

20.2.3 Music Player

Music player

20.3 Partitioning

Much of system design is partitioning a system into modules. While many consider this to be an art, most systems are partitioned by state, task, or interface. With state partitioning, the system is divided into modules associated with different pieces of state (user visible or strictly internal). Each module is responsible for maintaining its portion of system state and communicating appropriate *views* of this state to other modules in the system.

With task partitioning, the function performed by a system is divided into tasks and separate modules are associated with each task. A common form

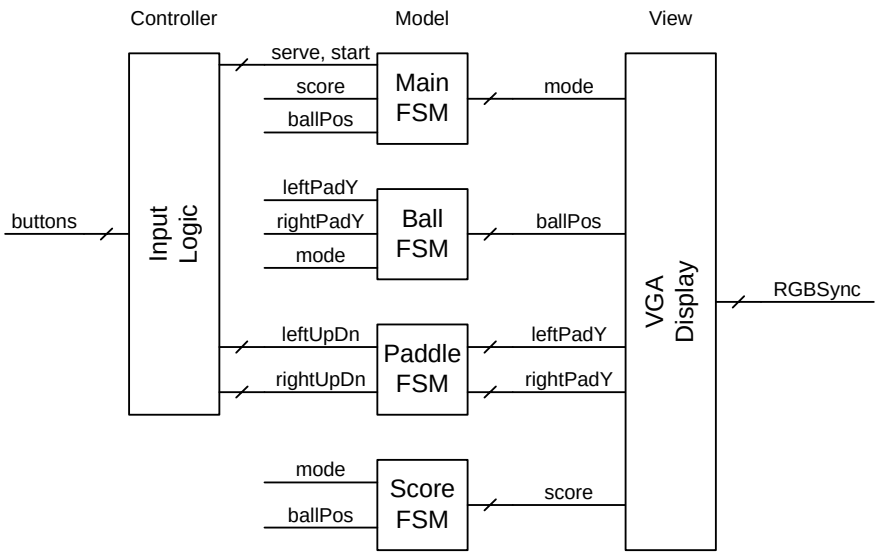


Figure 20.1: The Pong game is partitioned using the model-view-controller decomposition. The score and position of the ball and paddles constitute the *model*. This model is viewed by a VGA display module. A controller module conditions input buttons to affect the model. Note that the model is further partitioned by state into separate modules for ball, paddles, and score.

of task partitioning is model-view-controller partitioning. The system is partitioned into a model module — that contains most of the system funciton — a view module — that is responsible for all output (views of the model) — and a controller module — that is responsible for all input (controlling the model).

Finally, with interface partitioning, a separate module is associated with each interface (or related set of interfaces) of a system. Most systems employ some combination of these three partitioning techniques.

20.3.1 Pong

Figure 20.1 shows how the pong video game system is partitioned along two axes. Horizontally the system is partitioned by task, into model, view, and controller. The model portion of the system is further partitioned vertically by state with the ball state, the paddle state, the score, and the mode in separate modules. The figure also shows the interfaces defined between the modules. In most cases a module simply exports all or part of its state (e.g., score, ballPos).

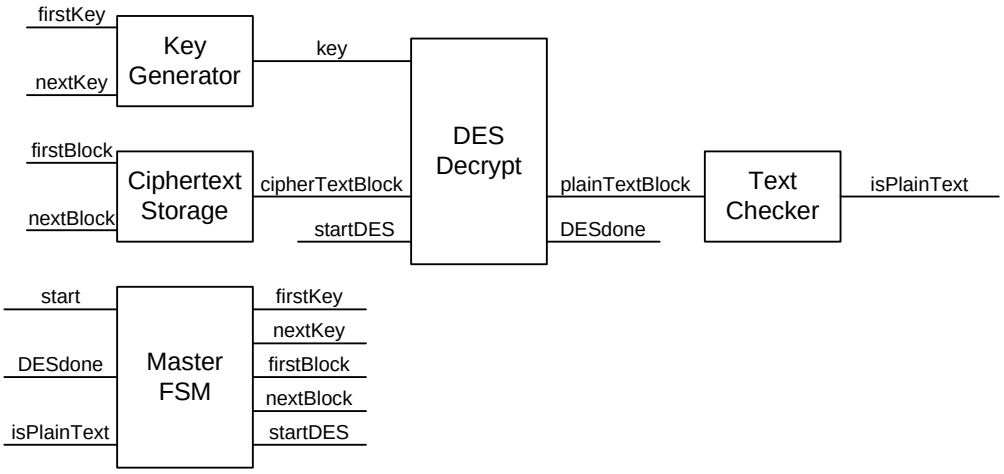


Figure 20.2: A DES *cracker* is partitioned by task. The modules perform the subtasks of generating keys, sequencing the ciphertext, decrypting the ciphertext to plaintext, and checking the plaintext to see if it is indeed plain text. A master FSM module controls overall timing and sequencing.

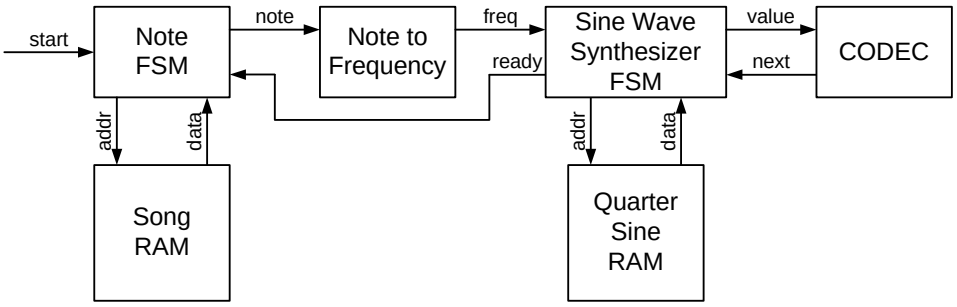


Figure 20.3: A simple music synthesizer is partitioned by task. A note FSM determines the next note to play. A note-to-frequency block converts the note into a frequency. A sine-wave synthesizer FSM synthesizes a sine wave of the specified frequency.

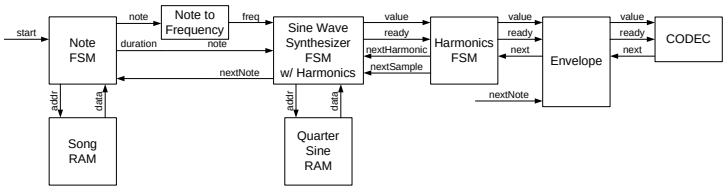


Figure 20.4: An expanded version of the music synthesizer uses a sine-wave synthesizer module that returns the value of multiple harmonics for each note, a harmonics FSM that combines these harmonics, and an envelope FSM that modulates the resulting waveform with an attack-decay envelope.

20.3.2 DES Cracker

20.4 Modules and Interfaces

modules and interfaces
- valid/ready flow control
- an example

20.5 System-Level Timing

20.6 System-Level Examples

20.7 Exercises

20–1 *Pong Specification.* The specification of the pong video game in Section 20.2.1 is purposely incomplete. Some of the missing specifications are listed in the text. Identify as many unspecified issues as you can and give a specification of each.

Chapter 21

Pipelines

A pipeline is a sequence of modules, called *stages* that each perform part of an overall task. Each stage is like a station along an assembly line — it performs a part of the overall assembly and passes the partial result to the next stage. By passing the incomplete task down the pipeline, each stage is able to start work on a new task before waiting for the overall task to be completed. Thus, a pipeline may be able to perform more tasks per unit time (i.e., it has greater *throughput*) than a single module that performs the whole task from start to finish.

load balance
elastic

21.1 Basic Pipelining

Suppose you have a factory that assembles toy cars. Assembling each car takes four steps. In step 1 the body is shapred from a block of wood. In step 2 the body is painted. In step 3 the wheels are attached. Finally, in step 4, the car is placed in a box. Suppose each of the four steps takes 5 minutes. With one employee, your factory can assemble one toy car each 20 minutes. With four employees your factory can assemble one car every 5 minutes in one of two ways. You could have each employee perform all four steps — producing a toy car every 20 minutes. Alternatively, you could arrange your employees in an assembly line with each employee performing one step and passing partially completed cars down to the next employee.

In a digital system, a pipeline is like an assembly line. We take an overall task (like building a toy car) and break it into subtasks (each of the four steps). We have a separate unit, called a pipeline stage (like each employee along the assembly line), perform each task. The stages are tied together in a linear manner so that the output of each unit is the input of the next unit — like the employees along the assembly line passing their output (partially assembled cars) to the next employee down the line.

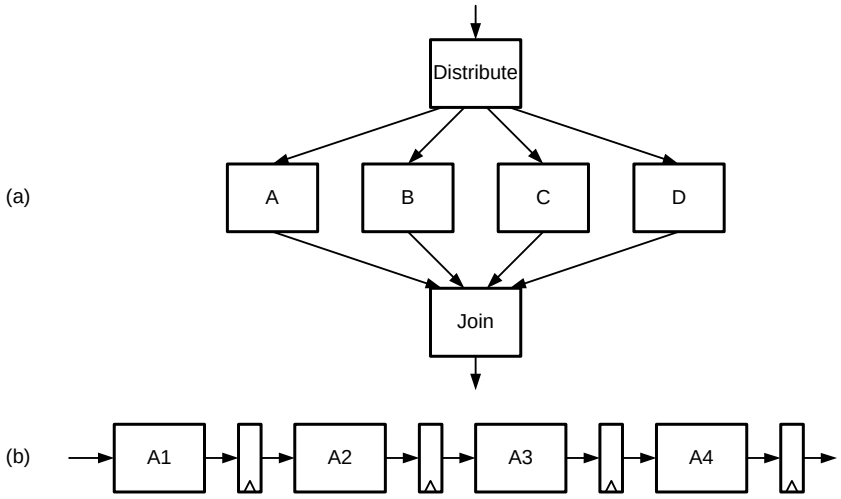


Figure 21.1: Throughput of a module can be increased by (a) using parallel copies of the module, or (b) pipeling a single copy of the module.

The *throughput* Θ of a module is the number of problems a module can solve (or tasks a module can perform) per unit time. For example, if we have an adder that is able to perform one add operation every 10ns, we say that the throughput of the adder is 100Mops. The latency T of a module is the amount of time it takes the module to complete one task from begininng to end. For example, if our adder takes 10ns to complete a problem from the time the inputs are applied to the time the output is stable, its latency is 10ns. Latency is just another word for delay. For a simple module, throughput and latency are reciprocals of one another: $\Theta = \frac{1}{T}$. If we accelerate modules through pipelining or parallelism, however, the relation becomes more complex.

Suppose we need to increase the throughput of a module with $T = 10$ns, $\Theta = 100$Mops by a factor of four. Further suppose the module is already highly optimized so that we are unlikely to get much increase in throughput by redesigning the module. Just as with our toy car factory, we have two basic options. First, we could build four copies of our module as shown in Figure 21.1(a). Modules A through D are identical copies of our original module. The distribute block distributes problems to the four modules, and the join block combines the results. Here we can start four copies of our problem in parallel as shown in Figure 21.2(a). Our latency is still $T = 10$ns, because it still takes 10ns to complete one problem from beginning to end. Our throughput, however, has been increased to $\Theta = 400$Mops since we are able to solve four problems every 10ns.

An alternative method of increasing throughput is to *pipeline* a single copy of the module as shown in Figure 21.1(b). Here we have taken a single module

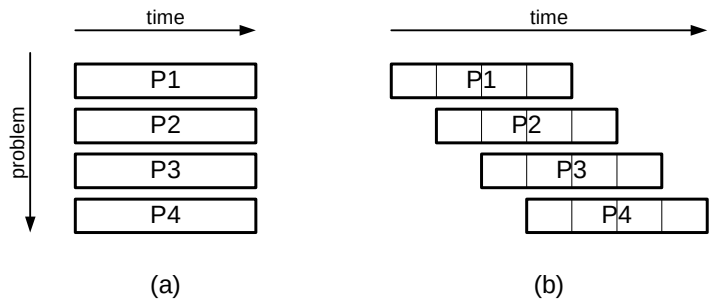


Figure 21.2: Timing diagram showing execution of four problems $P1, \ldots, P4$ on the parallel and pipelined configurations of Figure 21.1.

A and divided it into four parts $A1, \ldots, A4$. We assume that we were able to do this partition evenly so that the delay of each of the four submodules Ai is $T_{Ai} = 2.5$ns. (Such an even division of modules is not always possible as discussed below in Section 21.3). We insert a register between adjacent submodules. Each register holds the result of the preceding submodule on one problem freeing that submodule to begin working on the next problem. Thus, as shown in Figure 21.2(b), this pipeline can operate on four problems at once in a staggered fashion. As soon as submodule $A1$ finishes work on problem $P1$, it starts working on $P2$ while $A2$ continues work on $P1$. Each problem continues down the pipeline, advancing one stage each clock cycle, until it is completed by module $A4$. If we ignore (for now) register overhead, our latency is still $T = 10$ns (2.5ns $\times$ 4 stages) and our throughput has been increased to 400Mops.

Compared to using parallel moudules, pipelining has the advantage that it multiplies throughput without the cost of duplicating modules. Pipelining, however, is not without its own costs. First, pipelining requires inserting registers between pipeline stages. In some cases, these registers can be very expensive. Also, a pipelined implementation has more register overhead than a corresponding parallel design.

Now consider the affect of register overhead. Suppose each register has a total overhead $t_{\rm reg} = t_s + t_{dCQ} + t_k = 200$ps. For a single module or a parallel combination (Figure 21.1(a)), we only pay this overhead once, increasing our latency to $T = 10.2$ns and reducing the throughput of four parallel modules to $\Theta = 4/10.2 = 392$Mops. Pipelining the module, on the other hand, incurs the register overhead once per stage. Thus our latency is increased to $T = 10.8$ns and our througput is reduced to $\Theta = 1/2.7 = 370$Mops.

In practice, many implementations use parallel pipelined modules. One pipelines a module until the cost of the registers and the register overhead becomes expensive and then obtains further throughput increases by using multiple copies of the pipelines working in parallel.

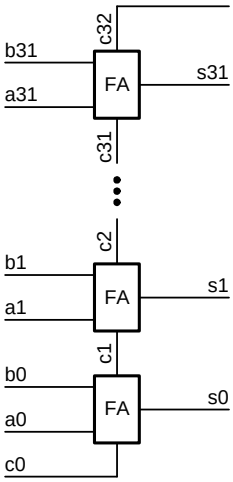


Figure 21.3: A 32-bit ripple carry adder. If each stage requires 100ps, an add can be performed in 3.2ns.

21.2 Example: Pipelining a Ripple-Carry Adder

As a more concrete example of pipelining, consider a 32-bit ripple-carry adder as shown in Figure 21.3. If each full-adder module has a delay from carry-in to carry-out of $t_{dcc} = 100$ps, then in the worst-case (where the carry must propagate all the way from the LSB to the MSB), the delay of the adder is $t_{d\text{add}} = 32t_{dcc} = 3.2$ns. This single adder is capable of performing one add every 3.2ns for a throughput of 312.5 million adds per second. Suppose our goal is to achieve a throughput of 1 billion adds per second — starting a new add every 1ns. We can pipeline this adder to achieve this goal.

In operation, only a single bit of this adder is *busy* at any point in time. For example, 1ns after the inputs are applied only bit 10 is active. Bits 0 through 9 have completed operation, and bits 11-31 are waiting on their carry inputs. (They have computed p and g, but cannot compute s until the carry input is available). By pipelining the adder n ways, we can get n of the 32-bits working at a given point in time.

The first step in pipelining a unit is to divide it into submodules. Figure 21.4 shows our 32-bit adder module divided into four submodules each of which performs an 8-bit add. Each 8-bit adder accepts an 8-bit slice of each input vector, a[i+7:i] and b[i+7:i], along with a carry in ci and generates an 8-bit slice of the sum s[i+7:i] and a carry out c[i+8]. The delay (from carry-in to carry-out) of each of these 8-bit adders is $t_{d8} = 8t_{dcc} = 800ps$ so it will fit in a 1ns clock cycle leaving 200ps for register overhead.

To allow our partitioned adder to work on muliple problems at the same

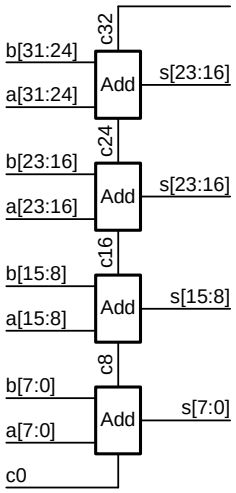


Figure 21.4: Dividing the 32-bit ripple-carry adder into four 8-bit adders.

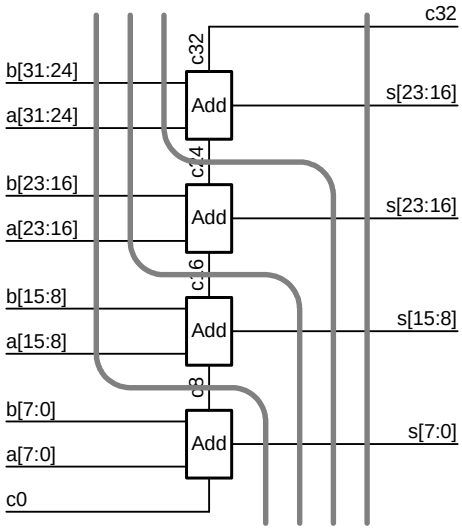


Figure 21.5: To pipeline the 32-bit ripple-carry adder, we insert registers to separate the stages at locations shown by the thick, gray lines. All paths from input to output must pass through each pipeline register.

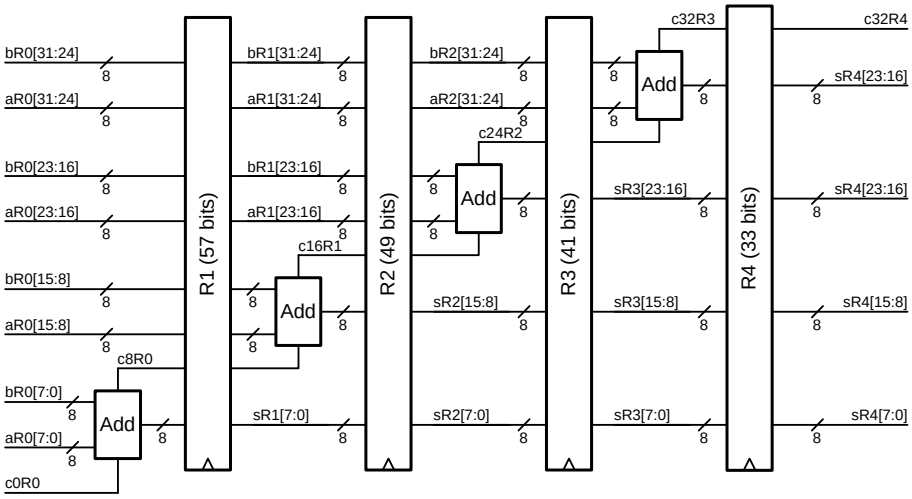


Figure 21.6: The pipelined 32-bit ripple-carry adder. Registers have been inserted and the schematic redrawn to make each pipeline stage one horizontal region. Signal names are augmented with the name of the pipeline register producing them.

time, we insert *pipeline registers* between the submodules. The thick, gray lines in Figure 21.5 show where the registers are to be inserted. The circuit redrawn after inserting the registers is shown in Figure 21.6. The combination of a submodule and its register is referred to as a *pipeline stage*. Pipeline register $R1$ is inserted after the first 8-bit adder. This register captures the partial result of the add after 800ps — this includes the low eight-bits of the sum sR0[7:0], bit 8 of the carry c8R0, and the upper 3-bytes of the input vectors aR0[31:8] and bR0[31:8] for a total of 57 bits.

We label all signals before $R1$ (including the primary inputs) with the suffix $R0$ to denote that they are in the 0^{th} pipeline stage and to distingish these signals from signals in other pipeline stages. In a similar manner, the outputs of $R1$, and all other signals in first pipeline stage, are labeled with a suffix $R1$. Note that signals sR1[7:0] and sR2[7:0] are different signals. While they are both the low byte of a sum, at any point in time they are the low bytes of different sums. By labeling signals with their pipeline stage we can easily spot the common pipeline error of combining signals from different stages. We know that an expression `fooR1 & barR2` is an error because signals from different pipeline stages should not be combined.

The second stage of the pipeline adds the second byte of data aR1[15:8] and bR1[15:8], using c8R1 as the carry in, giving sR1[15:8] and c16R1. The outputs of this second byte adder are captured by $R2$ along with sR1[7:0] and aR1[31:16] and bR1[31:16], a total of 49 bits. In a similar manner the third and

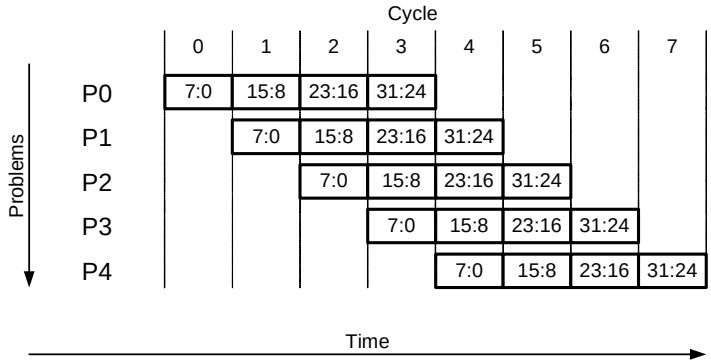


Figure 21.7: Pipeline diagram showing the timing of the pipelined 32-bit ripple-carry adder. The diagram illustrates on which cycle each sum byte of problems $P0, \ldots, P4$ is computed.

fourth pipeline stages add the third and fourth bytes of data. At the output of the fourth stage, register $R4$ captures the full 32-bit product sR3[31:0] and the carry out c32R3. The output of $R4$ is the result of the add: s[31:0] and c32.

Timing of the pipeline is illustratd in the *pipeline diagram* of Figure 21.7. The figure shows five problems, $P0, \ldots, P4$, proceeding down the pipeline. Problem Pi enters the pipeline on cycle i. Byte j (bits [8j+7:8j]) of the sum of Problem i is computed during cycle $i + j$. The result of the add for problem Pi appears on the output four clock cycles later, on cycle $i + 4$. At the time the result for a problem $P0$ appears on the output during cycle 4, four subsequent problems are in process in the various pipeline stages.

Pipelining our ripple-carry adder has increased both its throughput and its latency. If we assume that register overhead $t_o = t_s + t_{dCQ} + t_k$ is 200ps for each register, then the delay of each pipeline stage is 1ns — 800ps for the 8-bit adder, and 200ps for register overhead. Hence we can operate our pipeline at 1GHz, achieving our throughput goal of $\Theta = 1$Gops. The latency of our pipeline is $T = 4$ns, compared to the original 3.2ns for the unpiplined adder. The difference is the overhead of the four registers.

In general, if the delay of the combinational logic in the longest pipeline stage is $t_{\max}$, then the total delay of a pipeline stage is:

$$t_{\text{pipe}} = t_{\max} + t_o = t_{\max} + t_s + t_{dCQ} + t_k. \tag{21.1}$$

From t_{pipe} we see the latency of n stages is:

$$T = n\left(t_{\max} + t_o\right) = n\left(t_{\max} + t_s + t_{dCQ} + t_k\right), \tag{21.2}$$

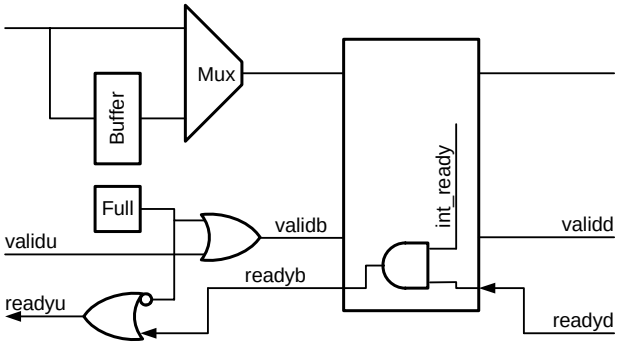


Figure 21.8:

and the throughput is:

(21.3)

21.3 Load Balancing

21.4 Variable Loads

21.5 Double Buffering

Chapter 22

Asynchronous Sequential Circuits

Asynchronous sequential circuits have state that is not synchronized with a clock. Like the synchronous sequential circuits we have studied up to this point they are realized by adding state feedback to combinational logic that implements a next-state function. Unlike synchronous circuits, the state variables of an asynchronous sequential circuit may change at any point in time. This asynchronous state update – from next state to current state – complicates the design process. We must be concerned with hazards in the next state function, as a momentary glitch may result in an incorrect final state. We must also be concerned with *races* between state variables on transitions between states whose encodings differ in more than one variable.

In this chapter we look at the fundamentals of asynchronous sequential circuits. We start by showing how to analyze combinational logic with feedback by drawing a flow table. The flow table shows us which states are stable, which are transient, and which are oscillatory. We then show how to synthesize an asynchronous circuit from a specification by first writing a flow table and then reducing the flow table to logic equations. We see that state assignment is quite critical for asynchronous sequential machines as it determines when a potential race may occur. We show that some races can be eliminated by introducing transient states.

After the introduction of this chapter, we continue our discussion of asynchronous circuits in Chapter 23 by looking at latches and flip-flops as examples of asynchronous circuits.

22.1 Flow Table Analysis

Recall from Section 14.1 that an asynchronous sequential circuit is formed when a feedback path is placed around combinational logic as shown in Figure 22.1(a). To analyze such circuits, we break the feedback path as shown in Figure 22.1(b)

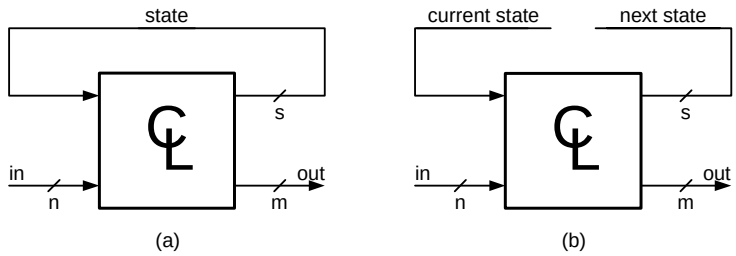


Figure 22.1: Asynchronous sequential circuit. (a) A sequential circuit is formed when a feedback path carrying state information is added to combinational logic. (b) To analyze an asynchronous sequential circuit, we break the feedback path and look at how the next state depends on the current state.

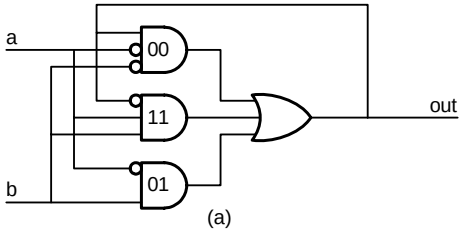
and write the equations for the *next* state variables as a function of the *current* state variables and the inputs. We can then reason about the dynamics of the circuit by exploring what happens when the current state variables are updated, in arbitrary order if multiple bits change, with their new values.

At first this may look just like the synchronous sequential circuits we discussed in Section 14.2. In both cases we compute a next state based on current state and input. What's different is the dynamics of how the current state is updated with the next state. Without a clocked state register, the state of an asynchronous sequential circuit may change at any time (asynchronously). When multiple bits of state are changing at the same time (a condition called a *race*. The bits may change at different rates resulting in different end states. Also, a synchronous circuit will eventually reach a steady state where the next state and outputs will not change until the next clock cycle. An asynchronous circuit on the other hand may never reach a steady state. It is possible for it to oscillate indefinitely in the absence of input changes.

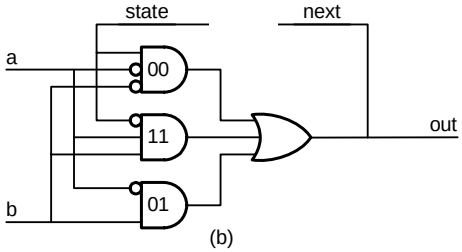
We have already seen one example of analyzing an asynchronous circuit in this manner - the RS flip-flop of Section 14.1. In this section we look at some additional examples and introduce the *flow table* as a tool for the analysis and synthesis of asynchronous circuits.

Consider the circuit shown in Figure 22.2(a). Each of the AND gates in the figure is labeled with the input state ab during which it is enabled. For example, the top gate, labeled 00, is enabled when a is high and b is low. To analyze the circuit we break the feedback loop as shown in Figure 22.2(b). At this point we can write down the next-state function in terms of the inputs, a and b, and the current state. This function is shown in tabular form in the *flow table* of Figure 22.2(c).

Figure 22.2(c) shows the next state for each of the eight combinations of inputs and current state. Input states are shown horizontally in Gray-code order. Current states are shown vertically. If the next state is the same as the current state, this state is *stable* since updating the current state with the next



(a)



(b)

State	Next			
	00	01	11	10
0	(0)	1	1	(0)
1	(1)	(1)	0	0

(c)

Figure 22.2: An example asynchronous sequential circuit. (a) The original circuit. (b) With feedback loop broken. (c) Flow table showing next-state function. Circled entries in the flow table are *stable* states.

state doesn't change anything. If the next state is different than the current state, this state is *transient* since as soon as the current state is updated with the next state, the circuit will change states.

For example, suppose the circuit has inputs ab = 00 and the current state is 0. The next state is also 0, so this is a stable state - as shown by the circled 0 in the leftmost position of the top row of the table. If from this state input b goes high, making the input state ab = 01, we move one square to the right in the table. In this case, the 01 AND gate is enabled and the next-state is 1. This is an unstable or transient situation since the current state and next state are different. After some amount of time (for the change to propagate) the current state will become 1 and we move to the bottom row of the table. At this point we have reached a stable state since the current and next state are now both 1.

If there is a cycle of transient states with no stable states we have an *oscillation*. For example, if the inputs to the circuit of Figure 22.2 are ab = 11, the next state is always the complement of the current state. With this input state, the circuit is never stable, but instead will oscillate indefinitely between the 0 and 1 states. This is almost never a desired behavior. An oscillation in an asynchronous circuit is almost always an error.

So, what does the circuit of Figure 22.2 do? By this point the estute reader will have realized that its an RS flip-flop with an oscillation feature added. Input a is the reset input. When a is high and b is low, the state is made 0 when a is lowered the state remains 0. Similarly b is the set input. Making b high while a is low sets the state to 1 and it remains at 1 when b is lowered. The only difference between this flip-flop and the one of Figure 14.2 is that when both inputs are high the circuit of Figure 22.2 oscillates while the circuit of Figure 14.2 resets.

To simplify our analysis of asynchronous circuits we typically insist that the environment in which the circuits operate obey the *fundamental mode* restriction:

> **Fundamental-Mode:** Only one input bit may be changed at a time and the circuit must reach a stable state before another input bit is changed.

A circuit operated in fundamental-mode need only worry about one input bit changing at a time. Multiple-bit input changes are not allowed. Our setup- and hold-time restrictions on flip-flops are an example of a fundamental-mode restriction. The clock and data inputs of the flip flop are not allowed to change at the same time. After the data input changes, the circuit must be allowed to reach a steady-state (setup time) before the clock input can change. Similarly, after the clock input changes, the circuit must be allowed to reach a steady-state (hold time) before the data input can change. We will look at the relation of setup and hold time to the design of the asynchronous circuits that realize flip-flops in more detail in Chapter 23.

In looking at a flow-table, like the one in Figure 22.2, operating in the fundamental mode means that we need only consider input transitions to adjacent

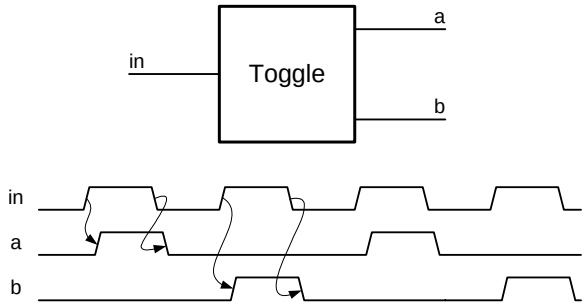


Figure 22.3: A toggle circuit alternates pulses on it input *in* between its two outputs *a* and *b*.

squares (including wrapping from leftmost to rightmost). Thus, we don't have to worry about what happens when the input changes from 11 (oscillating) to 00 (storing). This can't happen. Since only one input can change at a time, we must first visit state 10 (reset) or 01 (set) before getting to 00.

In some real world situations, it is not possible to restrict the inputs to operate in fundamental mode. In these cases we need consider multiple input changes. This topic is beyond the scope of this book and the interested reader is referred to some of the texts listed in Section 22.4.

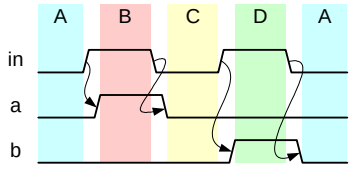
22.2 Flow-Table Synthesis: The Toggle Circuit

We now understand how to use a flow-table to analyze the behavior of an asynchronus circuit. That is, given a schematic, we can draw a flow table and understand the function of the circuit. In this section we will use a flow table in the other direction. We will see how to create a flow table from the specification of a circuit and then use that flow table to synthesize a schematic for a circuit that realizes the specification.

Consider the specification of a toggle circuit - shown graphically in Figure 22.3. The toggle circuit has a single input *in* and two outputs *a* and *b*.[1] Whenever *in* is low, both outputs are low. The first time *in* goes high, output *a* goes high. On the next rising transition of *in*, output *b* goes high. On the third rising input, *a* goes high again. The circuit continues steering pulses on *in* alternately between *a* and *b*.

The first step in synthesizing a toggle circuit is to write down its flow table. We can do this directly from the waveforms of Figure 22.3. Each transition of the input potentially takes us to a new state. Thus, we can partition the waveform into potential states as shown in Figure 22.4. We start in state A. When *in* rises we go to state B where output *a* is high. when *in* falls again we

[1] In practice a reset input *rst* is also required to initialize the state of the circuit.



State	Next (in) 0	Next (in) 1	Out (a,b)
A	(A)	B	00
B	C	(B)	10
C	(C)	D	00
D	A	(D)	01

Figure 22.4: A flow table is created from the specification of the toggle circuit by creating a new state for every input transition until the circuit is obviously back to the same state.

go to state C. Even though C has the same output as A, we know its a different state because the next transition on *in* will cause a different output. The second rising edge on *in* takes us to state D with output *b* high. When *in* falls for the second time we go back to state A. We know that this state is the same as state A since the behavior of the circuit at this point under all possible inputs is indistinguishable from where we started.

Once we have a flow table for the toggle circuit, the next step is to assign binary codes to each of the states. This state assignment is more critical than with synchronous machines. If two states X and Y differ in more than one state bit, a transition from X to Y requires first visiting a *transient* state with one state bit changed before arriving at Y. In some cases, a *race* between the two state bits may result. We discuss races in more detail in Section 22.3. For now, we pick a state assignment (shown in Figure 22.5(a) where each state transition switches only a single bit.

With the state assignment, realizing the logic for the toggle circuit is a simple matter of combinational logic synthesis. We redraw the flow table as a Karnaugh map in Figure 22.5(b). The Karnaugh map shows the symbolic next state function - i.e., each square shows the next state name (A through D) for that input and current state. The arrows show the path through the states followed during operation of the circuit. Understanding this path is important for avoiding races and hazards. We refer to this Karnaugh map showing the state transitions as a *trajectory map since it shows the trajectory of the state variables.*

We redraw the Karnaugh map with state names replaced by their binary codes in Figure 22.5(c), and separate maps for the two state variables s_0 and s_1

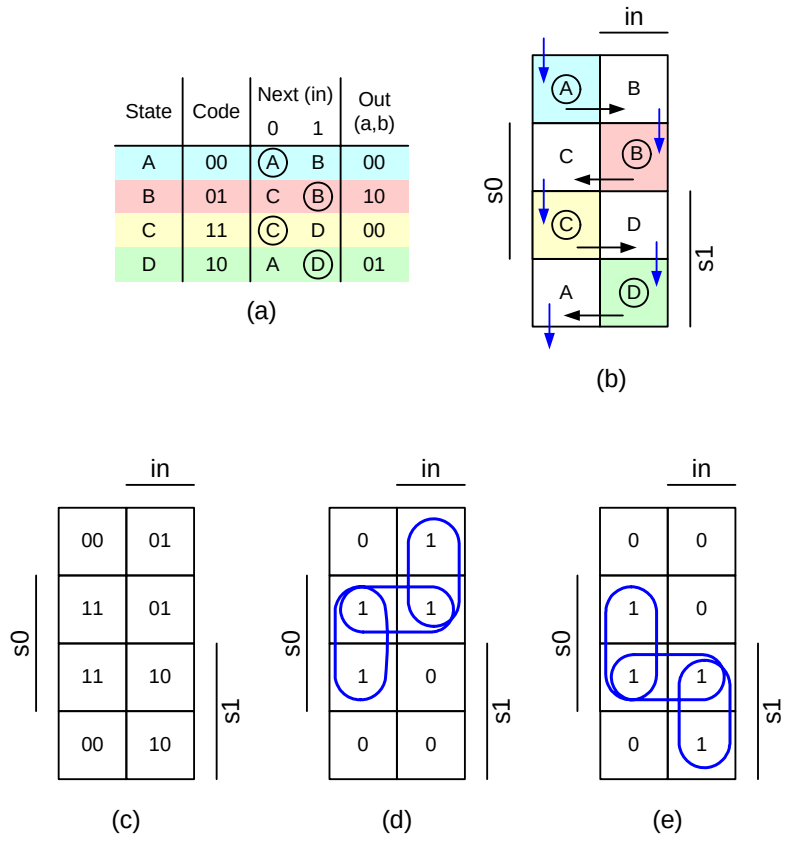


Figure 22.5: Implementing the toggle circuit from its flow table. (a) Flow table with state assignment. (b) Flow table mapped to Karnaugh map. (c) Next state codes mapped to Karnaugh map. (d) Karnaugh map for s0. (e) Karnaugh map for s1.

are shown in Figure 22.5(d) and (e) respectively. From these Karnaugh maps we write down the equations for s_0 and s_1:

$$s_0 = (\overline{s_1} \wedge in) \vee (s_0 \wedge \overline{in}) \vee (s_0 \wedge \overline{s_1}), \quad (22.1)$$

$$s_1 = (s_1 \wedge in) \vee (s_0 \wedge \overline{in}) \vee (s_0 \wedge s_1). \quad (22.2)$$

The last implicant in each expression is required to avoid a hazard that would otherwise occur. Asynchronous circuits must be hazard free along their path through the input/state space. Because the current state is being constantly fed back a glitch during a state transition can result in the circuit switching to a different state - and hence not implementing the desired function. For example, suppose we left the $s_0 \wedge \overline{s_1}$ term out of (22.2). When *in* goes low in state B, s_0 might go momentarily low before s_1 comes high. At this point the middle term of both equations becomes false and s_1 never goes high - the circuit goes to state A rather than C.

All that remains to complete our synthesis is to write the output equations. Output a is true in state 01 and output b is true in state 10. The equations are thus:

$$a = \overline{s_1} \wedge s_0, \quad (22.3)$$

$$b = s_1 \wedge \overline{s_0}. \quad (22.4)$$

22.3 Races and State Assignment

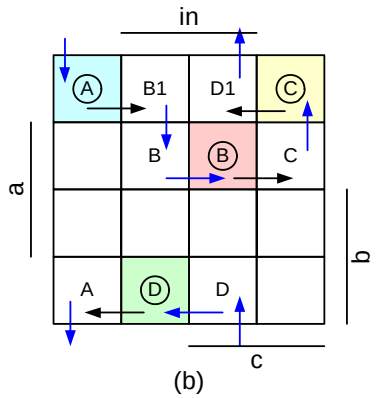
To illustrate the problem of multiple state variables changing simultaneously, consider an alternate state assignment for the toggle circuit shown in Figure 22.6(a). Here we observe that the two outputs, a and b can also serve as state variables, so we can add to the outputs just one additional state variable c to distinguish between states A and C giving the codes shown in the figure.[2]

With this state assignment, the transition from state A ($cab = 000$) to state B (110) changes both c and a. If the logic is designed so that *in* going high in state A makes both c and a go high, they could change in an arbitrary order. Variable a could change first, variable c could change first, or they could change simultaneously. If they change simultaneously, we go from state A directly to state B with no intermediate stops. If a changes first, we go first to state 010 which is not assigned and then, if the logic in state 010 does the right thing, to state 110. If c changes first, the machine will go to state C (100) where the high input will then drive it to state D. Clearly, we cannot allow c to change first. This situation where multiple state variables can change at the same time is called a *race*. The state variables are *racing* to see which one can change first.

[2]Note that the bit ordering of the codes is c,a,b.

State	Code (c,a,b)	Next (in) 0	Next (in) 1	Out (a,b)
A	000	(A)	B	00
B	110	C	(B)	10
C	100	(C)	D	00
D	001	A	(D)	01

(a)



(b)

Figure 22.6: An alternate state assignment for the toggle circuit requires multiple state variables to change on a single transition. (a) The flow table with the revised state assignment. (b) A trajectory map showing the introduction of transient states B1 = 010 and D1 = 101.

When the outcome of the race affects the end state - as in this case - we call the race a *critical race*.

To avoid the critical race that could occur if we allow both a and c to change at the same time, we specify the next-state function so that only a can change in state A. This takes us to a *transient state* 010, which we will call B1. When the machine reaches state B1, the next state logic then enables c to change.

The introduction of this transient state is illustrated in the trajectory map of Figure 22.6(b). When the input goes high in state A, the next state function specifies B1 rather than B. This enables only a single transition, downward as shown by the blue arrow - which corresponds to a rising, to state B1. A change in c is not enabled in this state to avoid a horizontal transition into the square marked D1. Once the machine reaches state B1, the next state function becomes B which enables the change in c, a horizontal transition, to stable state B.

A transient state is also required for the transition from state C 100 to state D 001. Both variables c and b change between these two states. An uncontrolled race in which variable c changes first could wind up in state A 000 which is not correct. To prevent this race, we enable only b to change when *in* rises in state C. This takes us to a transient state D1 (101). Once in state D1, c is allowed to fall, taking us to state D (001).

Figure 22.7 illustrates the process of implementing the revised toggle circuit. Figure 22.7(a) shows a Karnaugh map of the next-state function. Each square of the Karnaugh map shows the code for the *next* state for that present state and input. Note that where the next state equals the present state the state is stable. Transient state B1 (at 010 - the second square along the diagonal) is not stable since it has a next state of 110.

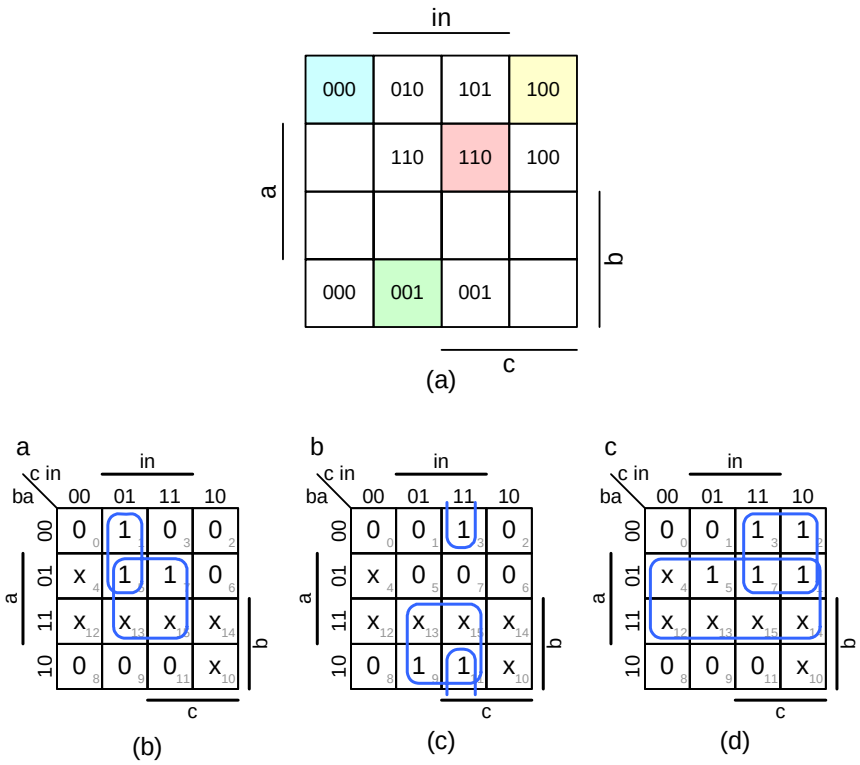


Figure 22.7: Implementation of the toggle circuit with the alternate state assignment of Figure 22.6(a). (a) Karnaugh map showing 3-bit next state function c,a,b. (b) Karnaugh map for a. (c) Karnaugh map for b. (d) Karnaugh map for c.

From the next-state Karnaugh map we can write the individual Karnaugh maps for each state variable. Figure 22.7 (b) through (d) show the Karnaugh maps for the individual variables - a, b, and c respectively. Note that the states that are not visited along the state trajectory (the blank squares in Figure 22.7(a)) are don't cares. The machine will never be in these states, thus we don't care what the next state function is in an unvisited state.

From these Karnaugh maps we write the state variable equations as:

$$a = (in \wedge \overline{b} \wedge \overline{c}) \vee (in \wedge a), \tag{22.5}$$

$$b = (in \wedge \overline{a} \wedge c) \vee (in \wedge b), \tag{22.6}$$

$$c = a \vee (\overline{b} \wedge c). \tag{22.7}$$

Note that we don't require separate equations for output variables since a and b are both state variables and output variables.

22.4 Bibliographic Notes

many early computer designs were completely asynchronous

over time, the simplicity of synchronous design won out.

today async is used mostly for flip-flops and interfaces to inherently asynchronous processes.

there is an active research area in asynchronous design - async conferences.

texts that give more detail on the field include Unger and Kohavi

22.5 Exercises

22–1 Analysis. write a flow table for a schematic. State what the circuit does.

22–2 Synthesis. write a flow table for a proportional phase comparator. Synthesize a gate-level circuit from your flow table.

22–3 *Toggle Synthesis.* Synthesize a version of the toggle circuit where the state assignments are as in Figure 22.6(a) except that state B is encoded as *cab* = 010.

22–4 *Edge Toggle.* The toggle circuit of Section 22.2 is a *pulse toggle* — a circuit in which alternate pulses on the input alternate between the two outputs. For this exercise you are to design an *edge toggle* — a circuit in which edges on the input cause edges that alternate between the two outputs.

22–5 *Three-Way Toggle.* Design a toggle circuit like the one in Section 22.2 except that pulses on the input alternate over three outputs.

22–6 *Three-Way Edge Toggle.*

22–7 State Reduction. Write a flow table for the toggle circuit but this time creating a new state for each of the first eight transitions on *in*. Then determine which states are equivalent to reduce your flow table to a four-state table.

22–8 Races.

Chapter 23

Flip Flops

Flip-flops are among the most critical circuits in a modern digital system. As we have seen in previous chapters, flip flops are central to all synchronous sequential logic. Registers (built from flip-flops) hold the state (both control and data state) of all of our finite-state machines. In addition to this central role in logic design, flip-flops also consume a large fraction of the die area, power, and cycle time of a typical digital system.

Up until now, we have considered a flip-flop as a black box.[1] In this chapter we *look inside* the flip-flop. We derive the logic design of a typical D-flip-flop and show how the timing properties introduced in Chapter 15 follow from this design.

We first develop the flip-flop design informally - following an intuitive argument. We start by developing the latch. The implementation of a latch follows directly from its specification. From the implementation we can then derive the setup, hold, and delay times of the latch. We then see how to build a flip-flop by combining two latches in a master-slave arrangement. The timing properties of the flip-flop can then be derived from its implementation.

Following this informal development, we then derive the design of a latch and flip-flop using flow-table synthesis. This serves both to reinforce the properties of these storage elements and to give a good example of flow-table synthesis. We introduce the concept of *state equivalence* during this derivation. This formal derivation can be skipped by a casual reader.

23.1 Inside a Latch

A schematic symbol for a latch is shown in Figure 23.1(a) and waveforms illustrating its behavior and timing are shown in Figure 23.1(b). A latch has two inputs data d and enable g, and one output q. When the enable input is high,

[1] A *black box* is a system that we understand the external specifications, but not the internal implementation - as if the system were inside an opaque (black) box that keeps us from seeing how it works.

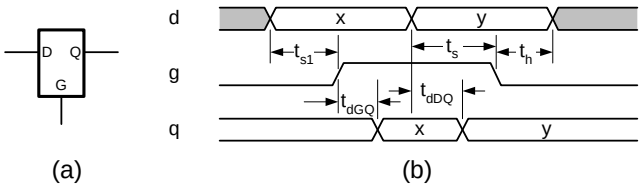


Figure 23.1: A latch. (a) Schematic symbol. (b) Waveforms showing timing properties.

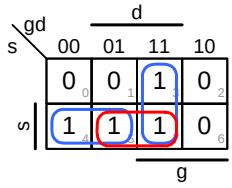


Figure 23.2: Karnaugh map for a latch showing hazard.

the output follows the input. When the enable input is low, the output holds its current state.

As shown in Figure 23.1(b) a latch, like a flip-flop has a setup time t_s and a hold time t_h. An input must be setup t_s before the enable *falls* and held for t_h after the enable falls for the input value to be correctly stored. Latch delay is characterized by both delay from the enable rising to the output changing, t_{dGQ}, and delay from the data input changing to the output changing, t_{dDQ}. For the enable to dominate the delay, the input must be setup at least t_{s1} before the enable rises. Usually, this is just a question of which signal (d or g) is on the critical path and $t_{s1} = t_{dDQ} - t_{dGQ}$. As we shall see below, these times can be derived from the logic design of the latch and the need to meet the fundamental-mode restriction (Section 22.1) during latch operation.

From the description of a latch, we can write down its logic equation:

$$q = (g \wedge d) \vee (\overline{g} \wedge q). \tag{23.1}$$

That is, when g is true, $q = d$, and when g is false, q holds its state ($q = q$). This is almost correct. As can be seen from the Karnaugh map of Figure 23.2, there is a hazard that may occur when g changes state and both d and q are high. To cover this hazard, we must add an additional implicant to the equation.

$$q = (g \wedge d) \vee (\overline{g} \wedge q) \vee (d \wedge q). \tag{23.2}$$

From Equation (23.2) we can draw a gate-level schematic for a latch as shown in Figure 23.3(a). This implementation of a latch is often called an *Earle* latch

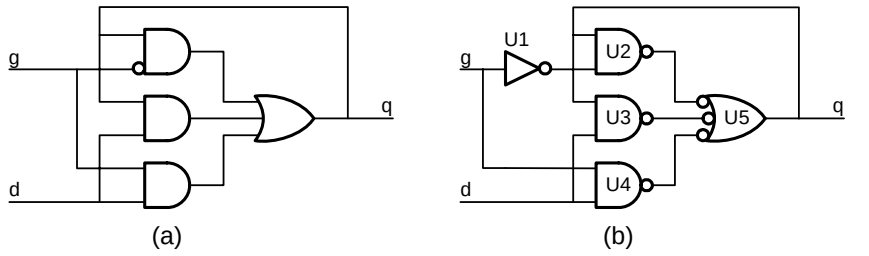


Figure 23.3: Gate-level schematic of an Earle latch. (a) Using abstract AND and OR gates. (b) CMOS implementation using only inverting gates.

after its original developer. For a CMOS implementation, we can redraw the Earle latch using only inverting gates (an inverter and four NAND gates) as shown in Figure 23.3(b).

From the schematic of Figure 23.3 we can now derive the timing properties of the latch. Let the delay of gate Ui in the figure be t_i - in practice we would calculate the delay of these gates as described in Section 5.4 — and the delays may be different for rising and falling edges and for different states. First consider the setup time, t_s. To meet the fundamental-mode restriction, after changing input d, the circuit must be allowed to reach a stable state before input g falls. For the circuit to reach a stable state the change in d (rising or falling) must propagate through gates U4, U5, and U3 in that order and both state variable q and the output of gate U3 must reach a stable state before g falls. This delay, and hence the setup time, is calculated as:

$$t_s = t_4 + t_5 + t_3. \tag{23.3}$$

This setup calculation illustrates some of the subtlety of asynchronous circuit analysis. One might assume that since d is connected directly to U3, the path from d to the output of U3 would be direct — d to U3. However, this is not the case when d is low. If d is low and the circuit is stable, then q is low. Thus, when d rises, it does not switch U3, because q, the other input of U3, is low. The change on d has to propagate through U4 and U5, making q high, before U3 switches. Hence the path that must stabilize is d to U4 to U5 to U3.

Now consider hold time. After g falls, the circuit must again come to a steady state before d can change again. In particular, if d is high, the change on g must propagate through U1 and U2 to enable the loop of U2 and U5 before d is allowed to fall. Hence the hold time is just:

$$t_h = t_1 + t_2. \tag{23.4}$$

To complete our analysis we see that each propagation delay is just the delay from an input g or d to the output q. For t_{dDQ} this path includes gates U3, U4, and U5, and for t_{dGQ} the path is through U4 and U5 (recall we are only

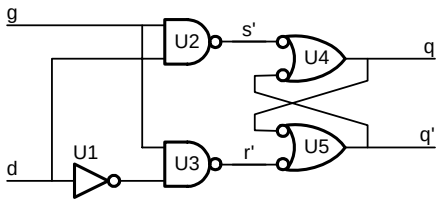


Figure 23.4: Latch built from an RS flip-flop.

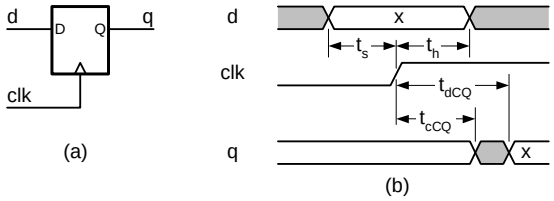


Figure 23.5: Edge-triggered D-flip flop. (a) Schematic symbol. (b) Timing diagram showing behavior.

worried about g rising here). Thus we have:

$$t_{dDQ} = \max(t_3, t_4) + t_5, \tag{23.5}$$

$$t_{dGQ} = t_4 + t_5. \tag{23.6}$$

An alternate gate implementation of a latch is shown in Figure 23.4. Here we construct a latch by appending a gating circuit to an RS flip-flop (Section 14.1). The RS flip-flop is formed by NAND gates U4 and U5. When the upper input of U4, $\overline{s}$, is asserted (low), the flip-flop is set ($q = 1$,$\overline{q} = 0$). When the lower input of U5, Br, is asserted (low), the flip-flop is reset ($q = 0$,$\overline{q} = 1$).

The gating circuit, formed by U1, U2, and U3, sets the flip-flop when $g = 1$ and $d = 1$, and resets the flip-flop when $g = 1$ and $d = 0$. Thus, when g is high, the output q follows input d. When g is low, the flip-flop is neither set nor reset and holds its previous state.

While the latch of Figure 23.4 has identical logical behavior to the Earle latch of Figure 23.3, it has very different timing properties. We leave derivation of these timing properties as Exercise 23–1.

23.2 Inside a Flip-Flop

An edge-triggered D-type flip-flop updates its output with the current state of its input on the rising edge of the clock. At all other times the output holds its current state. The schematic symbol and timing diagram of a D-flip-flop are

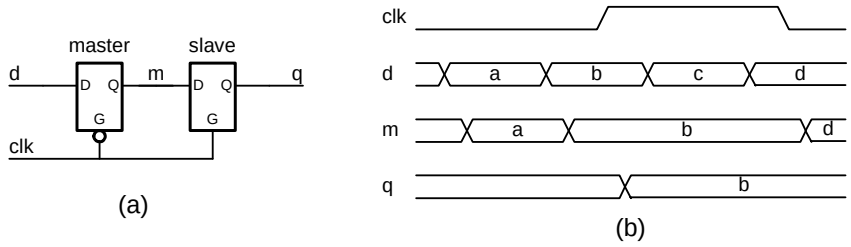


Figure 23.6: A master-slave D-flip-flop is constructed from two latches.

shown in Figure 23.5 (repeated from Figure 15.4). In Section 14.2 we saw how D-flip-flops are used for the state registers of synchronous sequential circuits and in Chapter 15 we looked at the detailed timing properties of the D-flip-flop. In this section, we derive a logic design for the D-flip-flop and see how this logic design gives rise to the timing properties of the flip-flop.

A latch with a negated enable does half of what we need to make a flip-flop. It samples its input when the enable rises and holds the output steady while the negated enable is high. The problem is that its output follows its input when the negated enable is low. We can use a second latch, in series with the first, to correct this behavior. This latch, with a normal enable, prevents the output from changing when the enable is low.

A D-flip-flop implemented with two latches in series with complemented enables is shown in Figure 23.6(a). Waveforms illustrating the operation of the flip flop are shown in Figure 23.6(b). The first latch, called the *master* samples the input on the rising edge of the clock onto intermediate signal m. In the waveforms, value b is sampled and held steady on m while the clock is high. Signal m, however follows the input when the clock is low. The second latch, called the *slave* (because it follows the master), enables the data captured on signal m onto the output when the clock is high. When the clock falls, it samples this value - still held steady by the master - and holds this value on the output when the clock is low. The net result of the two latches is a device that acts as a D-flip-flop. It samples the data on the rising edge of the clock and holds it steady until the next rising edge. The master holds the value when the clock is high - while the slave is transparent, and the slave holds the value when the clock is low - while the master is transparent.

For correct operation of the master-slave flip-flop, it is critical that the output of the master not change until t_h after the slave clock falls. That is, the hold time constraint of the slave must be met. If the output of the master latch were to change too quickly after the clock falls, the new value on intermediate signal m (d in the figure) could race through to the output before the slave latch blocks its flow. In practice this is rarely a problem unless there is a large amount of clock skew (Section 15.4) between the master and slave latches.

From the schematic of Figure 23.6(a) and the timing properties of the latch,

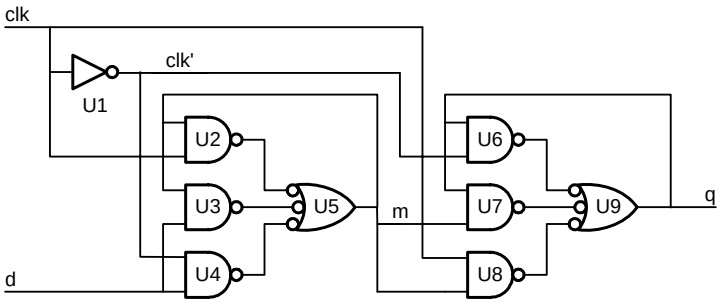


Figure 23.7: Gate-level schematic of a master-slave D-flip-flop.

we can derive the timing properties of the flip-flop. The setup and hold times of the flip-flop are just the setup and hold times of the master latch. This is the device doing the sampling. The delay of the flip-flop, t_{dCQ} is the delay of the slave latch t_{dGQ}.[2] The output q changes to the new value t_{dGQ} after the clock rises.

To illustrate the timing parameters of the D-flip-flop in a more concrete way, consider the gate-level schematic of Figure 23.7. This figure shows the master-slave flip-flop of Figure 23.6 expanded out to the gate level. The inverter on the enable input of each latch has been factored out to a single inverter U1 that generates $\overline{clk}$. The polarity of the clock connection to the master and slave are opposite.

The setup time t_s of the flip-flop shown in Figure 23.7 is the amount of time required for the circuit to reach a steady-state after d changes when clk is low.

Strictly speaking, the path here is through U4, U5, and U3 or U7 (if q is high). However, we don't really care if the output of U7 reaches steady-state or not since the clock is about to go high enabling U8. So we ignore U7 and focus on the path that ends at the output of U3 giving:

$$t_s = t_4 + t_5 + t_3 = t_{sm}. \tag{23.7}$$

where t_{sm} is the setup time of the master latch.

The hold time of this flip-flop is the time for the master part of the circuit to settle after the clock rises. This is just the maximum of delay of gates U1 and U2. If the data changes before U1 has brought $\overline{clk}$ low, state m might be affected. Similarly, the data cannot change until the output of U2 has stabilized or we may loose the value stored in the master latch. Thus we have:

$$t_h = \max(t_1, t_2) = t_{hm}. \tag{23.8}$$

where t_{hm} is the hold time of the master latch. Compared to (23.4) we don't

[2]This assumes that the setup time is large enough so that signal m is stable t_{s1} before the clock rises. See Exercise **??** for an example where this is not the case.

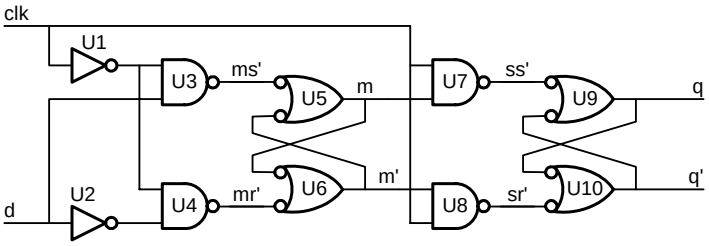


Figure 23.8: Alternate gate-level schematic of a master-slave D-flip-flop.

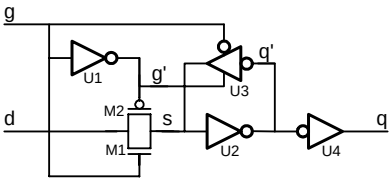


Figure 23.9: CMOS latch circuit using a transmission gate and a tri-state inverter.

add t_1 to t_2 here because with the low-true enable of the master latch, U1 is not in the path from *clk* to the output of U2.

For hold-time calculation we are only concerned with the master latch reaching steady state. When the clock rises, it may take longer for the slave to reach steady state. However, we will capture the data reliably as long as the master reaches steady state.

Finally, t_{dCQ} is the delay from the clock rising to the output of the flip-flop. The path here is through U8 and U9 giving:

$$t_{dCQ} = t_8 + t_9 = t_{dGQs}. \tag{23.9}$$

where t_{dGQs} is t_{dGQ} for the slave latch.

An alternate gate-level schematic for a master-slave D-flip-flop is shown in Figure 23.8. This circuit uses the RS-flip-flop based latch of Figure 23.4 for the master and slave latches of the flip flop. We leave the derivation of t_s, t_h, and t_{dCQ} for this flip-flop as Exercise 23–2

23.3 CMOS Latches and Flip-Flops

The latch circuits of either Figure 23.3 or Figure 23.4 using static CMOS gates. CMOS technology, however, also permits us to construct a latch with a transmission gate and a tri-state inverter as shown in Figure 23.9. Most CMOS

latches use transmission gates in this style because it results in a latch that is both smaller and faster than the alternative gate circuits.

When enable g is high (and $\overline{g}$ is low), the transmission gate formed by NFET M1 and PFET M2 is *on* allowing the value on input d to pass to storage node s. If $d = 1$, PFET M2 passes the 1 from d to s, and if $d = 0$, NFET M1 passes the 0 from d to s. Output q follows storage node s buffered by inverters U2 and U4. Thus, when g is high, the output q follows input d.

When enable g goes low the transmission gate formed by M1 and M2 turns off isolating storage node s from the input. At this time the input is sampled onto the storage node. At the same time, tri-state inverter U3 turns on, closing a storage loop from s back to s through two inverters. This feedback loop reinforces the stored value, allowing it to be retained indefinitely. The tri-state inverter is equivalent to an inverter followed by a transmission gate.

We can calculate the setup, hold, and delay times of the CMOS latch in the same manner as we did for the gate-based latches. When the input changes while g is high, the effect of the change on the storage loop must settle out before g is allowed to fall. The input change must propagate through the transmission gate, and inverter U2. We need not wait for U4 to drive output q since that doesn't affect the storage loop. Thus, the setup time is:

$$t_s = t_g + t_2, \tag{23.10}$$

where t_g is the delay of the transmission gate.

After g goes low, we need to hold the value on input d until the transmission gate is completely shut off. This is just the delay of inverter U1:

$$t_h = t_1. \tag{23.11}$$

We do not need to wait for feedback gate U3 to turn on. Its output, storage node s, is already in the correct state and there is sufficient capacitance on node s to hold its value steady until U3 turns on.

The delay times are calculated by tracing the path from input to output:

$$t_{dGQ} = t_1 + t_g + t_2 + t_4. \tag{23.12}$$

$$t_{dDQ} = t_g + t_2 + t_4. \tag{23.13}$$

A CMOS flip-flop can be constructed from two CMOS latches as shown in Figure 23.10. The master latch, formed by NFET M1, PFET M2, inverter U2, and tri-state inverter U3, connects input d to storage node m when enable g is low and holds m when g is high The slave latch, formed by NFET M3, PFET M4, inverter U4, and tri-state inverter U5 connects master latch state $\overline{m}$ to storage node $\overline{s}$ when enable g is high. An additional inverter, U6 generates output q. Output q is logically identical to node s at the output of U4. Isolating the output from the storage loop in this manner, however, is critical for the synchronization properties of the flip-flop as discussed in Chapter **??**. We leave the analysis of this flip-flop as Exercise 23–3.

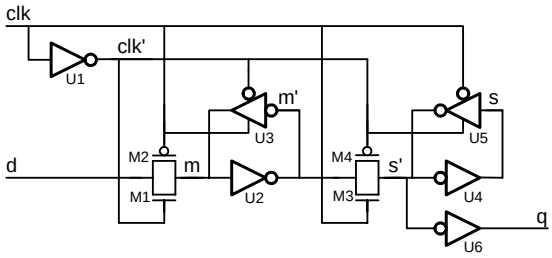


Figure 23.10: CMOS flip-flop circuit is constructed from two CMOS latches.

23.4 Flow-Table Derivation of The Latch*

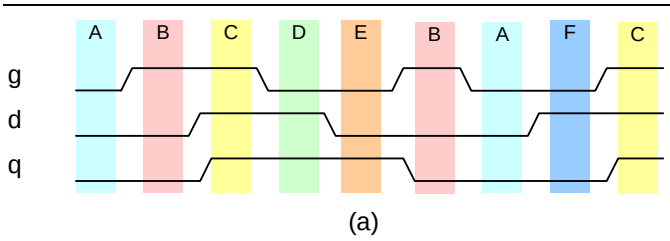
The latch and flip-flop are themselves asynchronous circuits and can be synthesized using the flow-table technique we developed in Chapter 22. Given an English-language description of a latch, we can write a flow-table for the latch as shown in Figure 23.11. To enumerate the states, we start with one state - all inputs and outputs low (state A) - and from this state toggle each input. We then repeat this process from each new state until all possible states have been explored.

In state A, toggling g high takes us to state B and toggling d high takes us to state F. This gives us the first line of the flow table of Figure 23.11(b). In state B, toggling g low takes us back to state A and toggling d high takes us to state C with q high. This gives us the second line of the flow table. We continue this way constructing the flow table line by line. until all states have been explored. The end result is the table of Figure 23.11(b).

In constructing this flow table we have created a new state for each input combination that didn't obviously match a previous state. This gives us six states, A-F, which would take three state variables to represent. However, many of these states are *equivalent* and can be combined. Two states are equivalent if they are indistinguishable from the inputs and outputs of the circuit.

We define equivalence recursively. Two states are *0-equivalent* if they have the same output for all input combinations. Here states A, B, and F are 0-equivalent as are states C, D, and E. A two states are *k-equivalent* they have the same output for all input combinations and their next-states for each input combination are $k-1$-equivalent. Here we see that the next states for A, B, and F aren't just equivalent, they are the same, so A, B, and F are also 1-equivalent. Similarly for C, D, and E.

In practice we find equivalent states by forming sets of states that are 0-equivalent (e.g., {A,B,F} and {C,D,E}) and then sets of states that are 1-equivalent, and so on. As soon as the sets don't change. That is, when we find a set of states that is both k-equivalent and $k+1$-equivalent we're done - these sets are equivalent.



(a)

State	Next (g d) 00	01	11	10	Out (q)
A	(A)	F		B	0
B	A		C	(B)	0
C		D	(C)	B	1
D	E	(D)	C		1
E	(E)	D		B	1
F	A	(F)	C		0

(b)

State	Next (clk d) 00	01	11	10	Out (q)
A	(A)	(A)	C	(A)	0
C	(C)	(C)	(C)	A	1

(c)

Figure 23.11: Flow table synthesis of a latch. (a) Waveforms, (b) Flow table, (c) Reduced flow table, (d) Karnaugh map, (e) Schematic.

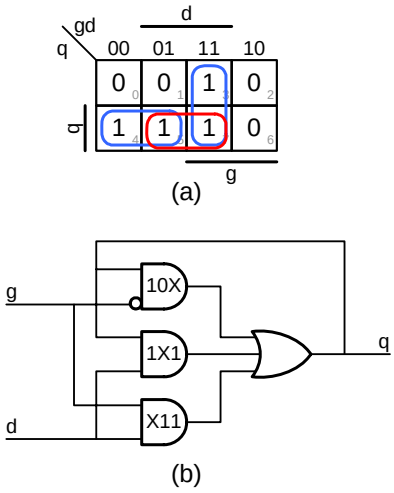


Figure 23.12: Logic design of a latch. (a) Karnaugh map. (b) Schematic diagram. The 1X1 implicant is needed to prevent a hazard.

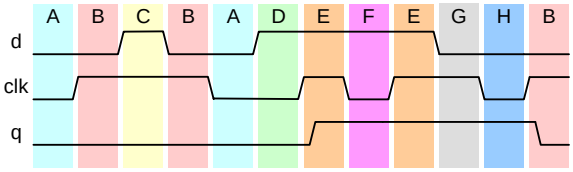
In this case, states A, B, and F are equivalent and C, D, and E are equivalent. Rewriting the flow table with just two states A and C (one for each equivalence class) gives us the reduced flow table of Figure 23.11(c).

If we use the output as the state variable, assigning state A an encoding of 0 and state C an encoding of 1, we can rewrite the reduced flow table of Figure 23.11(c) as the Karnaugh map of Figure 23.12(a). There are three implicants shown on the Karnaugh map. While we can cover the function with just two implicants (qgd = X11 $\vee$ 10X), we need the third, 1X1, to avoid a hazard (see Section 6.10). If the output were to momentarily dip low when the enable g input falls (going from qgd = 111 to 101), the end state could be 001 - with the latch loosing the stored 1. The added implicant (1X1) avoids this hazard. A schematic diagram for the latch circuit is shown in Figure 23.12(b).

23.5 Flow-Table Synthesis of a D-Flip-Flop*

Figure 23.13 shows the derivation of a flow table for a D-type flip-flop. We start by showing a waveform that visits all eight states shown in the flow table. As with the latch, we construct the flow table by toggling each input (clk and d) in each state. We create a new state for each input combination unless it is obviously equivalent to a state that we have already visited. The resulting flow table has eight states and is shown in Figure 23.13(b).

The next step is to find state equivalence classes. We start by observing that the 0-equivalent sets are $\{A, B, C, D\}$ and $\{E, F, G, H\}$. For 1-equivalence



(a)

State	Next (clk d) 00	01	11	10	Out (q)
A	(A)	D		B	0
B	A		C	(B)	0
C		D	(C)	B	0
D	A	(D)	E		0
E		F	(E)	G	1
F	H	(F)	E		1
G	H		E	(G)	1
H	(H)	F		B	1

(b)

State	Code	Next (clk d) 00	01	11	10	Out (q)
ABC	00	(A)	D	(C)	(B)	0
D	01	A	(D)	E		0
EFG	11	H	(F)	(E)	(G)	1
H	10	(H)	F		B	1

(c)

Figure 23.13: Flow-table synthesis of an edge-triggered D-flip-flop. (a) Waveforms. (b) Flow table. (c) Reduced flow table.

we observe that the next-state of D with an input of 11 is in a different class than that of the rest of its set. Similarly for H with an input of 10. Thus, the 1-equivalent sets become $\{A, B, C\}, \{D\}, \{E, F, G\}$, and $\{H\}$.

A reduced flow table with just these four states is shown in Figure 23.13(c). Our output q will be one of our state variables. We assign a second state variable as shown in the *Code* column of the table. Here q is the MSB of the two-bit state code and our new variable is the LSB.

Figure 23.14 shows how the flow-table of Figure 23.13(c) is reduced to logic. We start by redrawing the flow-table as a Karnaugh map with symbolic entries in Figure 23.14(a). Variable q is the output, and variable x is our additional state variable. The next step is to replace the symbolic next states with their state codes qx as shown in Figure 23.14(b). We next write down the Karnaugh maps for the two state variables in Figure 23.14(d) - q is on the left, x is on the right.

The two next-state variable functions can each be covered with two implicants. As with the latch, we must add a third to each to avoid a hazard. For variable q we have implicants $qxcd$ = X11X, 1X0X, and, to elimnate the hazard, 11XX. For x we have implicants X11X, XX01, and, for the hazard, X1X1. Drawing out these implicants as the logic diagram of Figure 23.14(d), we see that we have synthsized the master-slave D-flip-flop using Earle latches of Figure 23.7. The astute reader will notice that the top AND gate of the master latch and the bottom AND gate of the slave latch have identical inputs and can be replaced by a single shared AND gate.

23.6 Bibliographic Notes

Earle latch

Survey of FF circuits

23.7 Exercises

23–1 *Latch Timing Properties.* Compute t_s, t_h, t_{dDQ}, and t_{dGQ} for the latch of Figure 23.4 in terms of the delays of the individual gates. Assume that the delay of gate U_i is t_i.

23–2 *Flip-Flop Timing Properties.* Compute t_s, t_h, and t_{dCQ} for the D-flip-flop of Figure 23.8 in terms of the delays of the individual gates. Assume that the delay of gate U_i is t_i.

23–3 *CMOS Flip-Flop Timing Properties.* Compute t_s, t_h, and t_{dCQ} for the D-flip-flop of Figure 23.10 in terms of the delays of the individual gates (and transmission gates). Assume that the delay of gate U_i is t_i and the delay of each transmission gate is t_g.

23–4 *Flip-Flop Contamination Delay.* Consider a flip-flop constructed by placing a delay line with delay t_d between the master and slave latches. When

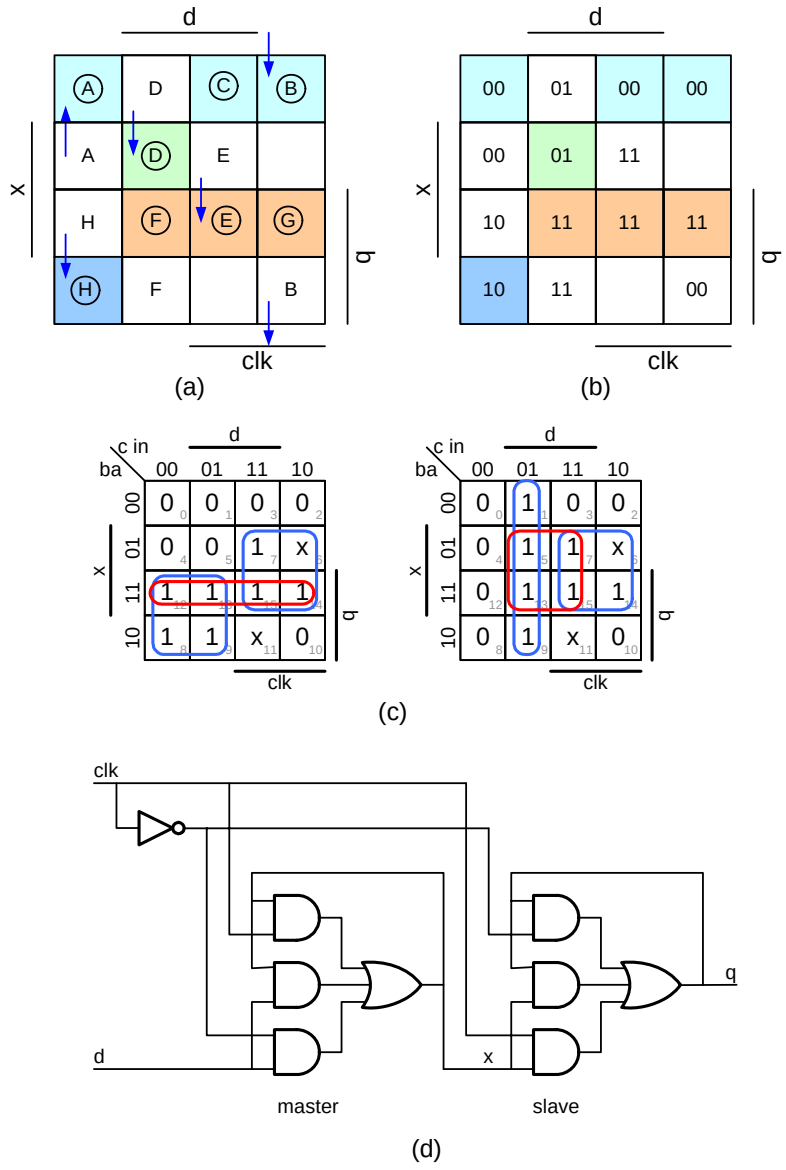


Figure 23.14: Deriving the logic design of a D-flip-flop from the flow-table of Figure 23.13(c). (a) Flow-table drawn on a Karnaugh map. (b) Karnaugh map showing next-state function. (c) Karnaugh maps for each bit of next-state function. (d) Logic diagram derived from Karnaugh maps.

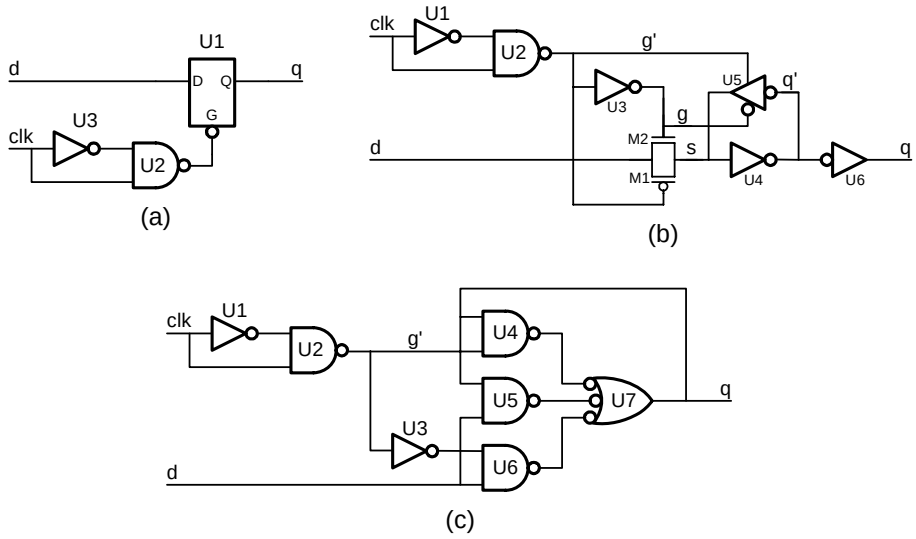


Figure 23.15: A pulsed-latch D-type flip-flop. (a) Schematic using latch symbol. (b) Schematic showing internals of CMOS latch. (c) Schematic showing internals of an Earle latch. When *clk* rises, NAND gate U2 generates a narrow low-going pulse on $\overline{g}$ that enables the latch to sample d.

the flip-flop input d changes exactly t_s before the clock rises, signal m at the input of the delay line becomes valid exactly at the clock edge, and signal $m1$ at the output of the delay line and input to the slave latch becomes valid t_d after the rising edge of the clock. What is the contamination delay t_{cCQ} and propagation delay t_{dCQ} of this modified flip-flop.

23–5 *Pulsed-Latch Flip-Flop.* One of my favorite D-flip-flop designs is shown in Figure 23.15. This flip flop consists of a single latch that is gated by a pulse generator. When *clk* rises, a narrow pulse is generated on enable $\overline{g}$ to sample d. The latch then holds the sampled value until the next rising edge of the clock. Answer the following questions about the circuit of Figure 23.15(c). Assume that the delay of gate U_i is t_i.
(a) What is the minimum pulse width generated by U2 for which this circuit will work properly?
(b) What is the maximum pulse width generated by U2 (if any) for which this circuit will work properly?
(c) Assuming that the pulse width out of U2 is the value you computed in part (a), compute t_s, t_h, and t_{dCQ} for this flip-flop and compare these values to the timing parameters of the master-slave flip-flop analyzed in the text. In particular compare the *overhead*, t_s+t_{dCQ} of the two flip-flops.

23–6 *CMOS Pulsed-Latch Flip-Flop.* Repeat Exercise 23–5 but for the CMOS pulsed latch of Figure 23.15(b).

23–7 *Combining Logic and Storage.*

23–8 *Two-Phase Clocking.*

23–9 *State equivalence.* Non-trivial state equivalence problem.

23–10 **SPICE analysis of setup and hold times.*

Chapter 24

Metastability and Synchronization Failure

What happens when we violate the setup- and hold-time constraints of a flip-flop? Up until now, we have considered only the normal behavior of a flip-flop when these constraints are satisfied. In this chapter we investigate the abnormal behavior that occurs when we violate these constraints. We will see that violating setup and hold times may result in the flip-flop entering a *metastable* state in which its state variable is neither a one or a zero. It may stay in this metastable state for an indefinite amount of time before arriving at one of the two stable states (zero or one). This *synchronization failure* can lead to serious problems in digital systems.

To stretch an analogy, flip-flops are a lot like people. If you treat them well, they will behave well. If you mistreat them, they behave poorly. In the case of flip-flops, you treat them well by observing their setup and hold constraints. As long as they are well treated, flip-flops will function properly never missing a bit. If, however you mistreat your flip-flop by violating the setup and hold constraints, it may react by misbehaving — staying indefinitely in a metastable state. This chapter explores what happens when these good flip-flops go bad.

24.1 Synchronization Failure

When we violate the setup or hold time constraint of a D-flip-flop we can put the internal state of the flip-flop into an *illegal* state. That is, the internal nodes of the flip-flop can be left at a voltage that is neither a 0 nor a 1. If the output of the flip-flop is sampled while it is in this state the result is indeterminate and possibly inconsistent. Some gates may see the flip-flop output as a 0 while others may see it as a 1 and still others may propagate the indeterminate state.

Consider the following experiment with a D-flip-flop: Initially both d and clk are low. During our experiment they both rise. If signal d rises t_s before clk the output q will be 1 at the end of the experiment. If signal clk rises t_h

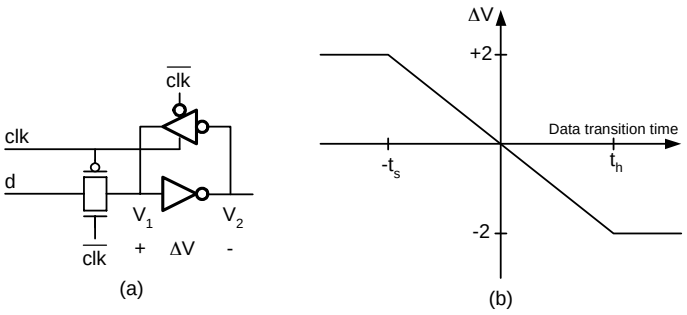


Figure 24.1: Abnormal operation of the master latch of a flip flop. (a) Schematic of the latch. (b) State voltage vs. data transition time. As the data transition time sweeps from t_s before the clock to t_h after the clock, the state voltage – i.e., the voltage across the storage inverters ($\Delta V = V_1 - V_2$) – changes from +2 to -2.

before d, the output q will be 0 at the end of the experiment. Now consider what happens as we sweep the rise time of d relative to clk from t_s before clk to t_h after clock. Somewhere during this interval, the output q at the end of our experiment changes from 1 to 0.

To see what happens when we change the input during this *forbidden interval*, consider the master latch of a CMOS D-flip-flop as shown in Figure 24.1(a). Here we assume that the supply voltage is 1V, so a logic 1 is 1V and a logic 0 is 0V. Figure 24.1(b) shows the initial state voltage of the flip-flop (the voltage across the inverter $\Delta V = V_1 - V_2$ the instant after the clock rises) as a function of data transition time. If d rises at least t_s before clk, the node lableled V_1 is fully charged to 1V and the node labeled V_2 is fully discharged to 0V before the clock falls. Thus the state voltage $\Delta V = V_1 - V_2$ is 2V. As d changes later - the state voltage is reduced as shown in Figure 24.1(b). At first, V_1 is still fully charged, but V_2 doesn't have time to fully discharge. As d rises still later, V_1 doesn't have time to fully charge. Finally, when d rises t_h after the clock V_1 doesn't have time to charge at all and we have $V_1 = 0$ and $V_2 = 1$ so $\Delta V = -2$.

The initial state voltage ΔV is a continuous function of the data transition time as it sweeps from $-t_s$ to t_h. It may not be an exact linear function as shown in the figure, but it is continuous, and it does cross zero at some point during this interval. Over this entire interval, the flip-flop has an initial state voltage that is not +2 or -2, its output is not a fully restored digital signal and may be misinterpreted by following stages of logic.

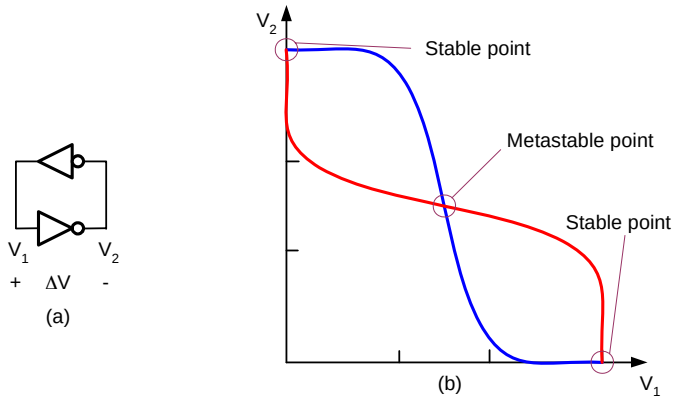


Figure 24.2: (a) The input latch of a flip-flop with the clock high acts as two back-to-back inverters. (b) DC transfer characteristics of the back-to-back inverters. The blue curve shows $V_2 = f(V_1)$ and the red curve shows $V_1 = f(V_2)$. The system has two stable points and one metastable point.

24.2 Metastability

The good news about synchronization failure is that most of the illegal states that our flip-flop can be left in after violating timing constraints decay quickly to a legal 0 or 1 state. Unfortunately it is possible for the circuit to be left in an illegal *metastable* state where it may remain for an arbitrary amount of time before decaying to a legal state.

After the clock rises, the latch of Figure 24.1(a) becomes a regenerative feedback loop (Figure 24.2(a)). The input transmission gate of the latch is off and the feedback tri-state inverter is enabled so that the equivalent circuit is that of two back-to-back inverters.

The DC transfer curve of these back-to-back inverters is shown in Figure 24.2(b). This figure shows the transfer curve of the forward inverter, V_2 as a function of V_1 ($V_2 = f(V_1)$ blue line), and the transfer curve of the feedback tri-state inverter, V_1 as a function of V_2 ($V_1 = f(V_2)$ red line). There are three points on the figure where the two lines cross. These points are *stable* in the sense that, in the absence of perturbations, the voltages V_1 and V_2 need not change. Since $V_1 = f(V_2) = f(f(V_1))$ the circuit can sit at any of these three points indefinitely.

At any point other than these three stable points, the circuit quickly converges to one of the outer two stable points. For example, suppose that V_2 is just slightly below the center point as shown in Figure 24.3. This drives V_1 to a point found by going horizontally from this point to the red line. This in turn drives V_2 to a point found by going vertically to the blue line, and so on. The circuit quickly converges to $V_1 = 1$, $V_2 = 0$.

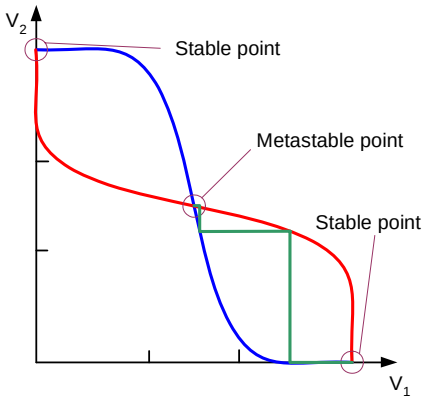


Figure 24.3: The dynamics of the back-to-back inverter circuit can be approximated by repeatedly applying the DC transfer characteristics of the two inverters as shown by the line on this figure.

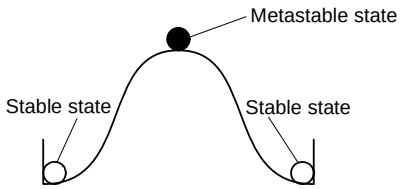


Figure 24.4: A ball at the top of a hill is in a *metastable* state. At rest, there is no force acting to push the ball left or right. However, a slight purturbation to the left or right will cause the ball to leave this state and fall to the left or right stable states at the bottom of the hill.

This iteration of the DC transfer curves is an oversimplification, but it gets the main point across. The circuit is stable at any of the three stable points. However, if we purturb the state slightly from either of the two end points, the state returns to that end point. If we disturb the state slightly from the midpoint, the state will quickly converge to the nearest end-point. A state, like the central stable state, where a small purturbation causes a system to leave that state is called *metastable*.

There are many physical examples of metastable states. Consider a ball at the top of a curved hill as shown in Figure 24.4. This ball is in a stable state. That is, it will stay in this state since, if its exactly centered, there is no force acting to pull it left or right. However, if we give the ball a slight push left or right (a perturbation), it will leave this state and wind up in one of the two stable states at the bottom of the hill. In these states, a small push will result

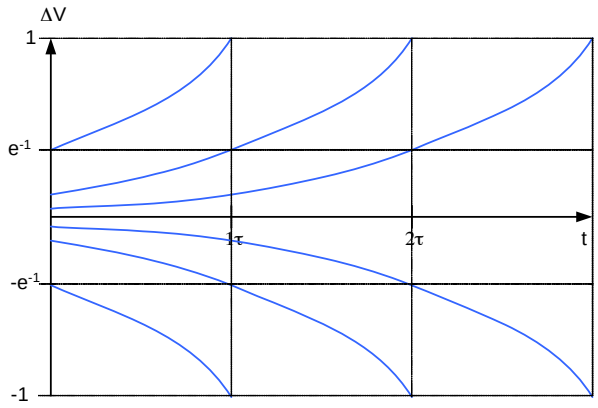


Figure 24.5: Plot of $\Delta V(t)$ for different values of $\Delta V(0)$. The magnitude of ΔV increases by e each time constant.

in the ball returning to the same state.

The ball on the hill is exactly analogous to our flip-flop. The flip flop in the metastable state (center point of Figure 24.2) is exactly like the ball on top of the hill. All it takes is one little push (or not being exactly centered to begin with) and the circuit *rolls* down hill to one of the two stable states.

The dynamics of the back-to-back inverter circuit are in fact governed by the differential equation:

$$\frac{d\Delta V}{dt} = \frac{\Delta V}{\tau_s}, \tag{24.1}$$

where τ_s is the time constant of the back-to-back inverters. In simple terms, the rate of change of the flip-flop state ΔV is directly proportional to its magnitude. The further it is from zero, the faster it moves away — until it is limited by the power supply.

The solution of this differential equation is:

$$\Delta V(t) = \Delta V(0) \exp\left(\frac{t}{\tau_s}\right). \tag{24.2}$$

This solution is plotted in Figure 24.6 for several initial values of ΔV. With the exponential regeneration of the back-to-back inverters, the magnitude of ΔV increases an e-fold each τ_s. Thus if the circuit starts with $\Delta V(0) = e^{-1}$ it takes time τ_s for the circuit to converge to $\Delta V(t) = 1$. Similarly if the circuit starts at $\Delta V(0) = e^{-2}$ it takes $2\tau_s$ to converge, and so on. Generalizing, we see that the amount of time that it takes to converge to $\Delta V = +1$ or -1 is:

$$t_s = -\tau_s \log\left(\Delta V(0)\right). \tag{24.3}$$

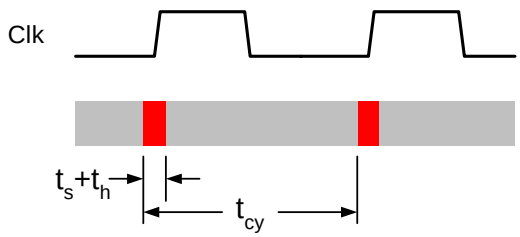


Figure 24.6: Sampling an asynchronous signal with a clock. If a transition of the asynchronous signal happens during the $t_s + t_h$ interval around the clock edge, our flip-flop may enter an illegal state.

24.3 Probability of Entering and Leaving an Illegal State

If we sample an asynchronous signal (one that can change at any instant), with a clock, what is the probability that our flip-flop will enter a metastable state? Suppose our clock period is t_{cy} and our flip-flop has setup time t_s and hold time t_h. Each transition of the asynchronous signal is equally likely to occur at any point during the cycle. Thus, there is a probability of:

$$P_E = \frac{t_s + t_h}{t_{cy}} = f_{cy}(t_s + t_h) \tag{24.4}$$

that a given transition will violate the setup or hold time of the flip-flop and hence cause it to enter an illegal state.

If the asynchronous signal has transitions with frequency $f_a\ s^{-1}$, then P_E of the asynchronous edges fall during the *forbidden* setup and hold region of the cycle. Thus, the frequency of errors (violating setup or hold time) is:

$$f_E = f_a P_E = f_a f_{cy}(t_s + t_h). \tag{24.5}$$

For example, suppose we have a flip-flop with $t_s = t_h = 100$ps and our cycle time is $t_cy = 2$ns. Then the probability of error is

$$P_E = \frac{t_s + t_h}{t_{cy}} = \frac{200ps}{2ns} = 0.1. \tag{24.6}$$

Now consider the case where the asynchronous signal has a transition frequency of 1MHz. Then the frequency of errors is

$$f_E = f_a P_E = 1\text{MHz}(0.1) = 100\text{KHz}. \tag{24.7}$$

We see that for realistic numbers, sampling an asynchronous signal can lead to an unacceptably high error rate.

As we shall see in Chapter 25, a partial solution to the problem of frequently entering illegal states is to *hide* the condition for a period of time, t_w, to allow the illegal state to decay to one of the two stable states. We can calculate the probability of a flip-flop still being in an illegal state after waiting time t_w by considering the probability of certain initial states and the time required for them to decay.

Given that a flip-flop enters an illegal state, it lands at a particular state voltage ΔV with some probability. We can (conservatively) estimate this probability distribution to be uniform. Then, from (24.3) the probability of taking longer than t_w to exit an illegal state is the same as the probability that

$$|\Delta V(0)| < \exp\left(\frac{-t_w}{\tau_s}\right).$$

With $|\Delta V(0)|$ uniformly distributed between zero and one, the probability of taking longer than t_w to reach a stable state is just:

$$P_S = \exp\left(\frac{-t_w}{\tau_s}\right). \tag{24.8}$$

For example, suppose we have a flip-flop with $\tau_s = 100$ps and we wait $t_w = 2ns = 20\tau_s$ for an illegal state to decay. To still be in an illegal state after $20\tau_s$, the flip-flop must have started with $|\Delta V(0)|$ smaller than $\exp(-20)$. With $|\Delta V(0)|$ uniformly distributed over $[0, 1]$, the probability of this is $P_S = \exp(-20)$.

24.4 A Demonstration of Metastability

To many students metastability is just and abstract concept until they see it for themselves. At that point, something clicks and they realize that metastabilty is real and it can happen to their flip-flops. This is best done via a laboratory exercise or a classroom demonstration. Hopefully you will have a chance to experience a live demonstration of metastabilty. If not, then this section will at least show pictures of what it really looks like.

figure of test setup

describe test setup

scope photos of metastable decay

24.5 Bibliographic Notes

24.6 Exercises

24–1 *Probability of Error.*

24–2 *Frequency of Error.*

24–3 *Time to Decay.*

24–4 *Probabiity of Exiting and Illegal State.*

24–5 *False synchronizer.* Metastable state in 1 region. Why doesn't this work.

Chapter 25

Synchronizer Design

In a synchronous system, we can avoid putting our flip-flops in illegal or metastable states by always obeying the setup- and hold-time constraints. When sampling asynchronous signals or crossing between different clock domains, however, we cannot guarantee that these constraints will be met. In these cases, we design a *synchronizer* that through a combination of waiting, for metastable states to decay, and isolation reduces the probablity of synchronization failure.

A brute-force synchronizer consisting of two back-to-back flip-flops is commonly used to synchronize single-bit signals. The first-flip-flop samples the asynchronous signal and the second flip-flop isolates the possibly bad output of the first flip-flop until any illegal states are likely to have decayed. Such a brute-force synchronizer cannot be used on multi-bit signals unless they are encoded with a Gray code. If multiple-bits are in transition when sampled by the synchronizer they are independently resolved, possibly resulting in incorrect codes with some bits sampled before the transition and some after the transition. We can safely synchronize multi-bit signals with a FIFO (first-in first-out) synchronizer. A FIFO serves both to synchronize the signals, and to provide flow control, ensuring that each datum produced by a transmitter in one clock domain is sampled exactly once by a receiver in another clock domain — even when the clocks have different frequencies.

25.1 Where are Synchronizers Used?

Synchronizers are used in two distinct applications. First, when signals are coming from a truly asynchronous source, they must be synchronized before being input to a synchronous digital system. For example, a push-button switch pressed by a human produces an asynchronous signal. This signal can transition at any time, and so must be synchronized before it can be input to a synchronous circuit. Numerous physical detectors also generate truly asynchronous inputs. Photodetectors, temperature sensors, pressure sensors, all produce outputs with transitions that are gated by physical processes, not a clock.

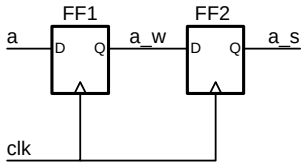


Figure 25.1: A brute-force synchronizer consists of two back-to-back flip-flops. The first flip flow samples asynchronous signal a producing signal a_w. The second flip flop waits one (or more) clock cycles for any metastable states of the first flip-flop to decay before resampling a_w to produce synchronized outptu a_s.

The other use of synchronizers is to move a synchronous signal from one *clock domain* to another. A clock domain is simply a set of signals that are all synchronous with respect to a single clock. For example, in a computer system it is not unusual to have the processor operate from one clock *pclk*, and the memory system operate from a different clock *mclk*. These two clocks may have very different frequencies. For example a *pclk* may be 2GHz while *mclk* is 800MHz. Signals that are synchronous to *pclk* — i.e., in the *pclk* clock domain — cannot be directly used in the memory system. It must first be synchronized with *mclk*. Similarly signals in the memory system, must first be synchronized with *pclk* before they can be used in the processor.

In moving signals between clock domains, there are two distinct synchronization tasks. If we wish to move a sequence of data where each datum in the sequence must be preserved, we use a *sequence synchronizer*. For example, to send eight words in sequence (one word at a time over a data bus) from the processor to the memory system we need a sequence synchronizer. On the other hand, if we wish to monitor the state of a signal we need a *state synchronizer*. A state synchronizer outputs a recent sample of the signal in question synchronized to its output clock domain. To allow the processor to monitor the depth of a queue in the memory system we use a state synchronizer - we don't need every sample of queue depth (one each clock), just one recent sample. On the other hand, we can't use a state synchronizer to pass data between the two subsystems, it may drop some elements and repeat others.

25.2 A Brute-Force Synchronizer

Synchronization of single-bit signals is often done with a *brute-force synchronizer* as shown in Figure 25.1. The flip-flop FF1 samples an asynchronous signal a producing output a_w. Signal a_w is unsafe because of the high frequency with which FF1 will enter an illegal state. To guard the rest of the system from this unsafe signal, we wait one (or more) clock periods for any illegal states of FF1 to decay before resampling it with FF2 to produce output a_s.

How well does the synchronizer of Figure 25.1 work? In other words, what is

the probability of a_s being in an illegal state after a transition on a? This will happen only if (1) FF1 enters an illegal state and (2) this state has not decayed before a_w is resampled by FF2. FF1 enters an illegal state with probability P_E (24.4) and it will remain in this state after a waiting time t_w with probability P_S (24.8). Thus, the probability of FF2 entering an illegal state is:

$$P_{ES} = P_E P_S = \left(\frac{t_s + t_h}{t_{cy}}\right) \exp\left(\frac{-t_w}{\tau_s}\right). \tag{25.1}$$

The waiting time t_w here is not a full clock cycle, but rather a clock cycle less the required overhead:

$$t_w = t_{cy} - t_s - t_{dCQ}. \tag{25.2}$$

For example, if we have $t_s = t_h = t_{dCQ} = \tau_s = 100$ps and $t_{cy} = 2$ns, then the probability of FF2 entering an illegal state is:

$$\begin{aligned} P_{ES} &= \left(\frac{t_s + t_h}{t_{cy}}\right) \exp\left(\frac{-t_w}{\tau_s}\right) \\ &= \left(\frac{100ps + 100ps}{2ns}\right) \exp\left(\frac{-1.8ns}{100ps}\right) \\ &= 0.1 \exp(-18) = 1.5 \times 10^{-9}. \end{aligned}$$

If signal a has a transition frequency of 100MHz, then the frequency of failure is:

$$f_{ES} = f_a P_{ES} = (100MHz)(1.5 \times 10^{-9}) = 0.15Hz. \tag{25.3}$$

If this synchonizer failure probability isn't low enough, we can make it lower by waiting longer. This is best accomplished by adding clock enables to the two flip flops and enabling them once every N clock cycles. This lengthens the wait time to

$$t_w = N t_{cy} - t_s - t_{dCQ}. \tag{25.4}$$

In our example above, waiting two clock cycles reduces our failure probability and frequency to:

$$P_{ES} = 0.1 \exp -38 = 3.1 \times 10^{-17}, \tag{25.5}$$

$$f_{ES} = (100MHz)(3.1 \times 10^{-17}) = 3.1 \times 10^{-9} Hz. \tag{25.6}$$

Using a clock enable to wait longer is more efficient than using multiple flip-flops in series because with the clock enables we only pay the flip-flop overhead $t_s + t_{dCQ}$ once, rather than once per flip-flop. Each clock cycle after the first adds a full t_{cy} to our waiting time. With flip-flops in series, each additional flip-flop adds $t_{cy} - t_s - t_{dCQ}$ to our waiting time.

How low a failure probability is low enough? This depends on your system and what it is used for. Generally we want to make the probability of synchronization failure significantly smaller than some other system failure mode. For

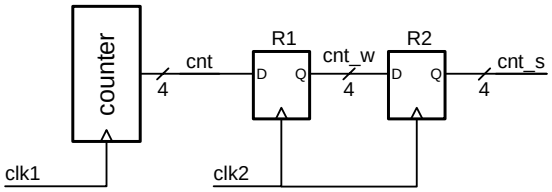


Figure 25.2: Incorrect method of synchronizing a multi-bit signal. Counter clocked by $clk1$ is sampled by synchronizer clocked by $clk2$. On a transition where multiple bits of signal cnt change, the synchronized output cnt_s may see some of the bits change, but not others giving an incorrect result.

example, in a telecommunication system where the bit-error rate of a line is 10^{-20} it would suffice to make a synchronizer with a failure probability P_{ES} of 10^{-30}. For some systems used for life-critical functions, the mean-time to failure (MTTF $= 1/f_{ES}$) must be made long compared to the lifetime of the system times the number of systems produced. So if the system is expected to last 10 years (3.1×10^8s), and we build 10^5 systems, we would like to make our MTTF much larger than 3.1×10^{13} (i.e., f_{ES} should be much less than 3×10^{-14}. Here we might set a goal of $f_{ES} = 10^{-20}$ (less than one failure every 10^{11} years - per system).

25.3 The Problem with Multi-bit Signals

While a brute-force synchronizer does a wonderful job synchronizing a single-bit signal, it **cannot** be used to synchronize a multi-bit signal unless that signal is Gray coded (i.e., unless only one bit of the signal changes at a time). Consider, for example, the situation shown in Figure 25.2. The output cnt of a four-bit counter clocked by one clock, $clk1$, is synchronized with a second clock, $clk2$, operating at a different frequency. Suppose on the count from 7 (0111) to 8 (1000) all of the bits of cnt are changing when $clk2$ rises - violating setup and hold times on register R1. The four flip-flops of R1 all enter illegal states. During the next cycle of $clk2$ these states all decay to 0 or 1 with high probability, so that when $clk2$ rises again, a legal four-bit digital signal is sampled, and output on cnt_s.

The problem is that with high probability the output on cnt_s is wrong. For each of the changing bits of cnt the synchronizer can settle to either a 0 or a 1 state. In this case, when all four bits are changing, the output of the synchronizer could be any number between 0 and 15.

The only time a brute-force synchronizer can be used with multi-bit signals is when these signals are guaranteed to change at most one bit between synchronizer clock transitions. For example, if we wish to synchronize a counter in this manner, we must use a Gray-code counter - that changes exactly one bit

```
module GrayCount4(clk, rst, out) ;
  input clk, rst ;
  output [3:0] out ;
  wire [3:0] out, next ;

  DFF #(4) count(clk, next, out) ;

  assign next[0] = !rst & !(out[1]^out[2]^out[3]) ;
  assign next[1] = !rst & (out[0] ? !(out[2]^out[3]) : out[1]) ;
  assign next[2] = !rst & ((out[1] & !out[0]) ? !out[3] : out[2]) ;
  assign next[3] = !rst & (!(|out[1:0]) ? out[2] : out[3]) ;
endmodule
```

Figure 25.3: Verilog description of a four-bit Gray-code counter.

on each count. A four-bit Gray-code counter uses the sequence 0, 1, 3, 2, 6, 7, 5, 4, 12, 13, 15, 14, 10, 11, 9, 8, ... Only one bit changes between adjacent elements of this sequence. For example, the eigth transition is from 4 (0100) to 12 (1100). Only the MSB changes during this transition. If we replace the counter of Figure 25.2 with a four-bit gray code counter, and the 4 to 12 transition takes place during a rising edge of $clk2$, then only the MSB of R1 will enter an illegal state. The low three bits will remain steady at 100. Either way the MSB settles, 0 or 1, the output is a legitimate value, 4 or 12.

Verilog code for a 4-bit Gray-code counter that generates this sequence is shown in Figure 25.3.

25.4 A FIFO Synchronizer

If we can't use a brute-force synchronizer on arbitrary multi-bit signals and our signal is not amenable to Gray coding, then how can we move it from one clock domain to another? There are several synchronizers that accomplish this task. The key concept behind all of them is removing synchronization from the multi-bit data path. The synchronization is moved to a control path that is either a single bit or a Gray-coded signal.

Perhaps the most common multi-bit synchronizer is the FIFO synchronizer. The datapath of a FIFO synchronizer is shown in Figure 25.4. The FIFO synchronizer works by using a set of registers R0 to RN. Data are stored into the registers under control of the input clock and read from the registers under control of the output clock. A *tail* pointer selects which register is to be written next, and a *head* pointer selects which register is to be read next. Data are added at the tail of the queue, and removed from the head of the queue. The head and tail pointers are Gray-coded counters that are decoded to one-hot to drive the register enables and multiplexer select lines. Using Gray-code encoding for

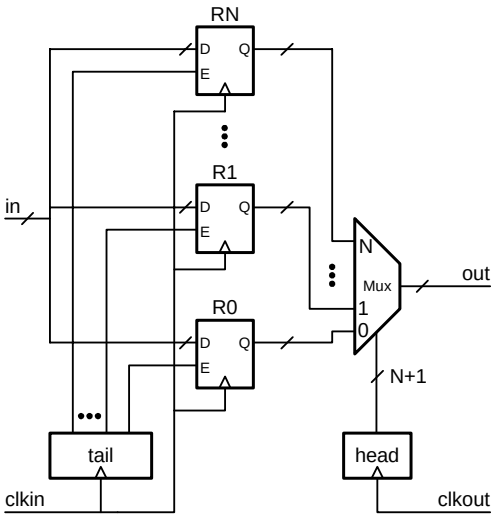


Figure 25.4: Data path of a FIFO synchronizer. Input data are placed in registers $R0, \ldots, RN$ by the input clock *clkin*. Output data are selected from among the registers by an output clock *clkout*. A control path, not shown, ensures that (1) data is placed in a register before it is selected for output and (2) that data is not overwritten before it is read.

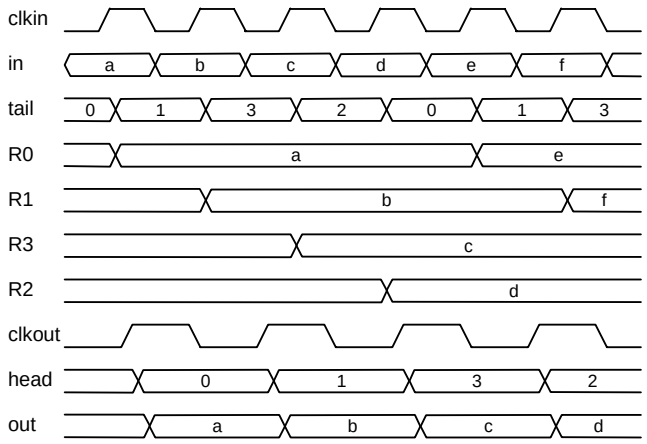


Figure 25.5: Timing diagram showing operation of a FIFO synchronizer with four registers.

the counters enables them to be synchronized using brute-force synchronizers in the control path.

A timing diagram showing operation of a FIFO synchronizer with four registers (R0 to R3) is shown in Figure 25.5. The input clock *clkin* in this example is faster than the output clock *clkout*. On each rising edge of the input clock, a new datum on line *in* is written to one of the registers. The register written is selected by the *tail* pointer which increments with a Gray-code pattern (0,1,3,2,0...). The first datum a is written to register R0, the second datum b is written to R1, c is written to R3, and so on. With four registers, the output of each register is valid for four input clocks.

On the output side, each rising edge of clock *clkout* advances the *head* pointer selecting each register in turn. The first rising edge of *clkout* sets head to 0 selecting the contents of register R0 a to be driven onto the output. The four input-clock valid period of R0 more than overlaps the one output-clock period during which it is selected, so no input-clock driven transitions are visible on the output. The only output transitions are derived from (and hence synchronous with) *clkout*. The second rising edge of *clkout* advances *head* to 1 selecting b from R1 to appear on the output, the third edge selects c from R3, and so on.

By extending the valid period of the input data using multiple registers, the FIFO synchronizer enables this data to be selected on the output without having *clkout* sample any signal with transitions synchronized with *clkin*. Thus, there is no probability of violating setup and hold times in this datapath.

Of course, we haven't eliminated the need to synchronize (and the probability of synchronization failure), we've just moved it to the control path. You will observe that with the input clock running faster than the output clock, our

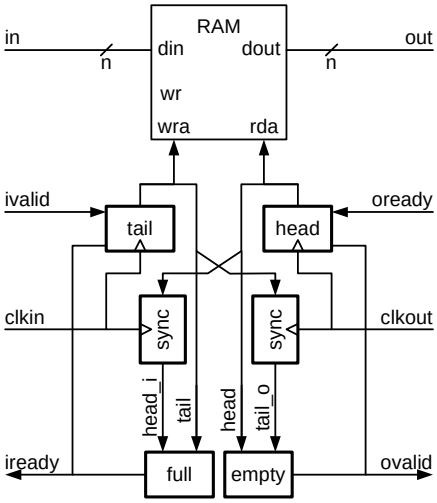


Figure 25.6: FIFO synchronizer showing control path.

FIFO synchronizer will quickly overflow unless we apply some *flow control*. The input needs to be stopped from inserting any more data into the FIFO when all four registers are full. Similarly, if *clkout* were faster than the input clock, we would need to stop the output from removing data from the FIFO when it is empty.

We add flow-control to our FIFO in the control path. A full block diagram of the FIFO including the control path is shown in Figure 25.6 (the registers have been grouped together into a RAM array) and the control path by itself is shown in Figure 25.7. In the control path we add two flow-control signals to each of the two interfaces. On both the input and output interfaces, the *valid* signal is true if the transmitter has valid data on the data line and the *ready* signal is true if the receiver is ready to accept new data. A data transfer takes place only if *valid* and *ready* are both true. On the input side, *ivalid* is an input signal, and *iready* is asserted if the FIFO is not full. On the output side, *oready* is an input signal, and *ovalid* is asserted if the FIFO is not empty.

The *iready* (not full) and *ovalid* (not empty) signals are generated by comparing the head and tail pointers. This comparison is complicated by the fact that head and tail are in different clock domains. Signal *head* is synchronous with *clkout* while *tail* is synchronous with *clkin*. We solve this problem by using a pair of multi-bit brute-force synchronizers to produce a version of *head* in the input clock domain *head_i* and a version of *tail* in the output clock domain *tail_o*. This synchronization is only allowed because head and tail are Gray-coded and thus will have only one bit in transition at any given point in time. Also, note that the synchronization delays the signals, so that *head_i* and *tail_o* are up to

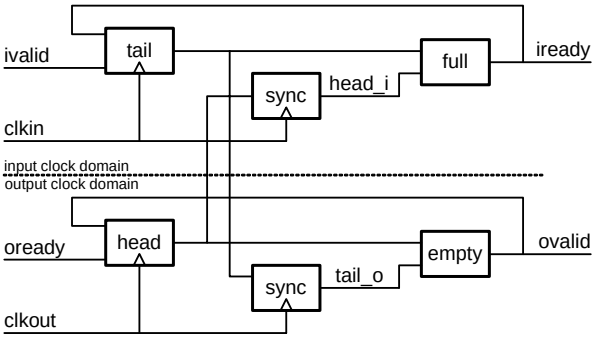


Figure 25.7: Control path of FIFO synchronizer showing clock domains.

two clock cycles behind *head* and *tail*.

Once we have versions of head and tail in the same clock domain, we compare them to determine the full and empty conditions. When the FIFO is empty, head and tail are the same, so we can write:

```
assign empty = (head == tail_o) ;
assign ovalid = !empty ;
```

When the FIFO is full, head and tail are also the same. Thus, if we were to allow all registers to be used, we would need to add additional state to discriminate between these two conditions. Rather than add this complexity, we simply declare the FIFO to be full when it has just one location empty and write:

```
assign full = (head_i == inc_tail) ;
assign iready = !full ;
```

where `inc_tail` is the result of incrementing the tail pointer along the Gray-code sequence. This approach always leaves one register empty. For example, with four registers, only three would be allowed to contain valid data at any time. Despite this disadvantage, this approach is usually preferred because of the high complexity of maintaining and synchronizing the extra state needed to discriminate between full and empty when `head == tail`. We leave to the reader, as Exercise 25–7, the design of a FIFO where all locations can be filled.

The Verilog code for the FIFO synchronizer is shown in Figure 25.9. The width and depth of the FIFO are parameterized — defaulting to 8-bits wide with 8 registers. A synchronous RAM module, not shown, contains the 8 registers. A register clocked by `clkout` holds `head` and one clocked by `clkin` holds `tail`. A pair of brute-force synchronizers creates signals `head_i` and `tail_o`. A pair of 3-bit Gray-code incrementers (verilog code in Figure 25.10) increments head and tail — along the Gray-code sequence — producing `inc_head` and `inc_tail`. Note that if this code were to be used for a FIFO deeper than 8 registers wider Gray-code incrementers would need to be used here — `GrayInc3` is not

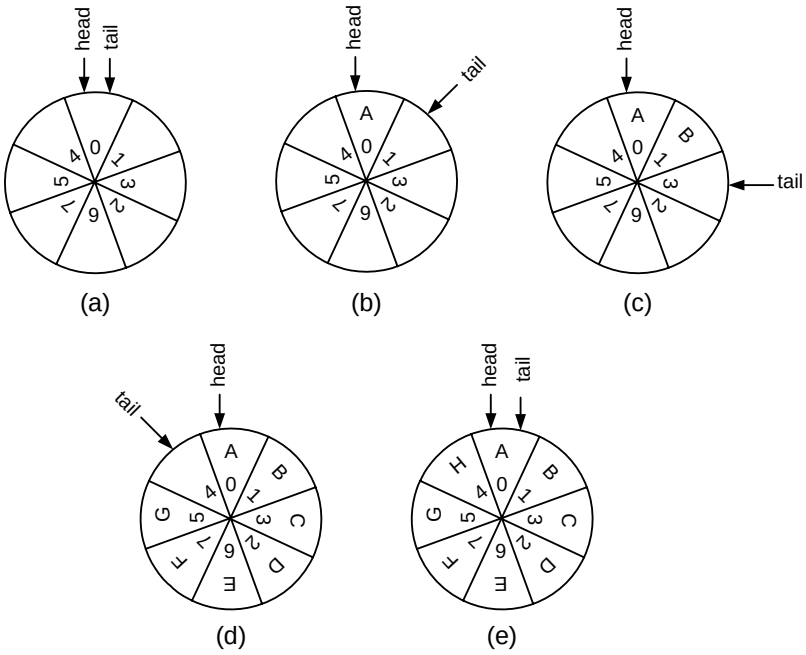


Figure 25.8: FIFO states. (a) FIFO is empty, `head == tail`, (b) After inserting one datum, (c) After inserting two data, (d) Almost full `head == inc(tail)`, (e) Completely full `head == tail`.

```
module AsyncFIFO(clkin, rstin, in, ivalid, iready,
                 clkout, rstout, out, ovalid, oready) ;

  parameter n = 8 ; // width of FIFO
  parameter m = 8 ; // depth of FIFO
  parameter lgm = 3 ; // width of pointer field

  input clkin, clkout, rstin, rstout, ivalid, oready ;
  output iready, ovalid ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  wire [n-1:0] out ;

  // words are inserted at tail and removed at head
  // head_i/tail_o is head/tail synchronized to other clock domain
  // inc_x is head/tail incremented by Gray code
  wire [lgm-1:0] head, tail, next_head, next_tail, head_i, tail_o ;
  wire [lgm-1:0] inc_head, inc_tail ;

  // Dual-Port RAM to hold data
  DP_RAM #(n,m,lgm) mem(.clk(clkin), .in(in), .inaddr(tail[lgm-1:0]),
                        .wr(iready&ivalid) ,
                        .out(out), .outaddr(head[lgm-1:0])) ;

  // head clocked by output, tail by input
  DFF #(lgm) hp(clkout, next_head, head) ;
  DFF #(lgm) tp(clkin, next_tail, tail) ;

  // synchronizers
  BFSync #(lgm) hs(clkin, head, head_i) ; // head in tail domain
  BFSync #(lgm) ts(clkout, tail, tail_o) ; // tail in head domain

  // Gray code incrementers
  GrayInc3 hg(head, inc_head) ;
  GrayInc3 tg(tail, inc_tail) ;

  // iready if not full, oready if not empty
  // input clock for full
  // full when head points one beyond tail
  assign iready = !(head_i == inc_tail) ;
  // output clock for empty
  assign ovalid = !(head == tail_o) ; // output clk

  // tail increments on successful insert
  assign next_tail = rstin ? 0 : (ivalid & iready) ? inc_tail : tail ;

  // head increments on successful remove
  assign next_head = rstout ? 0 : (ovalid & oready) ? inc_head : head ;
endmodule
```

Figure 25.9: Verilog description of a FIFO synchronizer.

```
module GrayInc3(in, out) ;
  input [2:0] in ;
  output [2:0] out ;
  assign out[0] = !(in[1]^in[2]) ;
  assign out[1] = in[0] ? !in[2] : in[1] ;
  assign out[2] = !in[0] ? in[1] : in[2] ;
endmodule
```

Figure 25.10: Verilog description of a 3-bit Gray-code incrementer.

parameterized. Next in the code, flow control signals `iready` and `ovalid` are computed as described above. Finally, the next state for `head` and `tail` are computed. The head and tail are only incremented if both the valid and ready signals for their respective interfaces are true.

Operation of the FIFO synchronizer is shown in the waveforms of Figure 25.11. On the input side, after reset, `iready` becomes true signaling that the FIFO is not full and hence able to accept data. Two cycles later, `ivalid` is asserted and seven data elements are inserted into the FIFO. At this point, the FIFO is full and `iready` goes low. On each data word inserted, `tail` is incremented in a 3-bit Gray-code sequence. Notice that `o_tail` is delayed by two output clocks from `tail`. Later, as words are removed from the FIFO on the output side, `iready` goes high again and additional words are inserted. Signal `ivalid` goes low for three cycles after 9 is inserted. During this period data is not input even though `iready` is true. Just before the end of the simulation, the input (which is running on a faster clock) gets 7 words ahead of the output and `iready` goes low.

On the output side, `ovalid` does not go high, signaling that the FIFO is not empty, until two output cycles after the first datum is inserted on the input side. This is due to the synchronizer delay of signal `tail_o` from which `ovalid` is derived. In the simulation we wait until the FIFO is full and then remove five words, one every other cycle. Note that from removing the first word (01), it takes two input cycles for `iready` to go high. This is due to the synchronizer delay of signal `head_i` from which `iready` is derived. After removing word 05, `iready` remains high for the remainder of the simulation and one word per cycle is removed.

25.5 Bibliographic Notes

There are many more types of synchronizers than we have had time to discuss here. See Dally and Poulton [] for a survey.

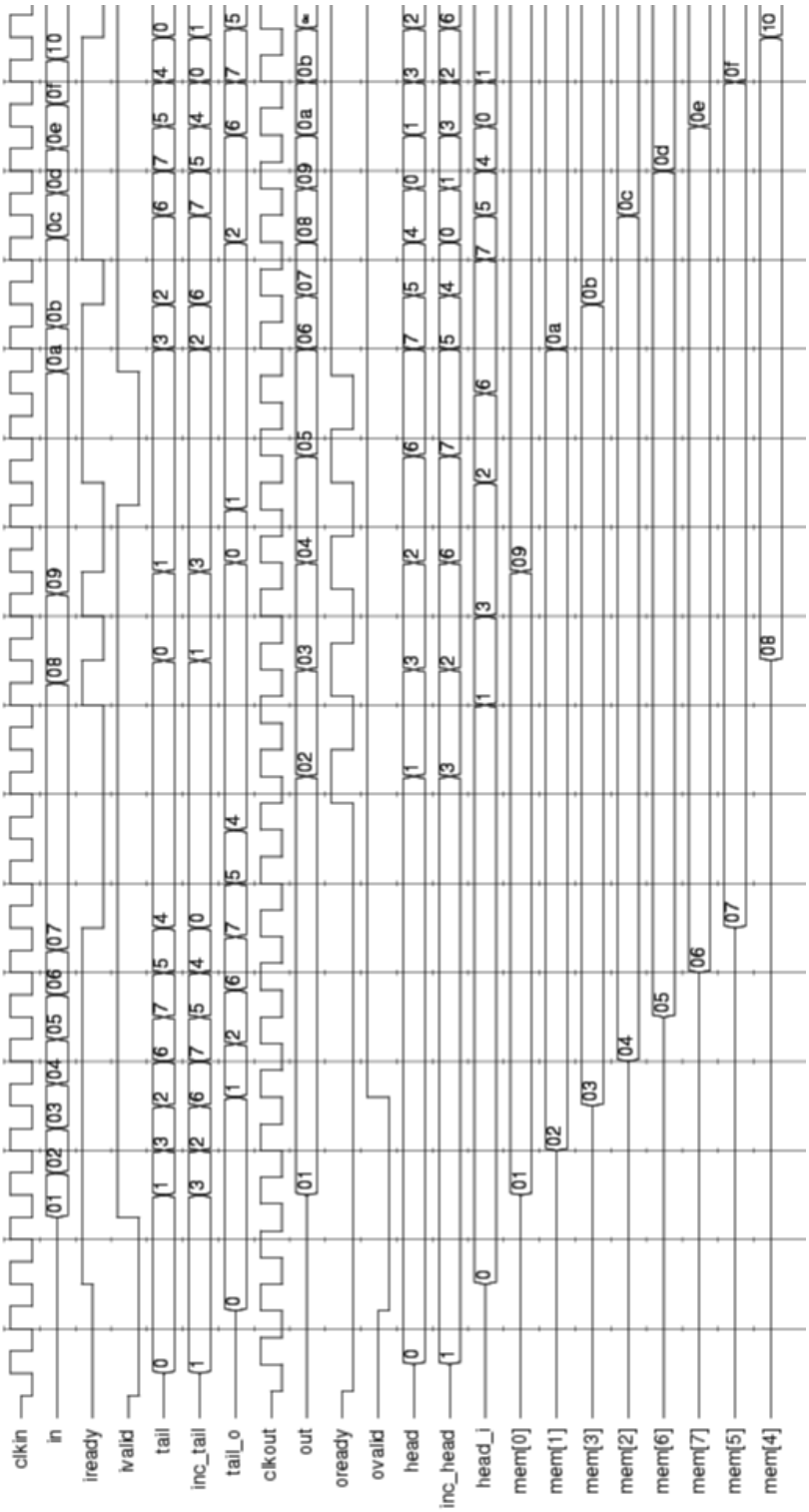


Figure 25.11: Waveforms from simulating the FIFO synchronizer of Figure 25.9

25.6 Exercises

25–1 *Brute force synchronizer.* Calculate failure probability.

25–2 *Brute force synchronizer.* Calculate failure frequency

25–3 *Brute force synchronizer.* Calculate cycles to wait for a given probability.

25–4 *Once-and-only-once synchronizer.* Design a once-and-only-once synchronizer. This circuit accepts and asynchronous input a and a clock clk and outputs a signal that goes high for exactly one clock cycle in response to each rising edge on the input a.

25–5 *Multi-bit synchronization.*

25–6 *FIFO synchronizer.*

25–7 *FIFO synchronizer.* Modify the logic of the FIFO synchronizer to allow the last location to be filled.

25–8 *Clock stopper.*