Title

Beginning Logic Design

Permalink

Author

Schlag, Martine

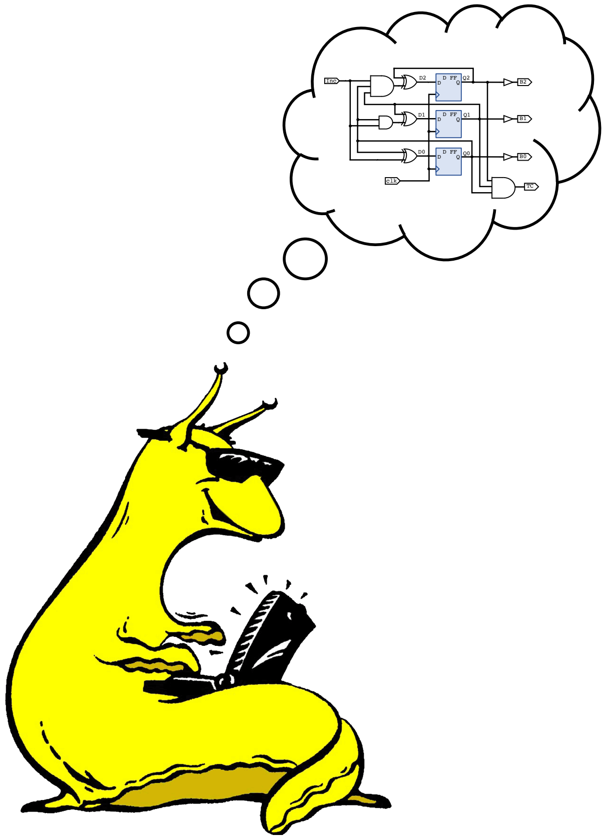
Publication Date

2024-09-06

Copyright Information

Beginning Logic Design

Martine Schlag



Contents

List of Figures

List of Tables

Foreword

These pages are based on 20 years of teaching the first course on digital design to engineering majors at UC Santa Cruz. Over this span, I have observed students abandon textbooks in favor of online searches for notes/videos or for answers in forums. While their cost is certainly an issue, textbooks on this subject often target current methodologies and technologies, perhaps even several. It's not surprising that students go elsewhere for explanations and answers to their questions at hand. Although there are many excellent sources available online, these resources have different conventions, notation, and assumptions that can easily confuse a novice. Searching online for an answer with little or no background can lead to incorrect or irrelevant material, wasting valuable time. For this reason, I wrote up my lecture notes for students to use as a reference. In these notes, the intent is to compartmentalize the topics as much as possible, get to the point quickly, and make the material efficiently accessible.

Since the 1990's students have been realizing their digital designs with FPGAs using software design tools. These tools have evolved from schematic entry with separate synthesis and simulation tools, to design environments that integrate entry, simulation, and synthesis (project managers), and now to HDL-only design entry. While HDLs have advanced and greatly facilitated hardware design, in a first course in logic design, it's important that the circuit structure is explicit and the implementation tools are transparent. For this reason, there is intentionally no HDL code in this text. Digital designs are presented as circuit diagrams, specification is given by boolean equations and state diagrams, and their behavior is revealed in waveforms. The development of logic design using these structural abstractions provides students with a firm basis for reasoning about logic and synchronization. It is these abstractions that have persisted over the last 20 years as tools have advanced.

Target Audience

At UC Santa Cruz, the first course in logic design assumes only the introductory course on computer organization (no physics or electronics courses). Basic logic gates are introduced without context and are used to assemble logic circuits. At UC Santa Cruz, discussion of the realization of gates and different technologies is relegated to the last two weeks of the quarter for the practical reason that in a 10-week quarter, it is a race to cover combinational and sequential logic sufficiently early so that students can complete a project involving multiple state machines. This arrangement also has two other benefits: the technology aspects are better absorbed once students understand circuit structures, and students new to electronics are more comfortable beginning with boolean logic.

Course infrastructure

The experience of conceiving, entering, and debugging a digital design is invaluable. A non-trivial design will rarely be correct on the first attempt. The process of analyzing and debugging will significantly clarify and deepen understanding of the operation of the design. Seeing one's design come to life and operate correctly (finally!) is often the positive reinforcement (joy) that fosters excitement for pursuing more advanced knowledge. While digital design can be studied with paper and pencil, using software tools and hardware to realize designs is highly recommended.

Design entry and simulation tools are a bare minimum. There are many free versions available for different programmable devices. Either schematic or language-based entry tools will do, though hardware languages are now the preferred medium. But for a beginning logic design class, restricting the language to structural constructs is recommended. Currently, in our classes, we use Verilog limited to assign statements and a library module for a D FF: no `always` blocks nor other statements. While hardware description languages are more efficient, for beginning logic design students it is important to be able to specify structure, without attempting to understand what the synthesizer will produce. By limiting the use of the language, the focus is on the structure of the design rather than the semantics of the description language.

Notes on topic selection

As mentioned no prior exposure to electronics is assumed. The development begins with the introduction of the NOT, AND, and OR logic gates as the building blocks. The dynamic behavior of logic elements is discussed early rather than later in the development. The transition from combinational circuits to sequential circuits is facilitated by viewing logic gates as continuously operating, especially when code is used to specify combinational logic.

With the use of design tools, even in beginning courses, mastering Kmaps (Karnaugh Maps) is no longer necessary for implementing designs. Yet a Kmap is a compact format that students are likely to encounter in documentation, and they are invaluable in exposing basic concepts in logic synthesis. Kmaps are introduced in Chapter 2 as an alternative format for truth tables. They are used in Chapter 3 for logic synthesis and in Chapter 5 to generate next-state and output equations for the examples in Sections 5.1 through 5.3. Beyond these examples, one-hot state encoding is used, and this allows the state equations to be directly derived from the state diagram (without Kmaps). Hence logic optimization (Chapter 3) can be skipped if desired, though I have found that this topic helps students better understand how their designs are being synthesized, and in particular, the effect that intentional and unintentional "Don't Cares" can have. For these reasons, Kmaps remain, followed by further optional topics. Chapters 1 through 3 contain optional sections that can be omitted: Sections 1.4, 2.11, 2.12, 3.4, 3.5, 3.6.

Chapter 1

Fundamentals

To get started quickly, assume that the basic logic gates (NOT, AND, OR) are available along with some means of assembling them. We will not now go into how these gates are made nor the technological aspects needed to keep them happy and cooperating in a logic circuit. Section 1.4 (which can be skipped) goes into some of the physical aspects that may be necessary for their operation.

1.1 Creating logic circuits

Our designs will be assembled by interconnecting logic components, either the basic logic gates or components we have constructed or been given. Logic components have pins (aka terminals) by which they can be connected to other components. Each component pin is one of two types, input or output. By convention, in our diagrams, the input pins are on the left and the output pins are on the right for each gate or component. The basic gates for the NOT, AND, OR logic operations will be represented in our logic diagrams using the standard gate symbols shown in Figure 1.1.

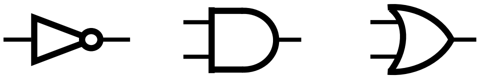


Figure 1.1: The three basic components we will use to build circuits: NOT, AND, OR.

As we will see, any logic function can be accomplished with just these three operations. However, it will be convenient to construct additional components as we build our projects. In our diagrams, we will use a symbol to represent a component we build. This is a box with the labeled input pins located on the left and the labeled output pins on the right.[1] In the diagram below the symbol for the component Vanilla has inputs a, b, and c. Its outputs are p and q.

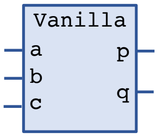


Figure 1.2: A symbol representing a logic circuit.

Our logic circuits are built by interconnecting the pins of components. In the diagram below we have assembled a circuit with two inputs, a and b, and two outputs, s and c. Note the labeled symbols in Figure 1.3 identifying the *external inputs* and *external outputs* of our circuit. This circuit forms a component that we have named HA and will be represented by the symbol below. The pins correspond to the labeled external inputs and outputs.

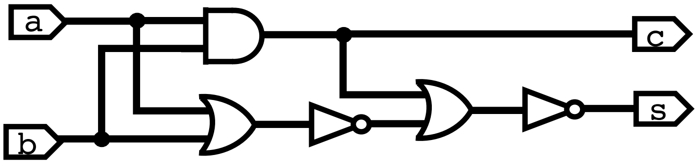


Figure 1.3: The logic diagram for the HA component.

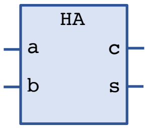


Figure 1.4: The symbol for the HA component.

In our diagram, the components are interconnected by drawing lines (wires) between their pins. A group of pins that have been connected forms a *net*. Each pin in the circuit belongs to only one net: if a pin belongs to two nets, then these two nets together should instead be one net. In Figure 1.5 each separate net of the HA circuit has been identified by assigning a color to the wires that interconnect the net's pins. A pin is the *source* of a net if it is either the output pin of a component or an external input. We will use the source pin to determine the logic value to associate with a net, so unless specifically allowed, we will assume that each net has at most one source. The

[1] It was not important to label the inputs of the basic gates because these operations are symmetric: swapping the connections of the two inputs has no effect.

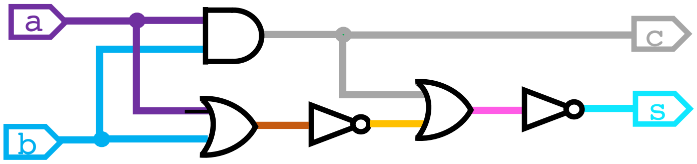


Figure 1.5: The nets of the HA logic diagram. Each net is a specific color.

other pins are called *loads* and will assume the same logic value as the net to which they belong. A net may have any number of loads.

We can use our components to build larger components rather than assembling gates directly. For example, we can build a circuit that performs the addition of three bits using two of the HA components from Figure 1.3. The resulting circuit at the gate level is shown below. It's the same as if we had drawn the logic diagram for the HA component twice.

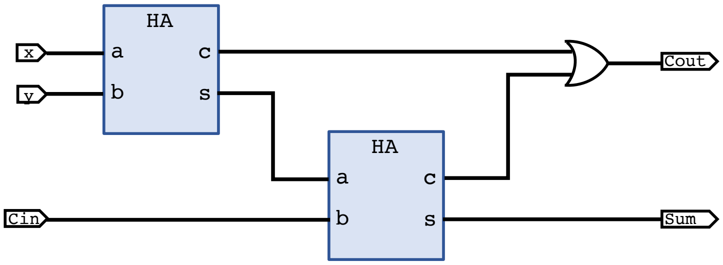


Figure 1.6: Logic diagram using HA components to sum three bits.

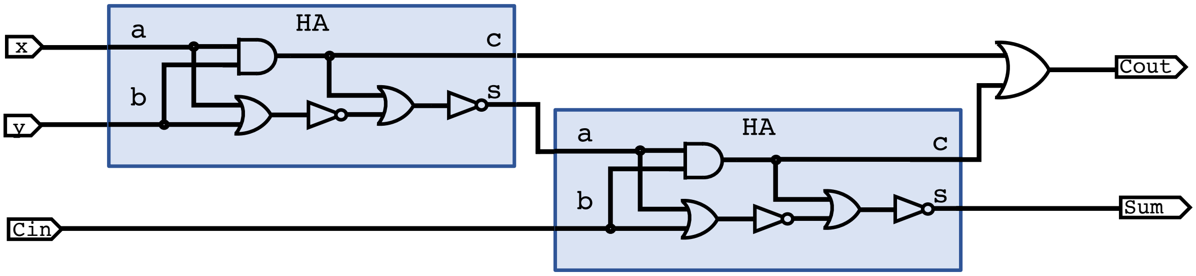


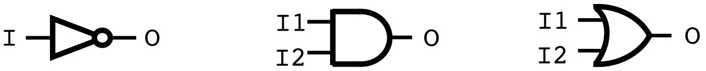
Figure 1.7: Flattened logic diagram to sum three bits.

Repeating regular structures allows us to easily assemble designs on a very large scale (VLSI).

1.2 Modeling the behavior of logic circuits

An output pin will have a value, either 0 or 1, since we are modeling our circuits at the logic level. The logic value of a net is the logic value of its source pin. When a net does not have a source pin we will use the value Z to indicate that the net and its loads are not *driven* to either 0 or 1. This might occur be an error, forgetting to provide a value for an external input, or it could be intentional. In some circumstances, we will not know the value associated with a net or pin even though it might be driven: we just don't happen to know whether it is driven to 0 or 1. This could occur when our circuit wakes up (power is first provided) or if we don't know the value of an external input. We use X to represent the unknown value.

When their input pins have been driven for a sufficient amount of time, the output pins of the basic gates will have values consistent with the logic operation they represent. These values are given for each of the gates in the tables in Figure 1.8.



I	O
0	1
1	0

I1	I2	O
0	0	0
0	1	0
1	0	0
1	1	1

I1	I2	O
0	0	0
0	1	1
1	0	1
1	1	1

Figure 1.8: Logic operation of the basic gates: NOT, AND, OR.

Our logic diagrams represent physical devices with outputs that vary over time, even possibly when their inputs are not changing. The basic gates will behave according to their expected logic operations, however, there is a delay between when an input value changes and when the output value changes as a result. Figure 1.9 below has four successive snapshots for the values of an AND gate's pins. Initially, both inputs of the AND gate are 1 and the output is also 1. This is consistent with the operation of an AND gate. At time 13.0 input a is 0 but the output c is 1; c's value is inconsistent with the value of the inputs at time 13.0. At time 13.4 the output c is now changed to 0 which is consistent.

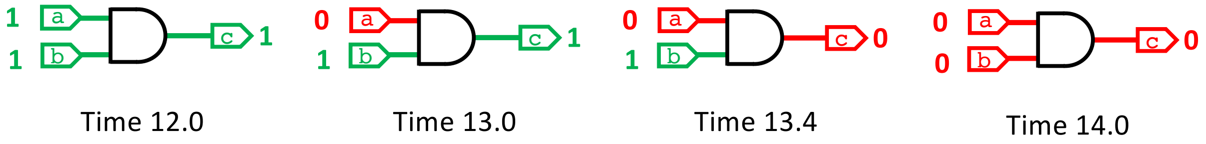


Figure 1.9: Four snapshots in the operation of an AND gate.

The snapshots in Figure 1.9 do not tell us when the values of the pins changed nor even if they changed multiple times. Instead of snapshots, we use *timing diagrams* to represent the logic value of pins over time. The timing diagram in Figure 1.10 is consistent with the four snapshots above. The horizontal scale along the bottom is the time in appropriate units. The values of the three pins are represented as horizontal lines (also called *waveforms*). Each of the values we are displaying (signals) is either 0 or 1. In this diagram, we see that `a` changed from 1 to 0 at time 12.8 and that the output `c` changed from 1 to 0 at time 13.2. And there were no other changes (aka transitions) on `a` or `c`. Note that `b` changes at time 14.0 but this does not cause any change on the output `c`.

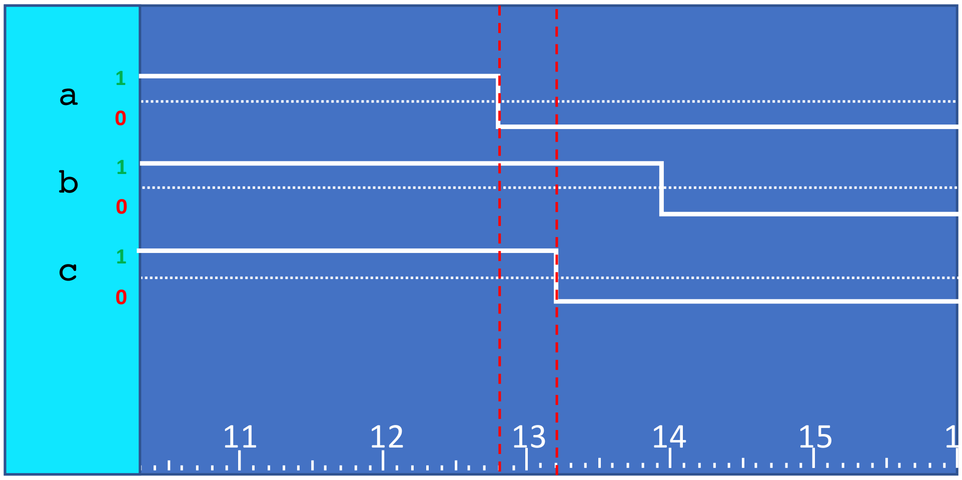


Figure 1.10: Timing diagram showing the continuous operation of an AND gate.

To keep our discussion independent of any particular technology we will assume that the basic gates provided to us faithfully implement the corresponding logical operations with some amount of delay d_g. Specifically,

> **Gate Model** The logic level of the output pin of a basic gate at time t is given by the gate's logic operation on the logic level of the gate's input pins at time $t - d_g$, where d_g is the constant delay associated with the gate.

As an example, Figure 1.11 shows the waveforms associated with the inputs of an AND gate, `a` and `b`. The waveform labeled `a·b` shows the AND operation applied to the logic values of `a` and `b`. The waveform labeled `c` is obtained by shifting the waveform `a·b` by 0.4 time units. Thus the waveform `c` is the output of the AND gate with inputs `a` and `b` and delay $d = 0.4$ in our Gate Model.

Section 1.4 discusses why this model may not always hold depending on the particular technology. Adopting this model does not mean that an output pin of a component can only change as a result of a change in one of its inputs.

In the circuit and timing diagram in Figure 1.12, the output of the AND gate is 0 while the external input `a` is 0. When `a` becomes 1, the output of the AND gate will alternate between 0 and 1, and continue to alternate despite no further change to the input `a`. The alternating behavior here is due to the feedback in this logic circuit. In analyzing and designing logic circuits we will separate

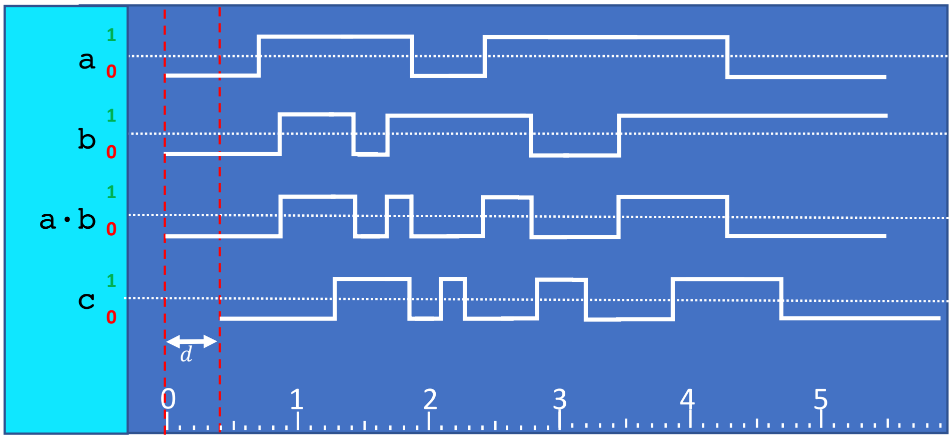


Figure 1.11: Timing diagram with Gate Model output of an AND gate.

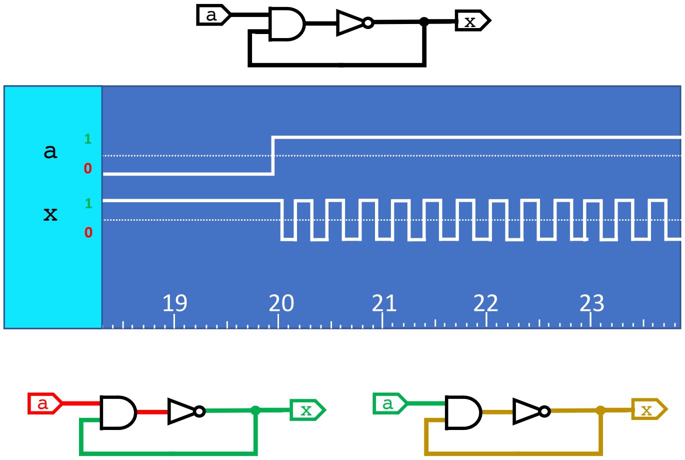


Figure 1.12: A logic circuit output can oscillate even while its inputs are steady

logic circuits into two classes according to whether they have feedback.

1.3 Combinational versus Sequential Logic

To simplify the analysis and synthesis of logic we classify circuits as either *combinational* or *sequential*.[2] In Chapters 2 and 3 we will analyze and synthesize combinational circuits, while Chapters 4 and 5 describe synchronous sequential design.

A combinational logic circuit does not have feedback. Its gates can be assigned numbers so that each gate's inputs are either connected to an external input or the output of a gate with a lower

[2] The term "combinatorial" is also used, but is less common.

number.[3] In addition, each component must itself be a combinational circuit. Our three basic gates are combinational. In Figure 1.13 below, the numbers indicate an ordering that satisfies the requirement. This is not the only such order. The AND gate could have been first or between the first OR and NOT gates. There is no such ordering for the gates in Figure 1.12. The AND gate and NOT gate each has inputs from the other.

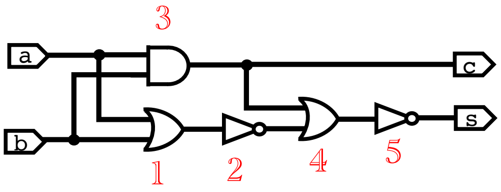


Figure 1.13: Ordering of gates in a combinational circuit

A key property of a combinational circuit is that when its external inputs are stable (not changing value) beginning at time t, all of the output values of its gates will also be stable beginning at time $t+D_c$ for some constant D_c. We say that the circuit has *settled* by time $t+D_c$. In our model, a gate output is stable by time $t+d_g$ if its inputs are stable beginning at time t. Hence in a combinational circuit, a gate output will be stable at time $t+D_c$ where D_c is at least as large as the sum of the gate delays along any path from an external input to that gate. The number of gates on a path as well as the type of gates affects this number D_c, and keeping D_c as small as possible is often a design goal.

A combinational logic circuit is "memory-less" in that the values of its outputs at time $t+D_c$ are determined completely by the value of its external inputs at time t. The values of the inputs before time t have no effect. Determining the output values of a combinational circuit and D_c is covered in Chapter 2. The circuit in Figure 1.12 is not memoryless. The value of the output `x` at any later time depends on knowing that the input `a` changed from 0 to 1 at time 20.

1.4 Technology aspects that might matter

We have assumed very little about our basic gates so that we can quickly focus on logic design. The subsequent chapters do not depend on anything in this section, so it can be safely skipped. But keep reading if you are interested in some of the technology-dependent aspects of implementing digital logic.

Logic has been built with several different electronic technologies such as relays, vacuum tubes, transistors, and even non-electronic ones such as hydraulics and Lego blocks. However, it is the miniaturization of the transistor that has provided the ability to assemble extremely large circuits. Typically, a logic design course will start with a discussion of transistors and switching circuits. Instead, we have chosen to begin with the basic logic gates without explaining how the logic values 0 and 1 are represented nor how they are transmitted between gates.

[3] You may have encountered this numbering in other domains and recognize it as a *topological sort*.

This makes our discussion independent of technology, but there are physical issues that may arise and constrain our designs depending on the scale.

First, it's important to remember that our gates are physical devices that require energy. Low power design has been an important focus of circuit design in the past decade, both to support longer battery life as well as reduce our overall energy consumption. A more immediate power concern in classroom lab settings is to remember to connect power and ground when we are implementing our circuits from discrete parts, or switching on the prototyping board we are using![4]

For the electronic devices, logic values 0 and 1 correspond to voltage levels and a current requirement. Metal wires are used to transmit these values. When implementing logic with discrete parts, it's important to ensure that the parts are compatible in terms of their voltage/current requirements so that they faithfully communicate their logic values. Discrete parts belong to logic families that are designed to work together. Using parts from different families may require additional circuitry to translate. The voltage/current requirements of input/output devices (e.g. switches, buttons, LEDs) must also be compatible.

The number of loads (input pins) on a net is its *fanout*. Nets with large fanouts can affect the delay of the circuit as well as whether the circuit operates at all. Depending on the discrete part type there may be a limit on the number of loads one output pin can drive due to current requirements. On an FPGA large fanout nets can be handled with specialized routing resources, but these resources are not unlimited. We have not mentioned wiring as a source of circuit delay since, generally, gate delays will dominate wiring delays. However, wiring delays are not always negligible and large fanout can result in wiring delays that significantly affect circuit speed.

As mentioned our gate model does not completely reflect reality. Gates are physical devices that have inertia. The delay before the gate output responds (changes its output to a new logic value) may depend on the direction of the change (from 0 to 1, or 1 to 0) and the number of loads and wiring of the net associated with the output pin. The behavior of the inputs may also have an effect. For example, if an input pin of a NOT gate transitions from 1 to 0 and then back to 1 in a sufficiently short time, the NOT gate output may never become 1. Hence the timing diagram in Figure 1.12 may not reflect what is occurring in the circuit. The delay of the gates in this circuit and possibly the wiring will affect whether these gates have 0 or 1 logic levels, and also the degree to which oscillation occurs. To fully understand the behavior of the circuit in Figure 1.12 we need to know more about the physical properties of the gates and model the circuit in much greater detail than our gate model provides.

[4] This is mentioned because of the surprising number of times this has turned out to be why nothing was happening.

Chapter 2

Combinational Logic

2.1 Introduction

In a combinational circuit, we can sort the gates so that each gate's inputs are either external inputs to the circuit or outputs of previous gates in the sorted order. There are no "loops" in the circuit (aka feedback), and so we can determine the output value of each gate by starting at the external inputs and tracing through the circuit's gates in the sorted order assigning each gate output the value consistent with its inputs' values. A circuit is said to have *settled* once all of its gates have output values consistent with their input values.

Figure 2.1 has four views of our HA circuit from Figure 1.3. Each view shows the logic values of the nets for one of the four possible input values to HA once all of the gates have settled.

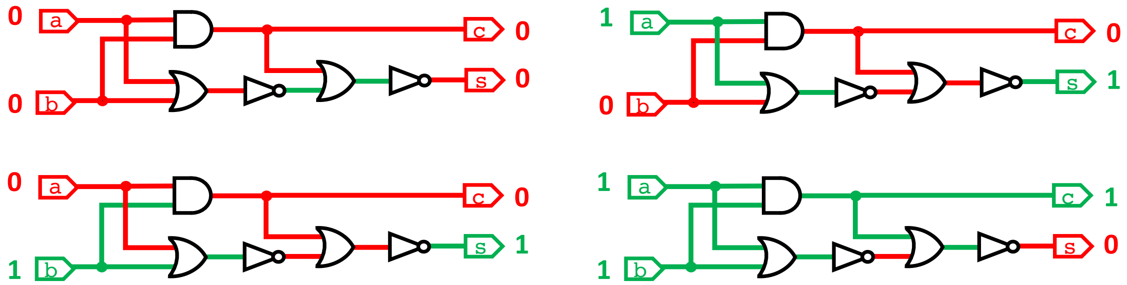


Figure 2.1: Four views of the HA logic diagram.

Once a combinational circuit has settled, as long as the external inputs remain stable (unchanged), the circuit's gate outputs will not change value. These values are determined solely by the values on the external inputs. We will see the same output values whenever the circuit has settled with these input values. The output values of a settled combinational circuit are determined solely by the values on its external inputs, while the output of a sequential circuit will also depend on their previous values. In this respect, combinational logic is "memory-less" while sequential logic has "state."

In analyzing and synthesizing combinational circuits we will focus first on their settled behavior. The *transient* behavior of combinational circuits is discussed in Section 2.10.

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 2.1: Truth table for the settled values of the HA component.

The settled behavior of HA can be summarized in a *truth table* as in Table 2.1. There is a column for each input and then a column for each output. There are four possible values for the two inputs, each listed in a separate row along with the associated output values.

2.2 Boolean functions

We can think of a combinational logic circuit as a mathematical function of its inputs by considering only its settled behavior. Since the values of inputs and outputs will be 0 or 1, this mathematical function is a boolean function [1].

A *boolean function* of n variables is a function mapping n-tuples of 0's and 1's to either 0 or 1.[1]

Our HA circuit implements two boolean functions of 2 variables (a and b), one function for output c and another function for output s.

Example 1 The *majority function* of n inputs has output values 1 when more than half of its inputs are 1, and has output value 0 otherwise.

Example 2 The *parity function* of n inputs has an output value of 1 when an odd number of its input variables are 1 and has an output value of 0 otherwise. (Recall that zero is an even number.)

The parity function is a usual suspect in arithmetic circuits. It even has its gate symbol (XOR). You may have noticed that the s output of the HA circuit is the parity of 2 inputs.

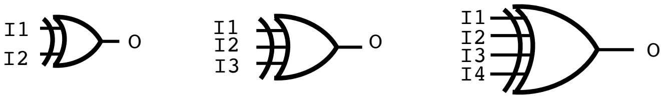


Figure 2.2: The XOR gate for 2, 3, and 4 inputs.

[1] We can think of n-tuples of 0's and 1's as bit vectors of length n.

2.3 Truth tables and Kmaps

A common way to represent a boolean function is a truth table. We've already seen it used to describe the operation of basic gates as well as the behavior of our HA circuit. A simple truth table has a column for each of the inputs, and then each of the outputs. There is a row for each of the possible input values. In each row, the associated output value of the output(s) is given. Below are the truth tables for the majority and parity functions of 1, 2, and 3 variables.

a	$f_{M1}(a)$
0	0
1	0

a	b	$f_{M2}(a,b)$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	c	$f_{M3}(a,b,c)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

a	$f_{P1}(a)$
0	0
1	1

a	b	$f_{P2}(a,b)$
0	0	0
0	1	1
1	0	1
1	1	0

a	b	c	$f_{P3}(a,b,c)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

A *Karnaugh-map* (Kmap for short) is a truth table in a different format [2]. This format will be practical for synthesizing logic. In a Kmap, the truth table is organized as a grid with 2^n cells. Each cell is associated with one of the possible input values. The format of a 2-variable Kmap is shown on the left in Figure 2.3. Within each cell, the label in its top left corner is the value of the variables a and b for that cell. A Kmap can be divided into half according to the value of a single variable. On the right in Figure 2.3 the halves for a=1 and b=1 have been shaded. The format

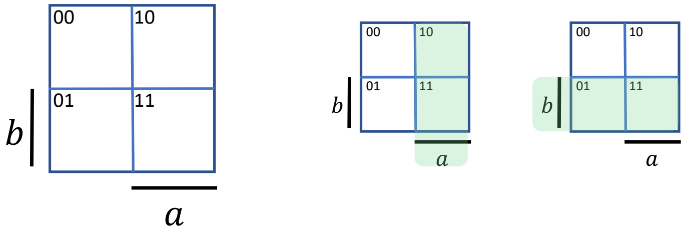


Figure 2.3: Format of the 2-variable Kmap and the regions in which its variables are 1.

of 3-variable and 4-variable Kmaps are shown in Figures 2.4 and 2.5.[2] The division of the Kmap according to each variable is shown on the right. The order of the columns and rows may seem unusual. They are arranged so that all of the cells corresponding to a fixed variable value form one region provided we imagine that the left and right columns are next to each other as well as the top and bottom rows. In a Kmap, two cells that share a side will have input labels that differ in exactly one variable. For example the cell labeled `0110` in Figure 2.5 has neighboring cells with labels `0010`, `0111`, `0100`, and `1110`. The Kmap for a boolean function of n variables is obtained by filling the cells of an n-variable Kmap with the output value corresponding to the input label of each cell. In Figure 2.6, the truth tables for our `HA` outputs are shown as Kmaps. The truth tables for our 3-variable majority and parity functions are shown as Kmaps in Figure 2.7.

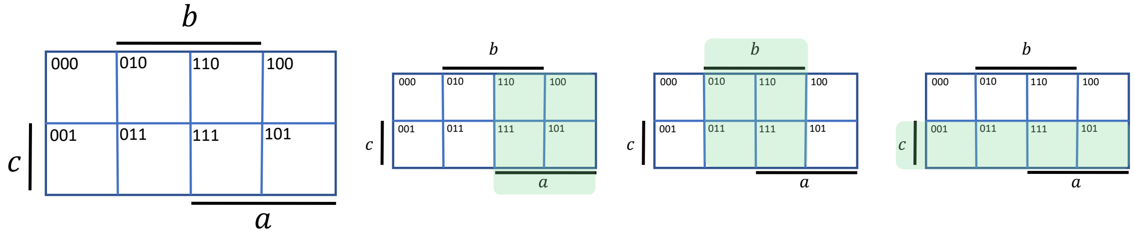


Figure 2.4: Format of the 3-variable Kmap and the regions in which its variables are 1.

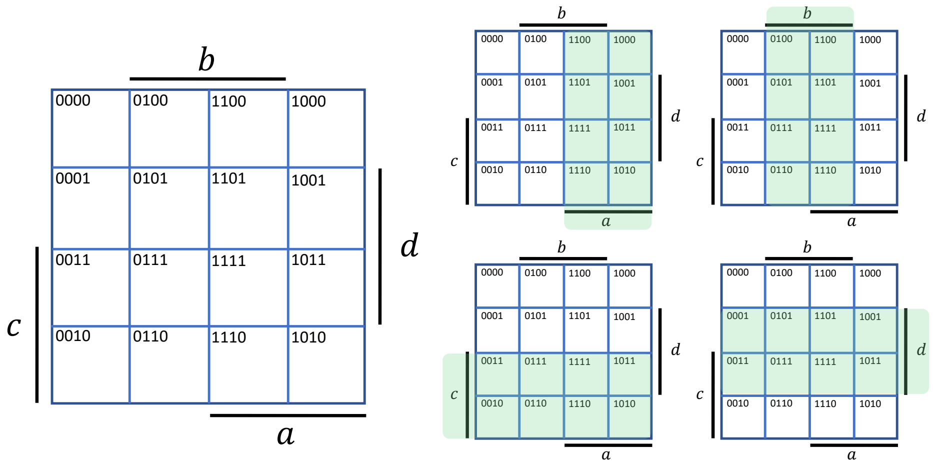


Figure 2.5: Format of the 4-variable Kmap and the regions in which its variables are 1.

2.4 Boolean Expressions

A function of n variables has 2^n different input values and hence will require a truth table with 2^n rows or a Kmap with 2^n cells. This quickly becomes impractical as n increases. Instead of

[2] This is the column-major format. The row-major is also widely used.

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

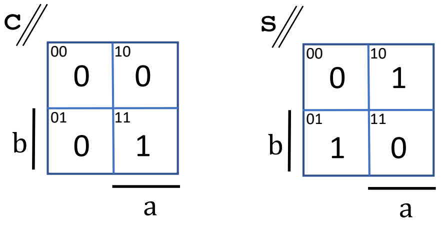


Figure 2.6: The 2-variable Kmaps for our `HA` circuit.

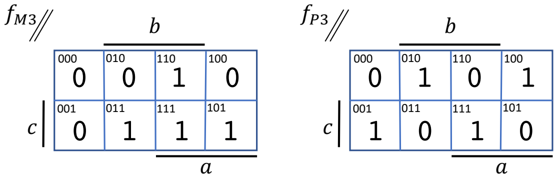


Figure 2.7: The Kmaps of the 3-variable majority and parity functions.

listing every output value, a boolean expression can be used to represent a boolean function. A boolean expression is formed from 0, 1, and variables using our three boolean operations, NOT, AND, and OR. If x and y are two boolean expressions, then we can form the following additional expressions:

$\overline{x}$	is the *complement* of x	(NOT),
$x * y$	is the *conjunction* of x and y	(AND),
$x + y$	is the *disjunction* of x and y	(OR).

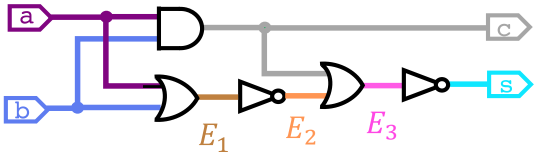
Given values for its variables, we can evaluate a boolean expression using our NOT, AND, and OR operations. The AND operation has precedence over the OR operation, and parentheses should be used when a different order is intended. For example, the order of operations in $a + b * c$ is the same as in $a + (b * c)$ but different than in $(a + b) * c$.

A boolean expression E represents a boolean function f if E's variables are inputs to f and evaluating E results in the same mapping of input values to 0 or 1 as f. There are many possible expressions for a boolean function. Finding a "good" expression is often a part of optimizing logic designs.

We can obtain boolean expressions for the gate outputs of a combinational circuit by using the labels of the external inputs as variables and assigning each gate output the expression corresponding to the gate's operation on the expressions obtained for its inputs. On the right in Figure 2.8, the expressions for each gate output are shown. The expression for output `c` is already using only the external input variables `a` and `b`.

$$c = a * b$$

To obtain an expression for `s` that uses only the external inputs, we can substitute for each inter-



$$\begin{aligned} c &= a * b \\ E_1 &= a + b \\ E_2 &= \overline{E_1} \\ E_3 &= c + E_2 \\ s &= \overline{E_3} \end{aligned}$$

Figure 2.8: Extracting boolean expressions from the HA logic diagram.

mediate expression as follows.

$$\begin{aligned} s &= \overline{E_3} \\ &= \overline{(c + E_2)} \\ &= \overline{(c + \overline{E_1})} \\ &= \overline{((a * b) + \overline{E_1})} \\ &= \overline{((a * b) + \overline{(a + b)})} \end{aligned}$$

Note that we have inserted parentheses when substituting a variable with an expression. This avoids any the possibility that we might inadvertently change the intended order of operations. The correct expression for the diagram in Figure 2.9(a) is $x * (y + z)$. Without the parentheses, we would obtain $x * y + z$ which is instead the correct expression for the diagram in Figure 2.9(b).

The two expressions $x * y + z$ and $x * (y + z)$ are not equivalent: they differ for some variable values. (Check $x = 0$, $y = 1$, and $z = 1$.). They represent different boolean functions.

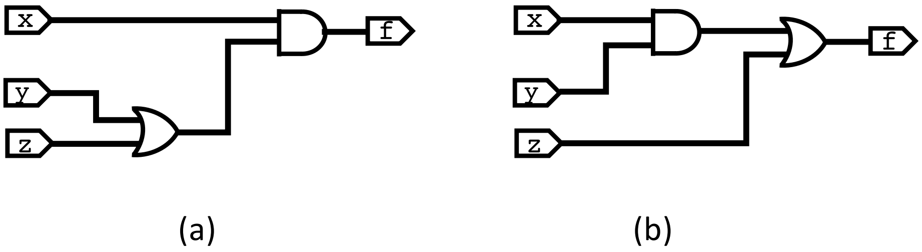


Figure 2.9: Parentheses may be required in boolean expressions obtained from logic diagrams.

2.5 Boolean Algebra

Two boolean expressions are equivalent if they represent the same boolean function. This merely means that they always agree when they are evaluated with the same input values. Checking all input values is impractical to determine equivalence. Instead, we can manipulate boolean expressions algebraically using the laws of boolean algebra. Many of these laws are familiar since they are learned in early mathematics courses, but a few will be unfamiliar, even seeming incorrect, since they do not hold in algebras familiar to you. Don't forget to use the unfamiliar laws.

A *boolean algebra* consists of

a set of elements B with designated identity elements 0 and 1,
the negation operation $\overline{a}$ for any element a in B,
the operation $a * b$ for any elements a, b in B, and
the operation $a + b$ for any elements a, b in B

satisfying the following laws for all elements a, b, c in B:

Commutative Laws
$a + b = b + a$
$a * b = b * a$

Distributive Laws
$a * (b + c) = a * b + a * c$
$a + (b * c) = (a + b) * (a + c)$

Complement Laws
$a + \overline{a} = 1$
$a * \overline{a} = 0$

Identity Laws
$0 + b = b$
$1 * b = b$

Associative Laws
$a + (b + c) = (a + b) + c$
$a * (b * c) = (a * b) * c$

Huntington showed that associativity can be derived from the other laws and so can be omitted from the definition [3]. However, it is typically included in the definition of boolean algebras.

The set $B = \{0, 1\}$ with our NOT, AND, OR operations qualifies as a boolean algebra. Essentially the same, but with different notation, propositional logic with $B = \{FALSE, TRUE\}$ and the operations ¬, ∧, and ∨ is also a boolean algebra. But these are not the only boolean algebras. Boolean algebras can have more than two elements, even an infinite number. Another boolean algebra you may have encountered is the algebra of sets with the complement, intersection, and union operations where the empty set and universal set serve as the identity elements. Bit vectors can also be the set of elements of a boolean algebra.

In addition to the required laws, all boolean algebras satisfy the identities below that can be derived from the laws above.

Idempotent
$a + a = a$
$a * a = a$

Domination
$1 + b = 1$
$0 * b = 0$

Absorption
$a + a * b = a$
$a * (a + b) = a$

Simplification
$a * (\overline{a} + b) = a * b$
$a + (\overline{a} * b) = a + b$

Involution
$\overline{\overline{a}} = a$

Uniqueness of complements
if $a + b = 1$ and $a * b = 0$ then $b = \overline{a}$

DeMorgan's Laws

$$\overline{(a+b)} = \overline{a} * \overline{b}$$
$$\overline{(a*b)} = \overline{a} + \overline{b}$$

These additional laws can be shown using only the laws that define boolean algebras. Their proofs can be found in Appendix C.

As in familiar algebras, the $*$ symbol is often omitted for clarity and size: $ab = a*b$. Using boolean algebra we can manipulate the expressions obtained from logic circuits. The expression for the `s` output of our `HA` can be simplified as follows:

$$\begin{aligned}
s &= \overline{((ab) + \overline{(a+b)})} \\
&= \overline{(ab)} * \overline{(\overline{(a+b)})} \\
&= (\overline{a} + \overline{b})(a+b) \\
&= \overline{a}(a+b) + \overline{b}(a+b) \\
&= (\overline{a}a + \overline{a}b) + (\overline{b}a + \overline{b}b) \\
&= (0 + \overline{a}b) + (\overline{b}a + 0) \\
&= \overline{a}b + a\overline{b} \\
&= f_{P2}(a,b) \\
&= a \oplus b
\end{aligned}$$

The $\oplus$ operation (XOR) corresponds to the parity function on 2 variables and the XOR gate. The XOR operation is commutative and associative, though generally complicated expressions with $\oplus$ are best resolved by replacing $x \oplus y$ by either $\overline{x}y + x\overline{y}$ or $(x+y)(\overline{x}+\overline{y})$.

Although many of the laws of Boolean algebra are familiar, some can easily be overlooked since they are new and seem unusual. The second Distributive Law is often forgotten. In simplifying the expression

$$(a+b+c+de) * (a+b+c+\overline{d}+\overline{e})$$

the first inclination might be to distribute the "$*$" (product) operation over the "$+$" (sum) obtaining 20 product terms.[3] Instead in boolean algebra, the common elements in the two sum terms can be factored out:

$$\begin{aligned}
(a+b+c+de) * (a+b+c+\overline{d}+\overline{e}) &= ((a+b+c)+de) * ((a+b+c)+\overline{d}+\overline{e}) \\
&= (a+b+c) + (de) * (\overline{d}+\overline{e}) \\
&= (a+b+c) + (de) * \overline{(de)} \\
&= (a+b+c) + 0 \\
&= a+b+c
\end{aligned}$$

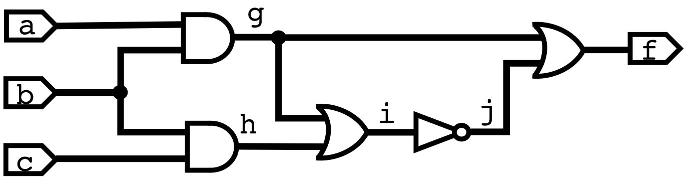
[3] This could eventually lead to the same expression, however with much more work and so greater risk of error.

When applying DeMorgan's Law it may be necessary to introduce parentheses to preserve the order of operations:

$$\begin{aligned} a\overline{bc} &= a * \overline{(bc)} \\ &= a * (\overline{b} + \overline{c}) \\ &= a * \overline{b} + a * \overline{c} \\ &\neq a * \overline{b} + \overline{c} \end{aligned}$$

2.6 Logic Diagrams and Boolean Expressions

We can obtain a boolean expression for the output of a combinational circuit from a logic diagram by labeling and obtaining an expression for each gate output in terms of its inputs. In Figure 2.10 we have added labels to the gate outputs that are internal net sources and obtained the expressions for each gate output on the right. The expression for the external output `f` in terms of the external inputs is obtained by substituting for the internal gate outputs.



$$\begin{aligned} \mathtt{g} &= \mathtt{a} * \mathtt{b} \\ \mathtt{h} &= \mathtt{b} * \mathtt{c} \\ \mathtt{i} &= \mathtt{g} + \mathtt{h} \\ \mathtt{j} &= \overline{\mathtt{i}} \\ \mathtt{f} &= \mathtt{g} + \mathtt{j} \end{aligned}$$

Figure 2.10: Obtaining a boolean expression for the output of a logic diagram.

$$\begin{aligned} f &= g + j \\ &= ab + j \\ &= ab + \overline{i} \\ &= ab + \overline{g + h} \\ &= ab + \overline{ab + h} \\ &= ab + \overline{ab + bc} \end{aligned}$$

Using boolean algebra we obtain a simpler expression for the output `f`.

$$\begin{aligned} f &= ab + \overline{ab + bc} \\ &= ab + \overline{ab} * \overline{bc} \\ &= ab + (\overline{a} + \overline{b}) * (\overline{b} + \overline{c}) \\ &= ab + (\overline{b} + \overline{a}) * (\overline{b} + \overline{c}) \\ &= ab + \overline{b} + (\overline{a} * \overline{c}) \end{aligned}$$

$$
\begin{aligned}
&= a + \overline{b} + (\overline{a} * \overline{c}) \\
&= a + \overline{a} * \overline{c} + \overline{b} \\
&= a + \overline{c} + \overline{b}
\end{aligned}
$$

This expression gives the following logic diagram.

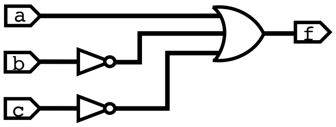


Figure 2.11: A simpler logic circuit.

A logic diagram can be represented by a single boolean expression as long as its gates each drive one pin. In Figure 2.10 the net `g` is connected to both OR gates. To represent this circuit, we need a boolean expression for `g` as well as one for `f`. The circuit in Figure 2.10 would correspond to the following boolean expressions:

$$
\begin{aligned}
g &= ab \\
f &= g + \overline{g + bc}
\end{aligned}
$$

2.7 Sum-of-Products, Minterms, and Canonical SOP

A single boolean function can be represented by different boolean expressions. There are specific formats for these expressions that are practical for building circuits and as input for tools that generate circuits. Sum-of-products (SOP) and product-of-sums (POS) are two such formats. These formats correspond to two-level logic circuits that can be optimized.

A *product term* is

- 1,
- a variable
- the complement of a variable, or
- the AND of several complemented or uncomplemented variables.

If the possible variables are a, b, c then the following are examples of product terms.

$$1,\ a,\ \overline{a},\ \overline{c},\ abc,\ a\overline{c},\ \overline{b}\overline{c}a$$

A *sum-of-products expression* is

- 0,

- a product term, or
- the OR of several product terms.

If the possible variables are $\{a, b, c\}$ the following are examples of sum-of-product expressions.

$$0,\ 1,\ a,\ a+\overline{b},\ \overline{a},\ \overline{c}+c,\ ab+c+\overline{c}\overline{b},\ abc+\overline{a}b\overline{c},\ ab+b\overline{c},\ \overline{b}\overline{c}a$$

Even when only SOP expressions are involved, there are still many possibilities. The following are all SOP expressions for the same boolean function.

$$a\overline{c}+b\overline{c}+\overline{a}c,\quad \overline{a}\overline{b}c+\overline{a}b+a\overline{c},\quad \overline{a}c+\overline{a}b\overline{c}+a\overline{c},\quad \overline{a}\overline{b}c+\overline{a}b\overline{c}+\overline{a}bc+a\overline{b}\overline{c}+ab\overline{c}$$

The fourth expression in the list above is in a format that is unique up to the ordering of terms and variables within terms. In each of its product terms, all possible variables appear, either complemented or uncomplemented.

A *minterm* for a set of variables is a product term in which each of the variables in the set appears exactly once, either complemented or uncomplemented.

A *canonical sum-of-products expression* for a boolean function f is a sum-of-products expression for f where each product term is a minterm for the variables of f.

A minterm will evaluate to 1 only when its complemented variables are 0 and its uncomplemented variables are 1. Since all variables appear in a minterm there is exactly one assignment of values for which the minterm evaluates to 1. Table 2.2 contains the eight possible value assignments for the variables a, b, c and the corresponding minterm that evaluates to 1 for that assignment. The notation m_n in Table 2.2 is "shorthand" for the minterm that is 1 for the binary vector that represents the value n. For example, $m_6 = ab\overline{c}$ evaluates to 1 for $a, b, c = 1, 1, 0$ and 110 is the binary representation of 6. **Changing the order of the variables affects this minterm notation!** In this text we will adopt the convention that the variable order is given by the order of the arguments to the function. For example, in the function $f(x, y, z, w, v) = \sum m(\ldots)$ the order of the variables is x, y, z, w, v. Care should be taken since this convention may not be in force elsewhere.

a	b	c	minterm	
0	0	0	$\overline{a}\overline{b}\overline{c}$	m_0
0	0	1	$\overline{a}\overline{b}c$	m_1
0	1	0	$\overline{a}b\overline{c}$	m_2
0	1	1	$\overline{a}bc$	m_3
1	0	0	$a\overline{b}\overline{c}$	m_4
1	0	1	$a\overline{b}c$	m_5
1	1	0	$ab\overline{c}$	m_6
1	1	1	abc	m_7

Table 2.2: The minterms for variables a, b, c and the m_n notation.

The minterms that appear in the canonical SOP expression of a function f correspond to the variable assignments for which f evaluates to 1. These assignments form the ONSET of f. There

a	b	c	f	$\bar{a}\bar{b}c$	$+$	$\bar{a}b\bar{c}$	$+$	$\bar{a}bc$	$+$	$a\bar{b}\bar{c}$	$+$	$ab\bar{c}$	minterm
0	0	0	0	0		0		0		0		0	
0	0	1	1	1		0		0		0		0	m_1
0	1	0	1	0		1		0		0		0	m_2
0	1	1	1	0		0		1		0		0	m_3
1	0	0	1	0		0		0		1		0	m_4
1	0	1	0	0		0		0		0		0	
1	1	0	1	0		0		0		0		1	m_6
1	1	1	0	0		0		0		0		0	

Table 2.3: The minterms corresponding to the ONSET for f.

is only one canonical sum-of-products for a function f since the minterms are fixed by the ONSET. Although this expression is unique, it is often cumbersome. A shorter notation for this expression is the "sum-of-minterms" form $f(\textit{variables}) = \sum m(\text{ONSET})$. The function from Table 2.3 would be written as:

$$f(a,b,c) = \sum m(1,2,3,4,6)$$

Here we have specified the order of the variables in the minterm notation by listing the arguments to f. From the sum-of-minterms format it is simple to fill in a K-map: we merely need to put 1's in the cells corresponding to the ONSET of the function as in Figure 2.12. In the remaining cells, the function will be 0. Here, each cell is labeled with the value corresponding to its binary vector, rather than the binary vector itself.

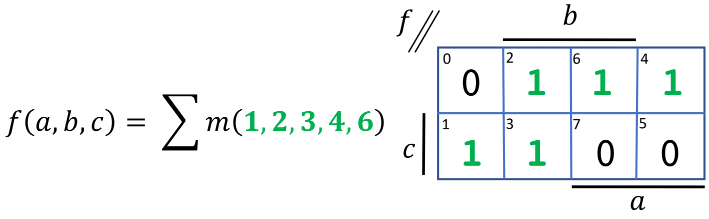


Figure 2.12: Using the ONSET of a function to obtain its K-map.

The sum-of-minterms and canonical sum-of-products expressions for our majority and parity function examples are

$$\begin{aligned} f_{M3}(a,b,c) &= \sum m(3,5,6,7) \\ &= \bar{a}bc + a\bar{b}c + ab\bar{c} + abc \end{aligned}$$

$$\begin{aligned} f_{P3}(a,b,c) &= \sum m(1,2,4,7) \\ &= \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c} + abc \end{aligned}$$

One "take-away" from this section is that any boolean function can be realized by a combination of NOT, AND, OR operations. For this reason, we say that together these three operators form a *universal set of operators.*

2.8 Product-of-Sums, Maxterms, and Canonical POS

The product-of-sums format is similar in structure to the sum-of-products format but with the roles of the AND and OR operations swapped. However swapping the two operations in an expression will not, in general, provide an equivalent expression!!

A *sum term* is

- 0,
- a variable
- the complement of variable, or
- the OR of several complemented or uncomplemented variables.

If the possible variables are a, b, c then the following are examples of sum terms.

$$0,\ \ a,\ \ \overline{a},\ \ \overline{c},\ \ a+b+c,\ \ a+\overline{c},\ \ \overline{b}+\overline{c}+a$$

A *product-of-sums expression* is

- 1,
- a sum term, or
- the AND of several sum terms.

If the possible variables are a, b, c then the following are examples of product-of-sum expressions.

$$0,\ \ 1,\ \ a,\ \ a\overline{b},\ \ \overline{a},\ \ \overline{c}c,\ \ (a+b)c(\overline{c}+\overline{b}),\ \ (a+b+c)(\overline{a}+b+\overline{c}),\ \ (a+b)(b+\overline{c}),\ \ \overline{b}+\overline{c}$$

A *maxterm* for a set of variables is a sum term in which every variable appears exactly once, either complemented or uncomplemented.

A *canonical product-of-sums expression* for a boolean function f is a product-of-sums expression where each sum term is a maxterm for the variables of f.

A maxterm will evaluate to 0 only when its complemented variables are 1 and its uncomplemented variables are 0. Since all variables appear in a maxterm there is exactly one assignment of values to the variables for which the maxterm will be 0. The table below lists the eight possible assignment values for the variables a, b, c and the corresponding maxterm that will be 0 for that assignment. The notation M_n in Table 2.4 is "shorthand" for the maxterm that evaluates to 0 for the binary vector that represents the value n. For example, $M_5 = \overline{a} + b + \overline{c}$ evaluates to 0 for $a, b, c = 1, 0, 1$ and 101 is the binary representation of 5. As before the order of the variables affects this notation and we adopt the convention that the variable order is given by the order of the arguments to the function. For example, in the function $f(x, y, z, w, v) = \sum m(\ldots)$ the order of the variables is x, y, z, w, v. Care should be taken since this convention may not be in force elsewhere.

The maxterms that appear in the canonical POS expression of a function f correspond to the variable assignments for which f evaluates to 0. These assignments form the OFFSET of f.

a	b	c	maxterm	
0	0	0	$a+b+c$	M_0
0	0	1	$a+b+\bar{c}$	M_1
0	1	0	$a+\bar{b}+c$	M_2
0	1	1	$a+\bar{b}+\bar{c}$	M_3
1	0	0	$\bar{a}+b+c$	M_4
1	0	1	$\bar{a}+b+\bar{c}$	M_5
1	1	0	$\bar{a}+\bar{b}+c$	M_6
1	1	1	$\bar{a}+\bar{b}+\bar{c}$	M_7

Table 2.4: The maxterms for variables a, b, c and the M_n notation.

a	b	c	f	$(a+b+c)$	*	$(\bar{a}+b+\bar{c})$	*	$(\bar{a}+\bar{b}+\bar{c})$	maxterm
0	0	0	0	0		1		1	M_0
0	0	1	1	1		1		1	
0	1	0	1	1		1		1	
0	1	1	1	1		1		1	
1	0	0	1	1		1		1	
1	0	1	0	1		0		1	M_5
1	1	0	1	1		1		1	
1	1	1	0	1		1		0	M_7

Table 2.5: The maxterms corresponding to the OFFSET for f.

There is only one canonical product-of-sums expression for a function f since the maxterms are fixed by the OFFSET (assuming the ordering of terms and variables within terms). Although this expression is unique, it is also often cumbersome. A shorthand for it the "product-of-maxterms" notation $f(variables) = \prod M(\text{OFFSET})$. The function from Table 2.5 would be written as:

$$f(a,b,c) = \prod M(0,5,7)$$

Similarly to the sum-of-minterms, it is simple to fill in a K-map from the product-of-maxterms format: we put 0's in the cells corresponding to the OFFSET and 1's in the remaining cells as in Figure 2.13. Again, here the cells are labeled with the value corresponding to their binary vector, rather than the binary vector itself.

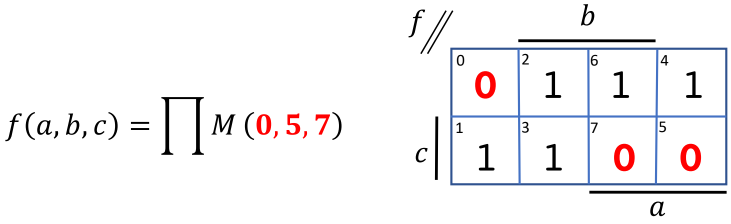


Figure 2.13: Using the OFFSET of a function to obtain its K-map.

The product-of-maxterm and canonical product-of-sums expressions for our majority and parity function examples are

$$\begin{aligned} f_{M3}(a,b,c) &= \prod M(0,1,2,4) \\ &= (a+b+c)(a+b+\overline{c})(a+\overline{b}+c)(\overline{a}+b+c) \end{aligned}$$

$$\begin{aligned} f_{P3}(a,b,c) &= \prod M(0,3,5,6) \\ &= (a+b+c)(a+\overline{b}+\overline{c})(\overline{a}+b+\overline{c})(\overline{a}+\overline{b}+c) \end{aligned}$$

2.9 Bubbles and DeMorgan's Laws

The NOT gate represents the complement operation in logic diagrams. But it can also appear as a "bubble" on an input or output pin. In Figure 2.14 the circuit from Figure 2.11 is shown on the left, and again on the right with the NOT gates replaced by bubbles on the input pins of the OR gate.

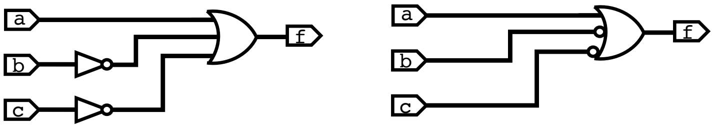


Figure 2.14: Using bubbles instead of NOT gates in a logic diagram.

In Figure 2.15 below the NOT gates and OR gates have been replaced with NOR gates. A NOR gate corresponds to the complementing result of the OR operation while a NAND gate corresponds to the complementing result of the AND operation. Depending on the technology we use to build the circuit, a NOR may be more efficient than an OR gate (e.g. CMOS).

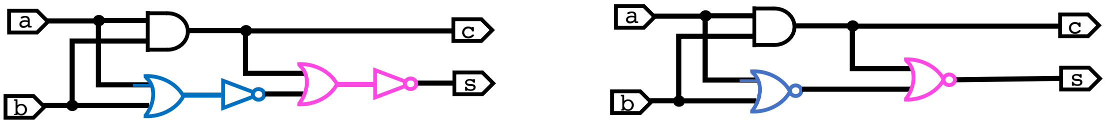


Figure 2.15: Using NOR gates in our HA circuit.

The Involution Law ($\overline{\overline{a}} = a$) and DeMorgan's Laws can be used to move the "bubbles" around in our logic diagrams. Figure 2.16 shows the equivalent circuits that these laws provide.

By using these laws we can transform a sum-of-products circuit into an equivalent circuit with only NAND gates. Figure 2.17 shows the three steps: (a) the original circuit, (b) introducing two bubbles on the connections between the AND gate outputs and the inputs to the OR gate, and

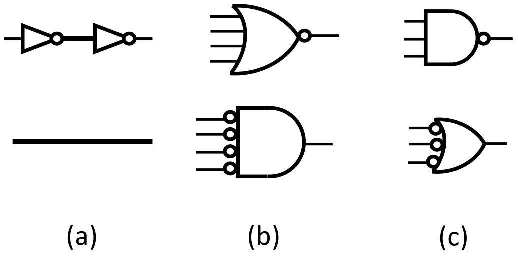


Figure 2.16: Equivalent circuits based on (a) Involution Law and (b and c) DeMorgan's Laws.

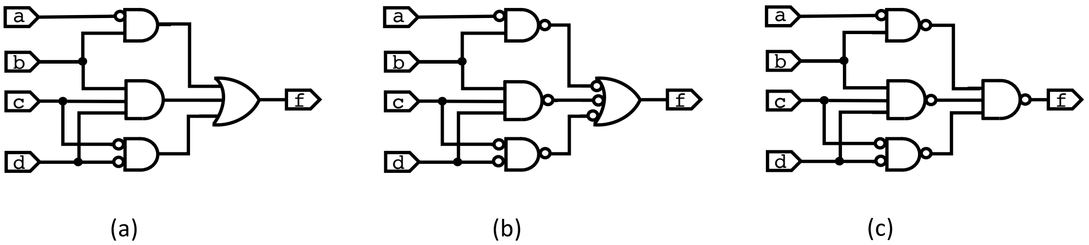


Figure 2.17: Converting an SOP circuit to an equivalent NAND-NAND circuit.

(c) using DeMorgan's Law on the OR gate. Since any boolean function has a sum-of-products expression that can be converted to a logic diagram with only NAND gates, the NAND operation, by itself, is a universal operator. This will also work for the NOR operation since we can transform a product-of-sums circuit into a circuit built only from NOR gates. Note that in Figure 2.18 (b) we introduced a NOT gate on the input that went directly to the AND gate. In (c) the NOT gate has been implemented with a NOR gate, although we are allowing negated inputs for other gates.

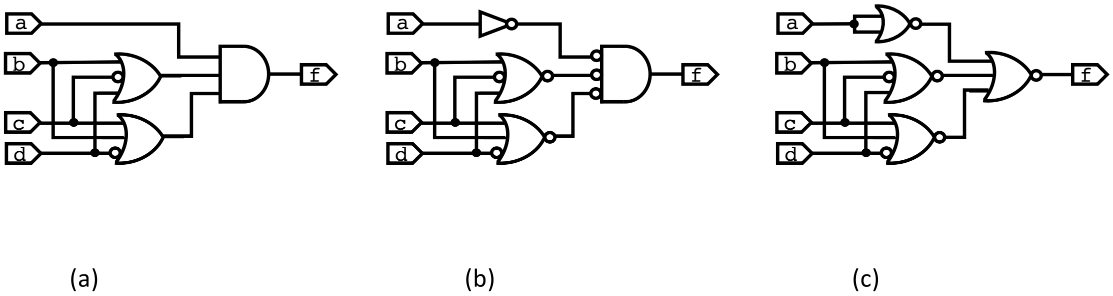


Figure 2.18: Converting a POS circuit to an equivalent NOR-NOR circuit.

2.10 Transient behavior of combinational circuits

So far in this chapter, we have been concerned with the "final" value of the outputs of combinational circuits. As discussed in Chapter 1 we are not considering any physical properties of our circuit and its environment that might result in a gate not providing its expected output value. However, we do expect that some amount of time is required for a gate's output value to change after its inputs change. And the outputs might not be consistent with the new input values until then. To build sequential logic, we will need to consider the amount of time required for our combinational circuits to react and settle when its inputs change value.

Consider the circuit in Figure 2.19. We can derive the following boolean expression for its output:

$$f = ab + \overline{a}c.$$

If b and c are both 1, then $f = a*1+\overline{a}*1 = a+\overline{a} = 1$ and the value of a does not affect f.

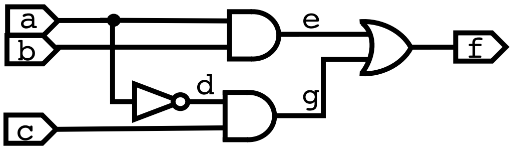


Figure 2.19: A combinational circuit with a hazard.

In the timing diagram in Figure 2.20, both inputs `b` and `c` are 1 and the input `a` changes from 1 to 0 at time 10.0. The NOT gate takes 5.0 time to react to this change, while the two AND gates take time 15.0 to react to their input changes.

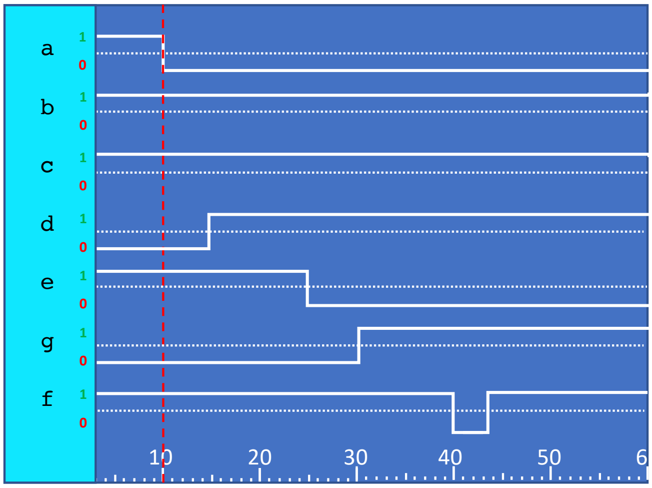


Figure 2.20: Timing diagram showing a hazard in a combinational circuit.

The OR gate sees its top input e change from 1 to 0 at time 25.0 and its other input g changes from 0 to 1 at time 30.0. Between time 25.0 and time 30.0, both inputs of the OR gate are 0. Determining whether, and for how long the output of the OR gate in Figure 2.19 will be 0 requires us to analyze this circuit in more detail than our gate model provides. Because there is a possibility that this circuit may temporarily provide an output that is inconsistent with both its previous and current input we say this circuit has a *hazard*. The hazard in the circuit in Figure 2.19 occurs because a change on the input a travels through multiple gates at different depths on parallel paths. However, hazards can also occur merely as the result of wiring delays. Although we may not know exactly when or if there will be a hazard, we can determine when an output will have settled to its final value.

Using *static timing analysis* we can obtain an upper bound on the amount of time needed for a combinational circuit to settle. In static timing analysis, we calculate when the output of a gate is "ready" by determining when all of its inputs have "arrived" and adding the time the gate will need for its output value to reflect these input values. The arrival time of an input is the time the source of its net is ready plus the delay associated with the connection between the source's output pin and the gate's input pin (wiring delay). In a combinational circuit, we can order the calculation of the gates' ready times so that the the arrival times of a gate's inputs are already known when its ready time is calculated. In Figure 2.21, the ready time of gate g9 is being calculated based on the ready time of the gates providing its inputs, g1 and g4. The output of gate g9 will be ready

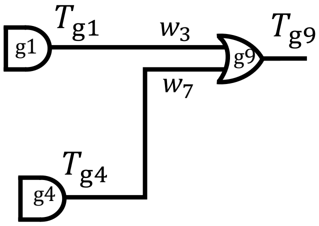


Figure 2.21: Static timing analysis calculation of the ready time of gate g9.

at time $T_{\texttt{g9}}$ where

$$T_{\texttt{g9}} = \max\{T_{\texttt{g1}} + D(w_3), T_{\texttt{g4}} + D(w_7)\} + D(\texttt{g9}).$$

The inputs to g9 are from g1 and g4. They are available at times $T_{\texttt{g1}}$ and $T_{\texttt{g4}}$. We add the delay of the connections w_3 and w_7 to these times and use the larger value as the time at which g9's input pins will have their final values. We then add the time required for g9's output to become consistent with these input values. This calculation begins with the times provided for the external inputs. It requires values for the delay of the connections (in blue) and the gate delays (in brown) as shown on the left in Figure 2.22. The results (in pink), are the ready times for the output of each gate as shown on the right in Figure 2.22. In some cases, a change in the input value of a gate might not cause the output value to change. For example, if the first input to arrive at g9 in Figure 2.21 gate has value 1, the value of the other input will not affect this OR gate's output value: the output will remain 1. Unfortunately timing analysis that takes into account logic values of the inputs/outputs (*dynamic timing analysis*) is computationally expensive and often not practical. Even more accurate timing estimates would be provided by modeling the physical properties of the

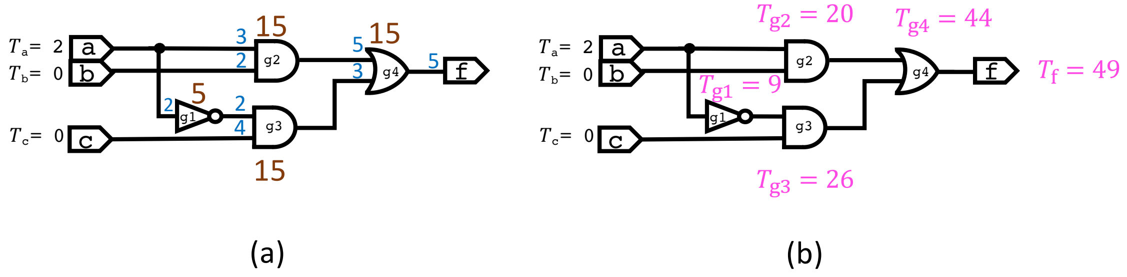


Figure 2.22: Static timing analysis example: (a) data, (b) result.

technology we are using. Static timing analysis provides a useful compromise between accuracy and efficiency.

2.11 Regular structures

Some logic functions can be implemented in combinational logic with regular structures. These may be available as components in a library. In this section, we will construct them from the basic gates (AND, OR, NOT) although they can often be implemented more efficiently in the technology used for the basic gates (e.g. transistors).

2.11.1 Decoders

A decoder with n inputs will have 2^n outputs. Each output is one of the 2^n minterms. For example, a 2-input decoder with inputs x_1 and x_0 will have four outputs (as in Figure 2.23(a)) corresponding to the minterms $m_0 = \overline{x}_1\,\overline{x}_0$. $m_1 = \overline{x}_1\,x_0$. $m_2 = x_1\,\overline{x}_0$. and $m_3 = x_1\,x_0$. This 2-input decoder can be implemented using four 2-input AND gates as shown in Figure 2.23(b). Generating all 2^n minterms separately will require 2^n n-input AND gates. Each n-input AND gate can be implemented with $n-1$ 2-input AND gates. A more efficient approach relies on *incident decoding*

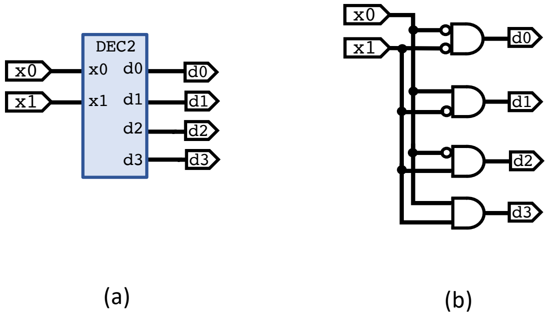


Figure 2.23: 2-input decoder: (a) symbol, (b) logic circuit.

where the inputs are divided into two groups, decoded separately, and then AND gates combine pairs of outputs from each decoder. Figure 2.24 illustrates incident decoding for a 4-input decoder. The inputs `x3`, `x2`, `x1`, and `x0` are divided into the lower bits, `x1` and `x0`, and upper bits `x3` and `x2`. As an example, consider the output `d13` which should be 1 when both `a3` and `b1` are 1 and this requires that `x3x2`=11 and `x1x0`=01. Since each 2-input decoder has four 2-input gates, this circuit requires a total of twenty-four 2-input gates versus the forty-eight 2-input gates needed to generate all 16 minterms separately. There is nothing special about 2 here.[4] The inputs could be divided into k groups of n/k inputs and then the n/k-decoders outputs would be combined using 2^n k-input AND gates. The choice of k will depend on the parameters of the technology used to implement the decoder.

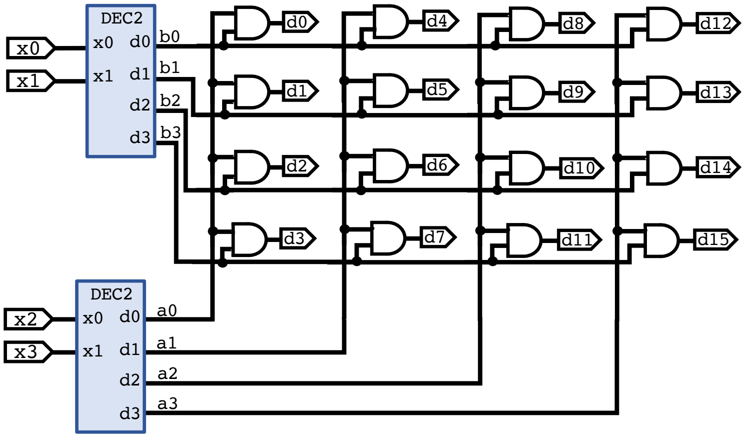


Figure 2.24: Co-incident decoding in a 4-input decoder.

2.11.2 Multiplexers

A *multiplexer* allows multiple sources to send their values to the same place, though not simultaneously. Additional inputs select which source is currently the output of the multiplexer. For example, a 2-input multiplexer with source inputs i_0 and i_1 would have one selector s and its output would be

$$o = \begin{cases} i_0 & \text{if } s = 0 \\ i_1 & \text{if } s = 1 \end{cases}$$

We could think of this mechanically as a two-pole switch, but in logic design, a multiplexer's output need only have the same logic value as the source. The output doesn't need to be physically connected to a source. The value of the output of a 2-input multiplexer is given by the boolean equation:

$$o = \overline{s} \cdot i_0 + s \cdot i_1.$$

The symbol for a 2-to-1 multiplexer is shown in Figure 2.25(a) and its logic circuit is in Figure 2.25(b). The symbols for the 2-input, 4-input, 8-input, and 2^n-input multiplexers are

[4] A surprising thing to say in digital design.

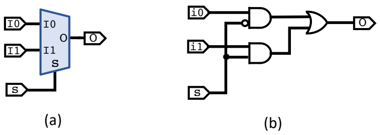


Figure 2.25: Two input multiplexer: (a) 2-to-1 multiplexer symbol, (b) 2-to-1 multiplexer logic circuit.

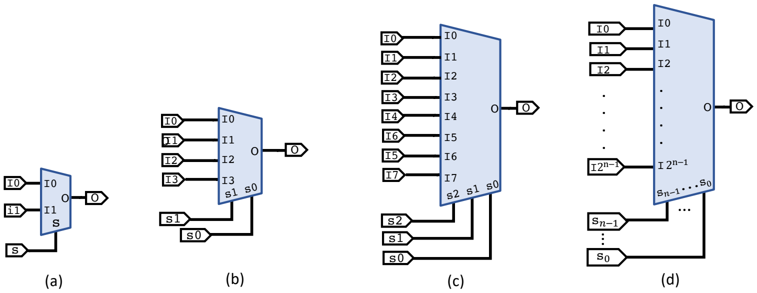


Figure 2.26: Larger multiplexers.

shown in Figure 2.26. The number of selectors grows as the base 2 logarithm of the number of inputs. With these larger multiplexers, the role of the multiplexer appears more akin to indexing in arrays. Multiplexers can be implemented as shown in Figure 2.27(a) where a 2^n-input multiplexer

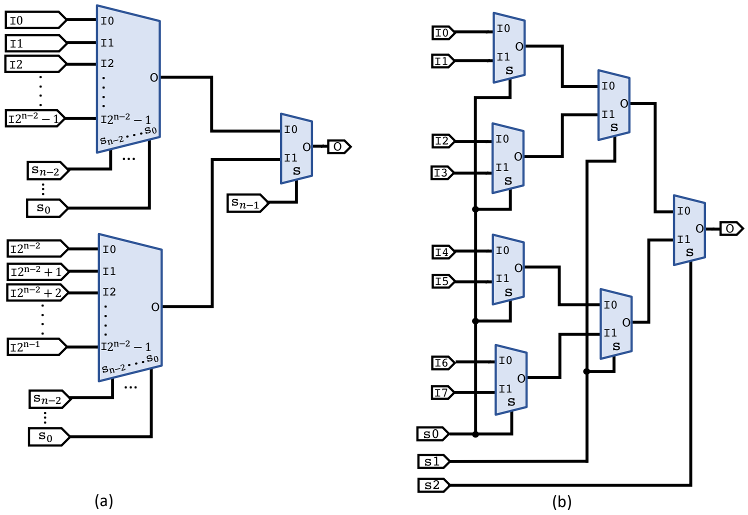


Figure 2.27: Structure of larger muxes (a) decomposition, (b) 8-input multiplexer.

has been decomposed into two 2^{n-1}-input multiplexers and the final output is determined by the most significant selector, s_{n-1}. Repeatedly applying this decomposition for an 8-input multiplexer results in the circuit in Figure 2.27(b).

2.11.3 Comparators

A common task is the need to compare two integer values to determine whether they are the same, or if one is greater than the other. Suppose we are given two n-bit unsigned integers, $A = a_{n-1} \dots a_1 a_0$ and $B = b_{n-1} \dots b_1 b_0$. For $n = 1$, the XNOR gate as in Figure 2.28(a) will tell us if A and B are the same, while $A > B$ occurs only when $a_0 = 1$ and $b_0 = 0$ and this can be recognized by an AND gate with an inverted input as shown in Figure 2.28(b).

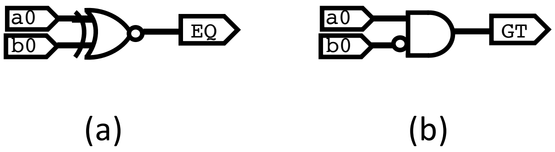


Figure 2.28: Comparison of single-bit integers: (a) equality, (b) greater-than.

This circuit is shown in Figure 2.29. Two n-bit integers (or any two n-bit vectors) will be the same if they agree at every bit position. We can apply an XNOR gate to the pair of bits at each position and then use n-input AND to determine if all the pairs agree.

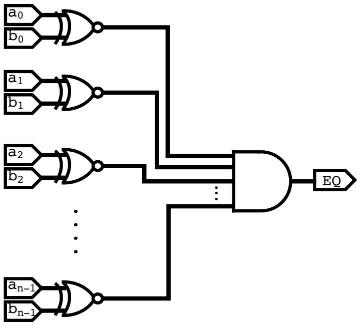
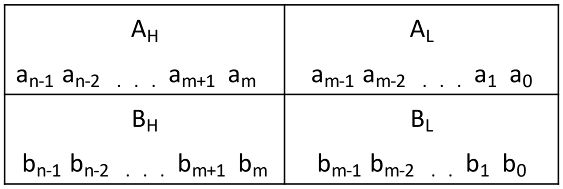


Figure 2.29: Comparison of n-bit vectors to determine equality.

To determine whether $A > B$ is more complicated since the bit positions have different significance. In Figure 2.30 the n-bit unsigned integers A and B have been split into smaller integers of size m and $n - m$. The bit vectors A_H and B_H are the high order bits (the leftmost $n - m$ bits, while A_L and B_L are the low order bits (the rightmost m bits).

In checking whether $A > B$ there are two possibilities. If $A_H > B_H$ then we know that $A > B$ without considering the low-order vectors. But if $A_H = B_H$ then $A > B$ depends on whether $A_L > B_L$. (If $A_H < B_H$ we know $A < B$.). So $A > B$ is equivalent to $A_H > B_H$, or $A_H = B_H$ and $A_L > B_L$. Figure 2.31 represents the structure of an n-bit comparison achieved in this manner. The symbol on the left in Figure 2.31(a) provides both the EQ and GT outputs. In Figure 2.31(b) the comparison of A and B is obtained from the EQ and GT outputs comparing A_H with B_H, and A_L with B_L.



A_H	A_L
$a_{n-1}\ a_{n-2}\ \ .\ .\ .\ \ a_{m+1}\ \ a_m$	$a_{m-1}\ a_{m-2}\ \ .\ .\ .\ \ a_1\ \ a_0$
B_H	B_L
$b_{n-1}\ b_{n-2}\ \ .\ .\ .\ \ b_{m+1}\ \ b_m$	$b_{m-1}\ b_{m-2}\ \ .\ .\ \ b_1\ \ b_0$

Figure 2.30: Splitting the n-bit unsigned integers into A and B into the high and low order parts.

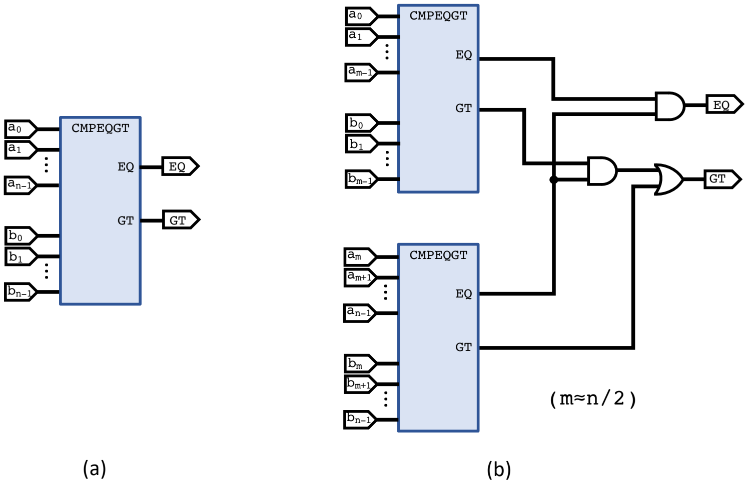


Figure 2.31: Comparison of single-bit integers: (a) equality, (b) greater-than.

For signed integers, the leading bit (the sign bit) must be considered. In 2's complement, $A > B$ for signed integers A and B, if A is non-negative and B is negative, or if they have the same sign and $A > B$ when considered as unsigned integers.

2.12 Pre-fabricated components

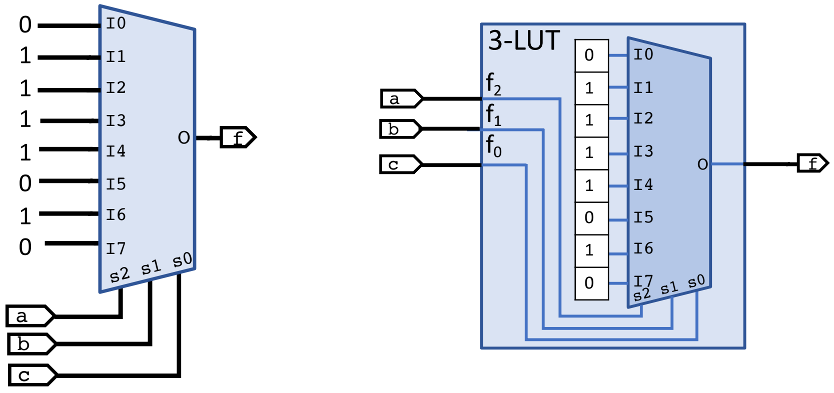
To facilitate and accelerate the assembly of logic circuits, prefabricated components provide structures that can be conveniently configured to become the desired logic circuit. Two common structures used to implement combinational logic are look-up tables and arrays of gates with programmable connections.

2.12.1 Look-Up Tables

In a *Look-Up Table* with n inputs (an n-LUT) the output is produced by using the n-inputs as selectors to "look-up" the value in a truth table. Recall that the role of a multiplexer is similar to indexing in an array, and so LUTs are in essence 2^n-input multiplexers where the 2^n inputs (sources) are the output values from the truth table for the desired function. In Figure 2.32 The truth table of the function $f(a,b,c) = \sum m(1,2,3,4,6)$ is shown in Figure 2.32(a). This function is implemented in Figure 2.32(b) by setting the inputs of an 8-input multiplexer to 0 or 1 according to the function's output in the truth table. Figure 2.32(c) shows a 3-LUT implementing this same function. The 3-LUT must be "configured" with the values from the truth table. In a reconfigurable logic device, these values are held in a shift register and are shifted in during the configuration of the device with a bitstream.

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(a)



(b) (c)

Figure 2.32: (a) Truth table of $f(a, b, c) = \sum m(1, 2, 3, 4, 6)$, (b) implementation of $f(a, b, c)$ with an 8-input multiplexer, (c) 3-LUT implementing $f(a, b, c)$.

An n-LUT can implement any boolean function of n variables or less, but functions of more than n variables must be decomposed into functions with fewer variables. Designers use software to enter their desired circuit and program the device. This software decomposes logic into the appropriate size LUTs and maps the gates and their interconnections to locations on the device.

2.12.2 Programmable Logic Devices

There are several different types of Programmable Logic Devices (PLDs). In these devices, the gates are already present. The desired circuit is obtained by programming the connections to the gate input pins. Figure 2.33 contains a representation of a small Programmable Logic Array.[5] This PLA has an *AND-plane* where the external inputs or their complements can be connected to the AND gate inputs. In *OR-plane* the product terms from the AND gates can be selected and possibly inverted. In the AND-plane and OR-plane, two wires that overlap are connected by programming the switches. The outputs of the OR gates can be inverted by connecting the first input of the XOR gate to either 0 or 1 ($0 \oplus x = x$ and $1 \oplus x = \overline{x}$). In Figure 2.33 the green switches are closed (connect the vertical/horizontal wires) while the red switches are open.

The PLA in Figure 2.33 is programmed to implement the functions $f(a, b, c) = \sum m(1, 2, 3, 4, 6)$ and $g(a, b, c) = \sum m(0, 2, 7)$. With only 4 AND gates available, and two functions that together have seven minterms, it might not appear feasible at first. But by obtaining an SOP expression for

[5] This is a representation of the logic structure within a PLA. The logic gates and their programmable inputs are realized by a network of transistors.

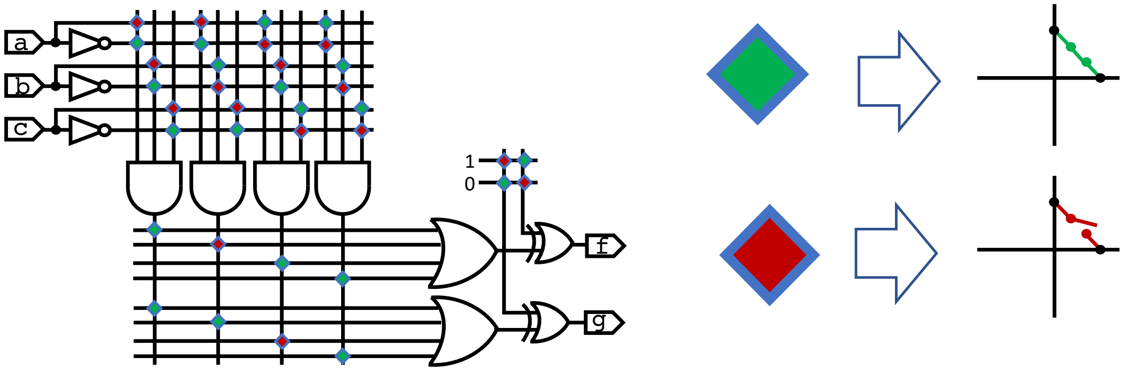


Figure 2.33: Programmable Logic Array organization. The vertical/horizontal wires are connected if the programmable switch at their intersection is green.

the complement of $f(a,b,c)$ as follows,

$$f(a,b,c) = \sum m(1,2,3,4,6) = \overline{\sum m(0,5,7)} = \overline{\bar{a}\,\bar{b}\,\bar{c} + a\,\bar{b}\,c + a\,b\,c}$$

we can implement both $f(a,b,c)$ and $g(a,b,c)$ from the minterms m_0, m_2, m_5, and m_7.

There are a variety of structures available as Programmable Logic Devices. Capacity, performance, and the configuration process are the main considerations in selecting a device to implement logic.

Chapter 3

Synthesis of Combinational Logic

3.1 Introduction

In Chapter 2 we presented several ways to represent, analyze, and manipulate combinational logic. In this chapter, we will discuss how to synthesize the best or better combinational logic. How this is approached will depend on the goal (cost versus performance) and also the technology used to implement the logic. But even without a specific technology in mind, obtaining small boolean expressions is useful as a starting point, or merely as a way to obtain a compact representation for logic. We will be considering this "technology-independent" logic minimization.

3.2 Literals

The number of literals in a boolean expression is one way to measure the "size" of our boolean expressions. It corresponds to the total number of 2-input gates in a logic diagram obtained from the expressions. This is considered a "technology independent" measure.

A *literal* in a boolean expression is an occurrence of a variable or its complement.

Every occurrence of a variable is a separate literal: $aaaaa$ has 5 literals and $\overline{a}a + bc$ has 4 literals. Table 3.1 has more examples. The number of literals in an expression corresponds to the number of 2-input gates required for that expression. The expressions abc and $ab+c$ both have three literals and will require two 2-input gates as shown in Figure 3.1. With the number of literals as the measure, there is no additional cost for the complementing variable. In some technologies, both

Expression	# of literals	Expression	# of literals
0	0	1	0
a	1	$\overline{c}$	1
$ab + \overline{c}$	3	$a + b + \overline{c}$	3
$\overline{b}\overline{(a + c)} + \overline{a\overline{c}}d$	6	$a\overline{a} + aaa$	5

Table 3.1: Examples for number of literals in an expression

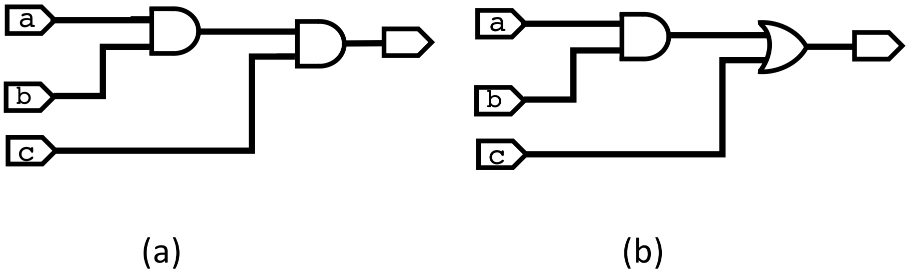


Figure 3.1: The number of 2-input gates for abc and $ab + c$ is the same.

the complemented and uncomplemented inputs are available, and in some technologies (CMOS) implementing an expression with a complemented input or output will be less costly than when it is uncomplemented.

3.3 Using Kmaps to obtain minimal SOP/POS expressions

Our goal is to obtain sum-of-products (SOP) and product-of-sums (POS) expressions with the minimum number of literals. We begin by considering the sum of products.

As an example, consider the majority-of-3 function from Section 2.2 and its expressions from Section 2.7.

$$\begin{aligned} f_{M3}(a,b,c) &= \sum m(3,5,6,7) \\ &= \overline{a}bc + a\overline{b}c + ab\overline{c} + abc \end{aligned}$$

Each product term in any sum-of-products expression for a function is considered an *implicant* of the function represented. Whenever such a product term evaluates to 1, its function will evaluate to 1. The input values for which an implicant evaluates to 1 are said to be *covered* by that implicant. Every element of the ONSET (the input values for which the function is 1) must be covered by at least one implicant. So in choosing product terms for our sum-of-products expression, we must meet two conditions:

1. Each product term must be an implicant of the function.
   (It should not cover any elements in the OFFSET.)
2. Each element of the ONSET must be covered by at least one implicant.

A sum-of-minterms expression meets these requirements since each minterm covers exactly 1 element of the ONSET. However, the sum-of-minterms is rarely the expression with the fewest literals. Consider the majority-of-3 function, f_{M3}. Its sum-of-minterms expression is

$$f_{M3}(a,b,c) = \overline{a}bc + a\overline{b}c + ab\overline{c} + abc$$

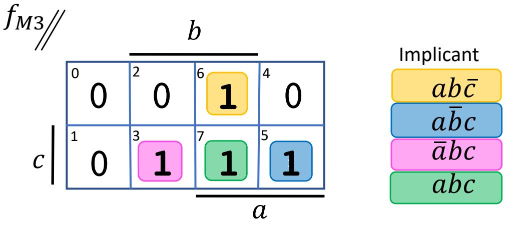
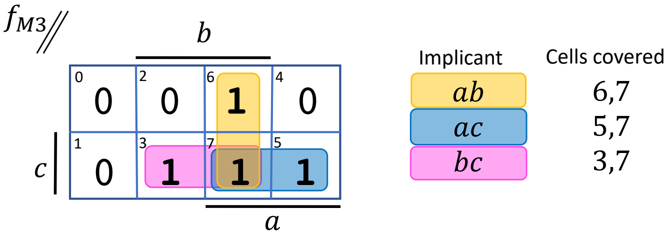


Figure 3.2: The cells covered by the four minterms for f_{M3}.

To have a majority of 1's among the three inputs, at least two of them must be 1. This observation leads to the following expression for f_{M3}

$$f_{M3}(a,b,c) \quad = \quad ab + ac + bc$$

This expression has 6 literals while the sum-of-minterms expression has 12. The implicants in the 6 literal expression are ab, ac, and bc. They each cover two elements of the ONSET. Since the ONSET has 4 elements, these implicants must overlap. In Figure 3.3 the cells covered by the three implicants are shown.



Implicant	Cells covered
ab	6,7
ac	5,7
bc	3,7

Figure 3.3: The cells covered by the three implicants of $ab + ac + bc$.

3.3.1 Finding implicants

The first step in finding a minimal sum-of-products for a boolean function f is to identify its implicants. For a small number of variables, this can be done by drawing rectangles in Kmaps.

Recall from Section 2.3 that for each variable, the Kmap can be divided into two: one half where the variable is 1 and in the other half it is 0. A product term will be 1 in the cells where its uncomplemented variables are 1, and its complemented variables are 0. For a variable not appearing in the product term, there will be equal numbers of cells where that variable is 1 versus 0. Two examples of implicants, $\overline{a}\overline{c}d$ and $b\overline{d}$, are shown in Figure 3.4 . The rectangle associated with $\overline{a}\overline{c}d$ is in the half where $a = 0$, in the half where $c = 0$, and in the half where $d = 1$. It overlaps both halves for the variable b. The rectangle associated with $b\overline{d}$ is in the half where $b = 1$ and in the half where $d = 0$. It overlaps both halves for the variables a and c.

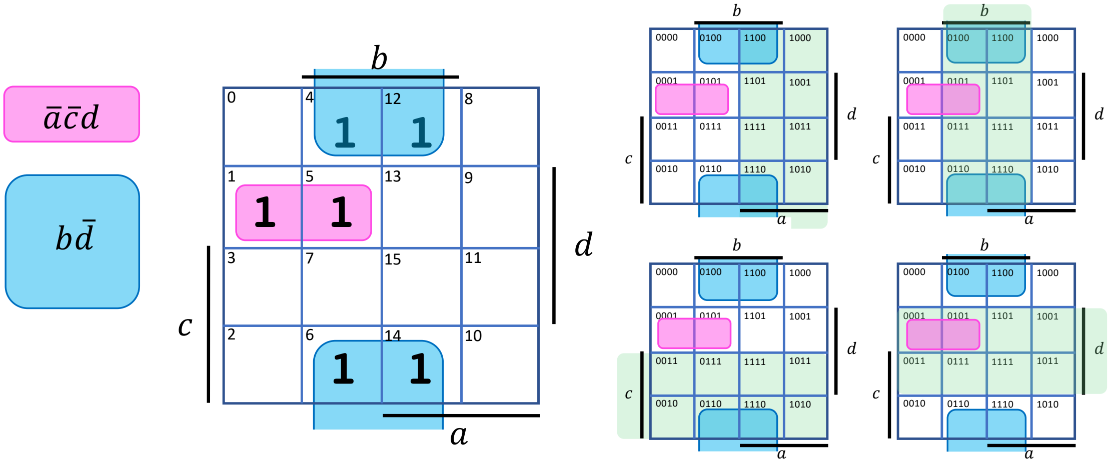


Figure 3.4: Two rectangles and their product terms.

The rectangles (aka cubes) associated with product terms will have dimensions that are powers of 2 (1×1, 1×2, 1×4, 2×2, 2×4, etc.). The product term associated with the smallest cube, a 1×1, is a minterm. When the size of a cube doubles, the number of variables in the associated product term goes down by 1. Since the goal is to have as few literals as possible, using a cube that fits inside a larger one is sub-optimal. A *prime implicant* of f is a product term that cannot lose any variable and remain an implicant. That is, the prime implicant is a product term that cannot be smaller. In terms of cubes, a prime implicant (aka PI) is a cube that is not entirely inside another cube; it cannot be larger. As an example, the PIs of the function $\sum m(0,1,3,4,5,7,9,10,11,13,14)$ are shown in Figure 3.5. Here we have indicated each PI on a separate Kmap for clarity. In practice, they will be drawn on one Kmap as in Figure 3.6. Note that P_4 and P_5 are entirely covered by other PIs, but neither can be expanded. P_4 and P_5 are prime implicants since they are not inside any other PI. The function in Figure 3.5 has 29 implicants, but only 6 of them are prime implicants. We can safely ignore the non-prime implicants since it would be sub-optimal to include them in our SOP expression. This greatly simplifies the next step: selecting implicants for the cover.

3.3.2 Forming the cover

Once we have identified the PIs we need to select enough of them to cover the cells in the ONSET. We could include all of them, but that may not be necessary. Our goal is to minimize the number of literals so a "smaller" cover is better. If a PI is the only PI that covers a particular cell then we cannot cover the ONSET without it: it must be included. Such a PI is said to be *forced* by that cell.[1]

In Figure 3.6 the five cells of the ONSET that have only one PI covering them are indicated by large boldface 1's. P_1 is forced by cell 7, P_2 is forced by cells 0 and 4, P_3 is forced by cell 13, and

[1] The term "essential" prime implicant is used in the literature rather than "forced" and can lead to confusion when the dictionary definition of essential is applied. It could be that a prime implicant that is not forced by any cell, will always be in a minimal SOP.

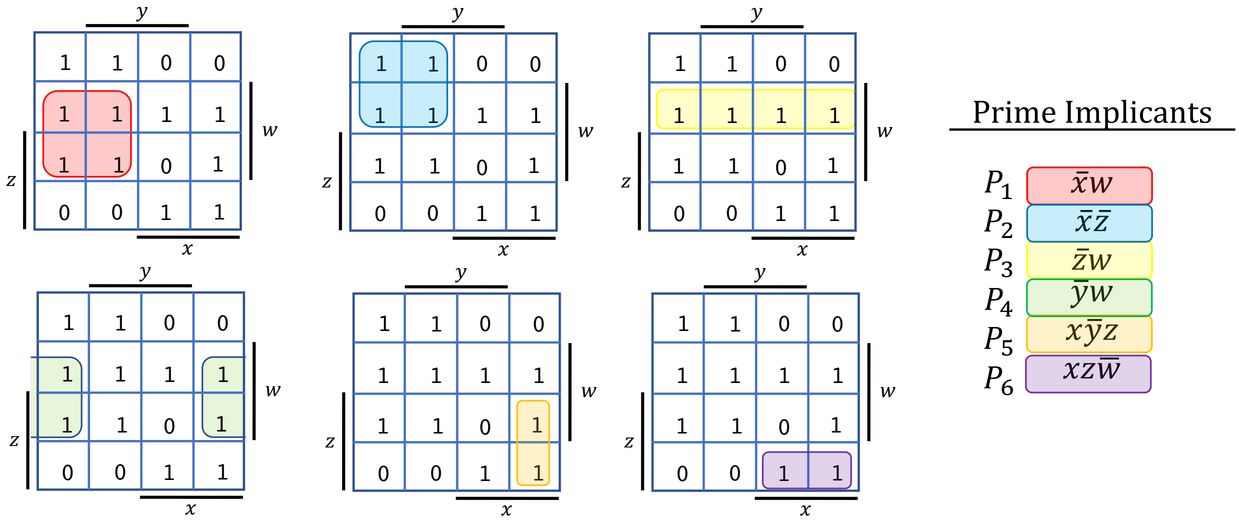


Figure 3.5: The prime implicants (PIs) of $\sum m(0, 1, 3, 4, 5, 7, 9, 10, 11, 13, 14)$.

P_6 is forced by cell 14. P_4 and P_5 are not forced. The remaining cells of the ONSET all have at least two PIs that cover them.

We must include P_1, P_2, P_3, and P_6 in the cover. Together these four PIs almost cover the ONSET; the only cell in the ONSET that they don't cover is 11 as shown in Figure 3.7. To cover cell 11, we can select either P_4 or P_5. The product term for P_4 has one less literal, so it should be selected instead of P_5. Our cover is then P_1, P_2, P_3, P_4, and P_6 and the minimum SOP expression is

$$\overline{x}w + \overline{x}\overline{z} + \overline{z}w + \overline{y}w + xz\overline{w}$$

3.3.3 Using Kmaps to obtain minimal SOP expressions

The procedure we have followed can be summarized in four steps.

1. Identify **PI**s. (It is tempting here to ignore a PI that appears unlikely to end up in the cover, but this will affect the next step.)
2. Identify the cells of the ONSET that force a PI.
3. Add the forced PIs to the cover and mark the cells they cover.
4. Select a "smallest" group of the remaining PIs to cover any remaining uncovered cells of the ONSET.

The fourth/final step in this procedure seems to have "punted the ball" so to speak, but for problems with at most 6 variables, the number of unforced PIs will likely be small and it will be simple to identify the best group to cover the remaining cells. A more complete and systematic method for step 4 can be found in Section 3.4. Beyond 6 variables using Kmaps becomes unmanageable and impractical, and the methods in Section 3.4 will be preferable.

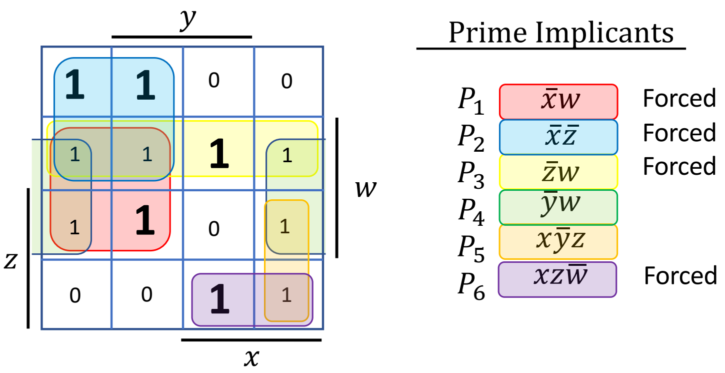


Figure 3.6: Identifying which PIs of $\sum m(0, 1, 3, 4, 5, 7, 9, 10, 11, 13, 14)$ are forced. Cells in the ONSET that are covered by only one PI are indicated by a large boldface **1**.

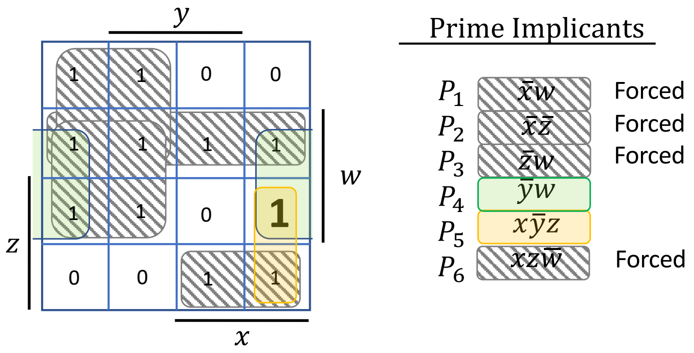


Figure 3.7: After selecting the forced PIs (striped), only cell 11 of the ONSET remains to be covered.

3.3.4 Using Kmaps to obtain minimal POS expressions

The method in Section 3.3.3 for obtaining a minimal SOP expression has a counterpart for POS expressions in which the OFFSET of the function is covered by *implicates* (sum terms). Instead of presenting a separate procedure with notation for sum terms, we will make use of the procedure for SOPs. To obtain a minimal POS expression for a function f we will first obtain the minimal SOP expression for $\overline{f}$, the complement of f. Then by complementing this SOP expression and applying DeMorgan's Laws, we will obtain a POS expression for f. The number of literals will not change when we apply DeMorgan's Laws, so the POS expression will also be minimal.

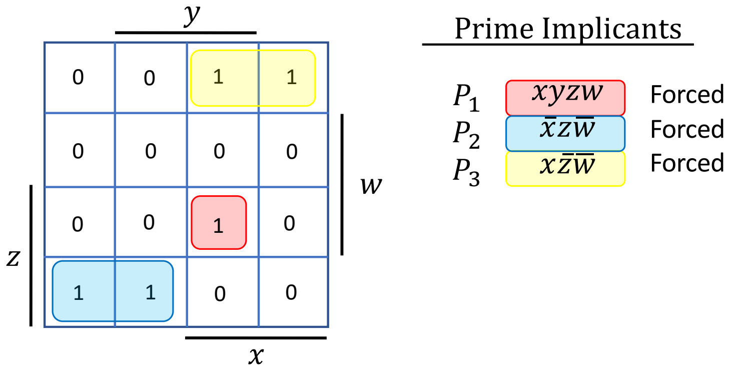
The Kmap for the complement of the function from Figure 3.5 is shown in Figure 3.8.

Since all of the PIs are forced, the minimal SOP expression is

$$\overline{f}(x, y, z, w) = x\,y\,z\,w + \overline{x}\,z\,\overline{w} + x\,\overline{z}\,\overline{w}.$$

We can then complement this expression to obtain a POS expression for $f(x, y, z, w)$.

$$f(x, y, z, w) \quad = \quad \overline{\overline{f}}(x, y, z, w)$$



0	0	1	1
0	0	0	0
0	0	1	0
1	1	0	0

P_1	$xyzw$	Forced
P_2	$\bar{x}z\bar{w}$	Forced
P_3	$x\bar{z}\bar{w}$	Forced

Figure 3.8: To find the POS for $f(x, y, z, w) = \sum m(0, 1, 3, 4, 5, 7, 9, 10, 11, 13, 14)$ the minimal SOP expression for $\overline{f}(x, y, z, w) = \sum m(2, 6, 8, 12, 15)$ is obtained.

$$\begin{aligned}
&= \overline{x\,y\,z\,w + \overline{x}\,z\,\overline{w} + x\,\overline{z}\,\overline{w}} \\
&= \overline{(x\,y\,z\,w)} * \overline{(\overline{x}\,z\,\overline{w})} * \overline{(x\,\overline{z}\,\overline{w})} \\
&= (\overline{x} + \overline{y} + \overline{z} + \overline{w})(\overline{\overline{x}} + \overline{z} + \overline{\overline{w}})(\overline{x} + \overline{\overline{z}} + \overline{\overline{w}}) \\
&= (\overline{x} + \overline{y} + \overline{z} + \overline{w})(x + \overline{z} + w)(\overline{x} + z + w)
\end{aligned}$$

For this particular function, the minimal SOP expression has 11 literals while the minimal POS expression has 10. For each function where the SOP expression has fewer literals, there is a function where the POS has fewer literals: (its complement).

The procedure to find a minimal POS expression for a function f can be summarized in three steps:

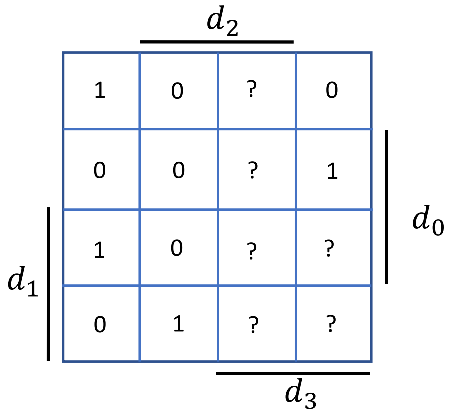
1. Obtain the minimal SOP expression for $\overline{f}$ using the procedure from Section 3.3.3.
2. Complement the expression obtained in Step 1.
3. Apply DeMorgan's Laws to transform the complemented expression into a POS.

3.3.5 Taking advantage of Don't Cares

A boolean function of n variables has 2^n possible input values. This may be more than we need, while 2^{n-1} is not enough. For example, if the input to our design represents a decimal digit (0-9), we will need a boolean function with 4 variables (d_3, d_2, d_1, d_0). This function will have six input values (representing 10-15) that are not used. For the input values that do not have a required output value, we say that the output value of the function is a *don't care*. A function with *don't care* inputs is said to be *incompletely specified*: some of the input values do not have a specific output value.[2]

We could randomly pick output values for 10-15, but a better approach is to decide on output values that help minimize our logic. As an example suppose we need a logic circuit that will recognize

[2] Technically to be a function, an output value for each possible input value must be provided. By inventing and using the *don't care* as a possible output value we have retained this property.



1	0	?	0
0	0	?	1
1	0	?	?
0	1	?	?

Figure 3.9: The Kmap of the incompletely specified function that recognizes multiples of 3 for a decimal digit.

when a decimal digit is a multiple of 3. There will be four inputs to this circuit and one output. The output should be 1 for the input values representing 0, 3, 6, and 9, and the output should be 0 for the input values representing 1, 2, 4, 5, 7, and 8. We leave the output for the inputs representing 10-15 as unspecified by using the "?" to represent a don't care. The Kmap for this function is shown in Figure 3.9. Our incompletely specified functions will have a DCSET (the set of don't care input values) in addition to their ONSET and OFFSET. Our function to detect multiples of 3 in a decimal digit can be written as

$$\begin{aligned} f(d_3, d_2, d_1, d_0) &= \sum m(0,3,6,9) + \mathrm{D}(10,11,12,13,14,15) \\ &= \prod M(1,2,4,5,7,8) + \mathrm{D}(10,11,12,13,14,15) \end{aligned}$$

To obtain a boolean expression for our circuit there are several possibilities. We could decide to set the output values of the DCSET all to 0, all to 1, or extend the function to recognize only 12 and 15 as multiples of 3. These three options correspond to the Kmaps in Figure 3.10(a), (b), and (c). They produce SOP expressions with 16, 16, and 20 literals. In Figure 3.10(d), the "?" cells are used to create larger prime implicants than in (a). Setting all the DCSET cells to 1 as in (b) would also result in these larger prime implicants, but the increased ONSET forces more PIs into the cover. The best option is to consider the DCSET cells as 1's to find PIs while treating them as 0's to form the cover. In Figure 3.10(d) the four PIs are all forced, so together they form the cover. The resulting SOP expression has 12 literals and results in the following Boolean function:

$$f_1(d_3, d_2, d_1, d_0) = \sum m(0,3,6,9,11,13,14,15) = \overline{d_3}\ \overline{d_2}\ \overline{d_1}\ \overline{d_0} + d_3\ d_0 + \overline{d_2}\ d_1\ d_0 + d_2\ d_1\ \overline{d_0}$$

The only change needed to our minimal SOP procedure is to consider elements of both the DCSET and ONSET in identifying implicants. Now a product term is an implicant if it evaluates to 1 only for elements that are in either the ONSET or DCSET. We will require prime implicants to cover at least one element of the ONSET; they cannot only consist of DCSET cells.

The DCSET should also be used for obtaining a minimal POS. Since the elements of the DCSET can either evaluate to 0 or 1, the minimal SOP and POS may not be equivalent expressions. The

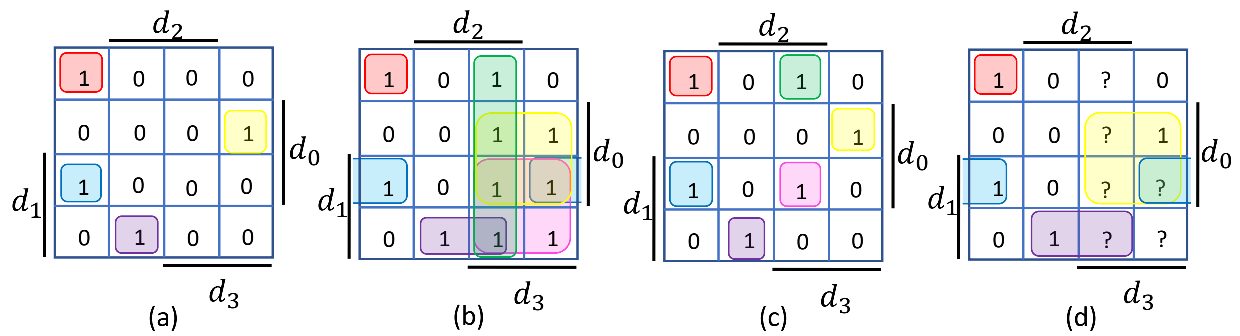


Figure 3.10: Kmaps and PIs for the options in resolving the don't cares.

minimal POS expression for our example, also results in an expression of 12 literals, however, this POS expression corresponds to a different boolean expression.

$$f_2(d_3, d_2, d_1, d_0) = \sum m(0, 3, 6, 9, 11) = (\overline{d_2} + d_1)(\overline{d_2} + \overline{d_0})(\overline{d_3} + d_0)(d_2 + \overline{d_1} + d_0)(d_3 + d_1 + \overline{d_0})$$

3.4 Beyond Kmaps

The Kmap method in Section 3.3.3 for finding minimal SOP expressions quickly becomes impractical as the number of variables grows. Identifying PIs in Kmaps with more than 6 variables is complex. And even with 5 or 6 variable Kmaps, it can be challenging to correctly identify them. In this section, we describe the Quine-McCluskey method which can handle a large number of variables and is suitable for automation. Though this method will find the minimal SOP expressions, alternate heuristic methods will be needed to handle larger problems.

We will use the function $f(x, y, z, w) = \sum m(0, 1, 4, 5, 6, 7, 9, 10, 11, 14)$ whose Kmap is shown in Figure 3.11 to illustrate the method.

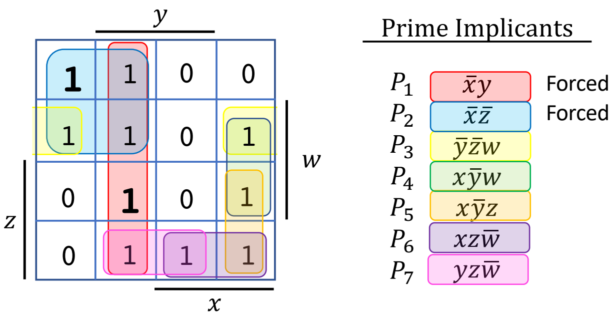


Figure 3.11: The prime implicants (PIs) of $\sum m(0, 1, 4, 5, 6, 7, 9, 10, 11, 14)$.

3.4.1 Tabular method for generating PIs

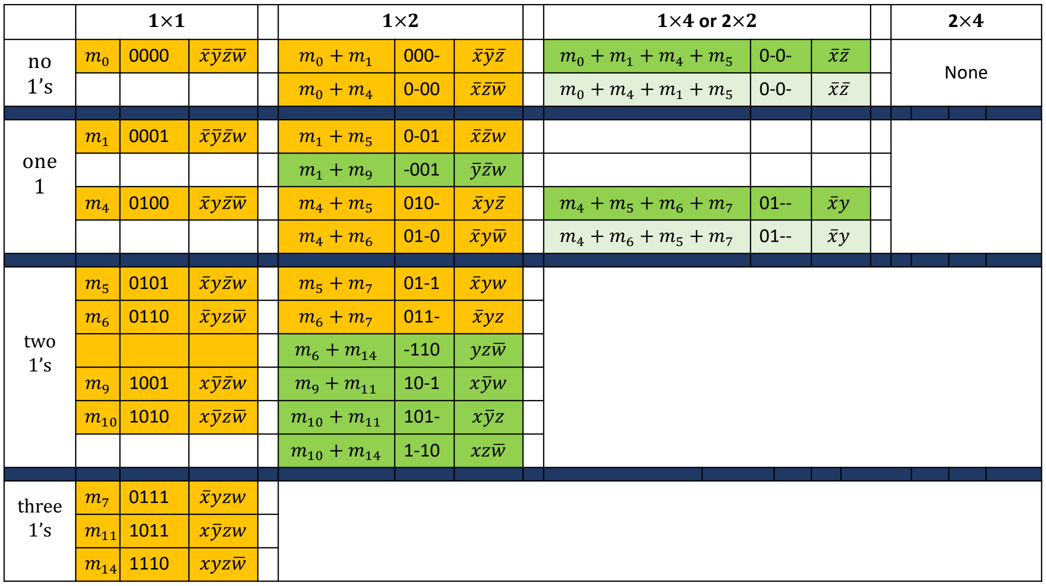
Finding prime implicants in Kmaps is practical for up to 5 or 6 variables. Beyond that, we can use the Quine-McCluskey tabular method. This method generates implicants in increasing size beginning with the minterms which are 1×1 cubes. Given a function of n variables, an implicant (a product term) is represented by a string of 0's, 1's, and -'s. The i^{th} position in this string corresponds to the i^{th} variable of the function. The i^{th} position of the string for a product term will be

0	if the i^{th} variable is complemented in the term
1	if the i^{th} variable is uncomplemented in the term
-	if the i^{th} variable does not appear in the term

Here are some examples of the strings that represent product terms for the variables x, y, z, and w.

$\overline{x}\, y\, \overline{z}\, \overline{w}$	0100
$\overline{x}\, \overline{y}\, w$	00-1
x	1---
$x\, \overline{z}$	1-0-
1	----

The process begins by grouping the minterms for the ONSET of the function by the number of 1's in their associated strings. In Figure 3.12 there are four horizontal sections separated by the dark blue lines for the implicants with no, one, two, and three 1's in their strings. In the



	1×1			1×2			1×4 or 2×2			2×4
no 1's	m_0	0000	$\bar{x}\bar{y}\bar{z}\bar{w}$	$m_0 + m_1$	000-	$\bar{x}\bar{y}\bar{z}$	$m_0 + m_1 + m_4 + m_5$	0-0-	$\bar{x}\bar{z}$	None
				$m_0 + m_4$	0-00	$\bar{x}\bar{z}\bar{w}$	$m_0 + m_4 + m_1 + m_5$	0-0-	$\bar{x}\bar{z}$	
one 1	m_1	0001	$\bar{x}\bar{y}\bar{z}w$	$m_1 + m_5$	0-01	$\bar{x}\bar{z}w$				
				$m_1 + m_9$	-001	$\bar{y}\bar{z}w$				
	m_4	0100	$\bar{x}y\bar{z}\bar{w}$	$m_4 + m_5$	010-	$\bar{x}y\bar{z}$	$m_4 + m_5 + m_6 + m_7$	01--	$\bar{x}y$	
				$m_4 + m_6$	01-0	$\bar{x}y\bar{w}$	$m_4 + m_6 + m_5 + m_7$	01--	$\bar{x}y$	
two 1's	m_5	0101	$\bar{x}y\bar{z}w$	$m_5 + m_7$	01-1	$\bar{x}yw$				
	m_6	0110	$\bar{x}yz\bar{w}$	$m_6 + m_7$	011-	$\bar{x}yz$				
				$m_6 + m_{14}$	-110	$yz\bar{w}$				
	m_9	1001	$x\bar{y}\bar{z}w$	$m_9 + m_{11}$	10-1	$x\bar{y}w$				
	m_{10}	1010	$x\bar{y}z\bar{w}$	$m_{10} + m_{11}$	101-	$x\bar{y}z$				
				$m_{10} + m_{14}$	1-10	$xz\bar{w}$				
three 1's	m_7	0111	$\bar{x}yzw$							
	m_{11}	1011	$x\bar{y}zw$							
	m_{14}	1110	$xyz\bar{w}$							

Figure 3.12: Tabular method for finding PIs of $\sum m(0, 1, 4, 5, 6, 7, 9, 10, 11, 14)$.

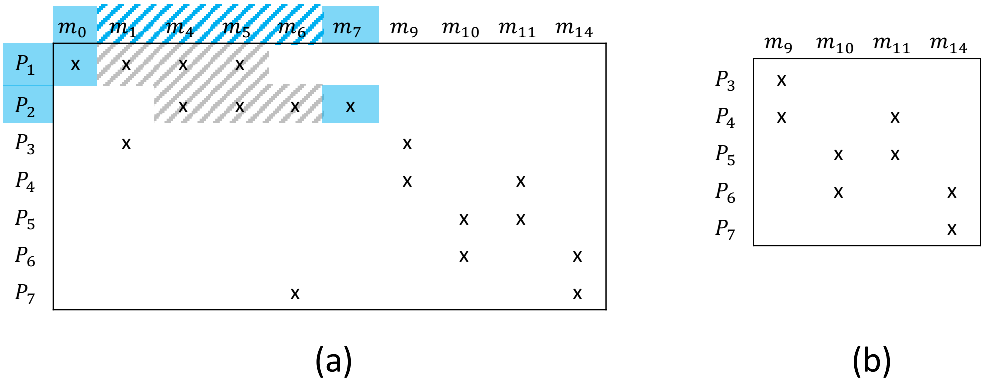
first three columns, we have the minterms (1×1 rectangles) that correspond to our ONSET. In the second group of three columns, the minterms have been combined to form implicants of size 2. Two minterms can be combined if the polarity (complementation) of their variables is the same except for exactly one variable. In our example, $m_9 = x\,\overline{y}\,\overline{z}\,w(1001)$ can be combined with $m_{11} = x\,\overline{y}\,z\,w(1011)$ since only the polarity of variable z is different.

The method continues, generating the implicants of size 2^{k+1} by pairing up two implicants of size 2^k. Two implicants (product terms) can be combined if they have the same variables present (so they must have -'s in the same positions) and exactly one of their variable's polarity is different. For example, in $\overline{x}\,y\,\overline{z}$(010-) can be combined with $\overline{x}\,y\,z$(011-) to form $\overline{x}\,y$(01--) since only the polarity of variable z is different. But $\overline{x}\,y\,\overline{z}$(010-) cannot be combined with $\overline{x}\,\overline{y}\,z$(001-), nor $\overline{x}\,y\,w$(01-1). Since two implicants can be combined when they have the same number of -'s and their number of 1's differs by one, we need only consider combining implicants with the implicants in the section just below.

If we can combine an implicant, then it is not prime since it is covered by a larger implicant. The implicants we were able to combine are in gold in Figure 3.12: they are not prime. The green implicants were not combined: they are prime. Note that we generated duplicates for the implicants of size 8 (light green). In general, an implicant with k -'s will be generated k times. There are a total of seven deep green implicants corresponding to the seven prime implicants as confirmed in Figure 3.11.

3.4.2 PI chart for selecting a cover

Once we have identified the PIs as in Figure 3.11 we can create the chart shown in Figure 3.13(a).



(a)

	m_0	m_1	m_4	m_5	m_6	m_7	m_9	m_{10}	m_{11}	m_{14}
P_1	x	x	x	x						
P_2			x	x	x	x				
P_3		x					x			
P_4							x		x	
P_5								x	x	
P_6								x		x
P_7					x					x

(b)

	m_9	m_{10}	m_{11}	m_{14}
P_3	x			
P_4	x		x	
P_5		x	x	
P_6		x		x
P_7				x

Figure 3.13: The PI charts for $\sum m(0, 1, 4, 5, 6, 7, 9, 10, 11, 14)$.

In this chart, there is a column for each element of the ONSET (identified by its minterm) and a row for each PI. We place an "x" in a location if the column's minterm is covered by the row's PI. We can then identify the minterms that have only one x in their column (m_0 and m_7). These minterms force the PIs that correspond to their "x". In Figure 3.13(a), the forcing minterms and forced PIs are shown in blue. We can then mark all of the minterms covered by the forced PIs (striped). This accomplishes Steps 2 and 3 of our procedure from Section 3.3.3. Removing the rows of the covered minterms and columns of the forced PIs results in the reduced chart in Figure 3.13(b). In the reduced chart each minterm will have at least two "x"s in its column. When the chart is this small we can see by inspection that selecting P_4 and P_6 is optimal. When it is larger, we can use Petrick's method which forms a boolean expression that corresponds to the selections of PIs that complete the cover. Specifically, we create a boolean variable for each non-forced PI, say p_i. For each remaining uncovered minterm of the ONSET, we form a sum term with the variables of the PIs that cover it. The product of these sum terms is a POS expression that evaluates to 1 only if the PIs whose variables are 1 complete the cover. The expression we would obtain for the chart in Figure 3.13(b) is

$$(p_3 + p_4)(p_5 + p_6)(p_4 + p_5)(p_6 + p_7)$$

Using boolean algebra we can transform this expression to an SOP expression. Using the Idempotent and Absorption Laws will help reduce the number of terms during the expansion.

$$(p_3 + p_4)(p_5 + p_6)(p_4 + p_5)(p_6 + p_7) \quad = \quad p_3p_5p_6 + p_3p_5p_7 + p_4p_5p_7 + p_4p_6$$

For this SOP expression to evaluate to 1, we need at least one product term to evaluate to 1. Each product term's cost can be evaluated by summing the number of literals in the PIs associated with its variables. For example the term $p_3p_5p_6$ from the equation above will produce an expression with $2 + 2 = 2 = 6$ literals since P_3, P_5, and P_6 all have 2 literals. The best choice to complete the cover is p_4p_6 which results in 4 literals. Hence the best cover for the entire function will consist of P_1, P_2, P_4, and P_6 and the minimal SOP expression is

$$\overline{x}\,y + \overline{x}\,\overline{z} + x\,\overline{y}\,w + x\,z\,\overline{w}.$$

3.5 Multiple outputs and two-level synthesis

When our combinational logic has more than one output it may be advantageous to consider SOP expressions that are not minimal by themselves, but have common terms. Consider the minimal SOP expressions for the following two functions f_1 and f_2:

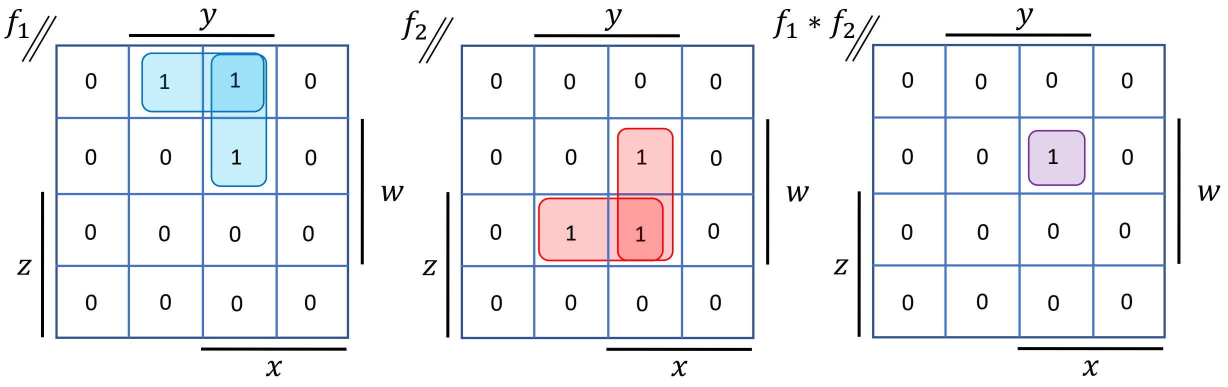
$$\begin{aligned} f_1(x,y,z,w) &= \overline{y}\,\overline{z}\,\overline{w} + x\,y\,\overline{z} \\ f_2(x,y,z,w) &= y\,z\,w + x\,y\,w \end{aligned}$$

We instead could use the following expressions that have a total of 10 literals.[3]

$$\begin{aligned} g &= x\,y\,\overline{z}\,w \\ f_1(x,y,z,w) &= \overline{y}\,\overline{z}\,\overline{w} + g \\ f_2(x,y,z,w) &= y\,z\,w + g \end{aligned}$$

From the Kmaps of f_1 and f_2 in Figure 3.14 (left and middle) it is shown that the original expressions given for f_1 and f_2 were minimal SOP expressions. The Kmap on the right is for the product

[3] Here g is not an input variable and hence does not count as a literal.



0	1	1	0
0	0	1	0
0	0	0	0
0	0	0	0

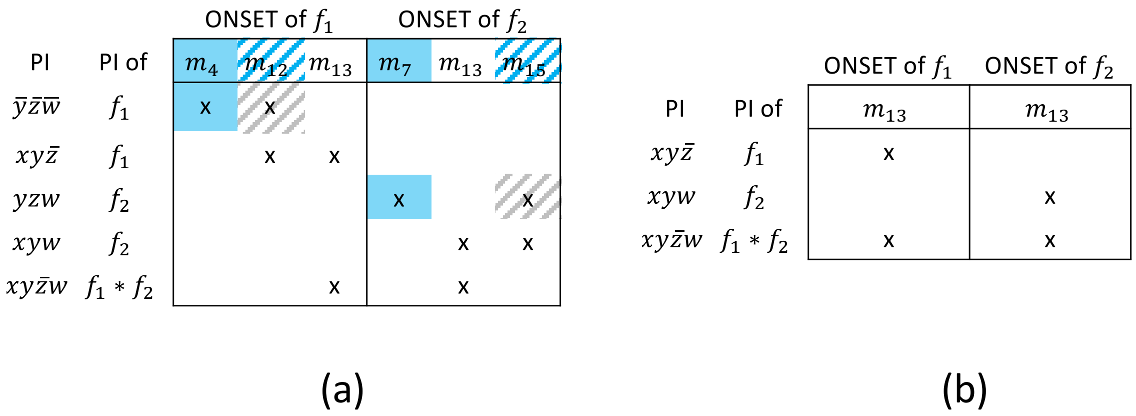
0	0	0	0
0	0	1	0
0	1	1	0
0	0	0	0

0	0	0	0
0	0	1	0
0	0	0	0
0	0	0	0

Figure 3.14: The Kmaps for minimal multiple function SOP expressions.

of f_1 and f_2. The implicant $x\,y\,\bar{z}\,w$ is neither a prime implicant of f_1 nor f_2 but it is a prime implicant of their product $f_1 * f_2$. Including an implicant which is not prime for either f_1, f_2, nor $f_1 * f_2$ will not be minimal since we can replace it with a larger implicant (product term with fewer literals).

To form the cover, create a PI chart as described in Section 3.4.2. The chart will now have separate sections for the ONSETs of the two functions. As before the PIs are in the leftmost column, and an "x" indicates that the row's PI covers the column's minterm. But the PIs of f_1 or $f_1 * f_2$ can be used only for f_1's cover, while the PIs of f_2 or $f_1 * f_2$ can be used for f_2's cover. The chart



		ONSET of f_1			ONSET of f_2		
PI	PI of	m_4	m_{12}	m_{13}	m_7	m_{13}	m_{15}
$\bar{y}\bar{z}\bar{w}$	f_1	x	x				
$xy\bar{z}$	f_1		x	x			
yzw	f_2				x		x
xyw	f_2					x	x
$xy\bar{z}w$	$f_1 * f_2$			x		x	

(a)

		ONSET of f_1	ONSET of f_2
PI	PI of	m_{13}	m_{13}
$xy\bar{z}$	f_1	x	
xyw	f_2		x
$xy\bar{z}w$	$f_1 * f_2$	x	x

(b)

Figure 3.15: PI chart for minimal multiple function SOP expressions.

for our example is in Figure 3.15(a). The minterm m_{13} is listed in both sections, but it is not covered by $x\,y\,\bar{z}$ for f_2, nor $x\,y\,w$ for f_1. But $x\,y\,\bar{z}\,w$ will cover m_{13} in both functions. Removing the rows of the forced PIs and the columns of the cells they cover, we obtain the reduced chart in Figure 3.15(b). Selecting the PI $x\,y\,\bar{z}\,w$ covers the remaining two cells.

The procedure to find a set of minimal SOP expressions for two functions is summarized as follows.

1. Identify the **PI**s of the three functions f_1, f_2, and $f_1 * f_2$.
2. Identify the cells of the f_1's ONSET that force a PI of f_1 or $f_1 * f_2$.
3. Add the forced PIs to the cover for f_1 and mark the cells of f_1 they cover.
4. Identify the cells of the f_2's ONSET that force a PI of f_2 or $f_1 * f_2$.
5. Add the forced PIs to the cover for f_2 and mark the cells of f_2 they cover.
6. Select a "smallest" group of the remaining PIs to cover any remaining uncovered cells of f_1 and f_2.

This procedure can be extended to k functions by considering the PIs of the products of any subset of the k functions. For example, with three functions we would obtain the PIs of f_1, f_2, f_3, $f_1 * f_2$, $f_1 * f_3$, $f_2 * f_3$, and $f_1 * f_2 * f_3$.

3.6 ESPRESSO, MISII, and BDDs

By the early 1980's the growth in the number of gates that could be placed on one integrated circuit was exceeding what could be handled without automation. Computer-assisted design (CAD) tools became indispensable for designing integrated circuits at this larger scale. Thus began a symbiotic cycle between computer-assisted design tools and circuit technologies. Faster processing and greater storage capacity were needed for the tools that would support the design of faster processors and larger storage devices.

The methods described in Section 3.4 can be automated, but even these limit the size of the functions that can be handled (the number of PIs grows exponentially with the number of variables). New methods, algorithms, and heuristics were needed. Three of the most notable for logic synthesis were ESPRESSO, MIS, and BDDs [4, 5, 6]. ESPRESSO was a program developed at UC Berkeley for two-level SOP/POS logic synthesis. MIS and MISII, also from UC Berkeley, provided a set of operations for multi-level combinational logic (sets of boolean equations) that could be combined in scripts. The Binary Decision Diagram (BDD) provides a compact representation of boolean functions that is now widely used in CAD tools.

Chapter 4

Synchronous Sequential Circuits

In previous chapters, we have analyzed and synthesized combinational circuits. Our combinational circuits were assembled without loops from basic gates (acyclic graphs) and their settled behavior could be represented by boolean functions of their inputs. But not all computations can be performed without "memory" or "state." In some cases, it may be necessary to "take notes" based on previous input values. Even when a combinational circuit is feasible, it may be more efficient to perform the computation in steps using a smaller circuit to iterate. Sequential circuits have "memory." The logic values of their outputs depend not only on the current values of the circuit inputs but also on their previous values.

When there are loops (feedback) in our circuits, analyzing and predicting their behavior can be complicated. Even their settled behavior may depend on the physical properties of the gates, wiring, and the circuit's environment. Timing can not only affect when the output values will be available but also the values themselves. We can simulate our designs, but the accuracy of the simulation will depend on estimates of the physical characteristics. These estimates may only be sufficiently accurate in the final stages and still have margins of error. And Murphy's Law ensures that if something can go wrong, it will.[1] Fortunately, we can handle this issue by following some simple rules that result in a *synchronous sequential design.*

Synchronous sequential circuit design follows some simple rules so that the behavior of the circuit can be reliably predicted. The following ingredients are needed:

1. A *clock* signal with which the external inputs and memory elements of the circuit will be synchronized.

2. A *memory* device that updates its logic value only in response to the clock signal.

3. *Synchronized inputs* that are valid with respect to the clock.

[1] And no doubt this will occur first when it causes the greatest cost, harm, and embarrassment – a corollary to Murphy's Law.

4.1 The clock

Synchronous sequential circuits make use of an input that is designated as a *clock*. The clock signal alternates between 0 and 1 at a fixed rate referred to as its *frequency*. One full oscillation (say from 0 to 1 back to 0) is a *clock cycle*. The elapsed time for a clock cycle is the *clock period* and the frequency is the reciprocal of the clock period. The clock `clk` shown in the timing diagram in

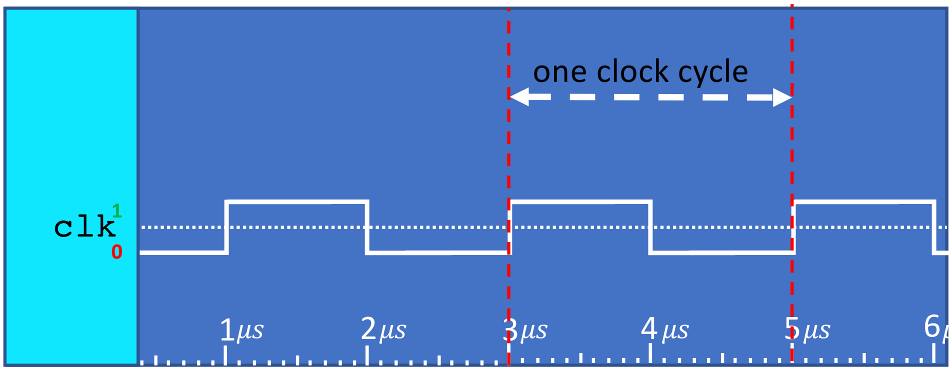


Figure 4.1: A 0.5MHz clock.

Figure 4.1 has a clock period of 2 microseconds. Its frequency is

$$\frac{1}{2*10^{-6}\text{ secs}} = \frac{5*10^{5}}{\text{secs}} = 0.5*10^{6}\text{ Hz} = 0.5\text{ MHz}.$$

Entire texts have been written on generating and wiring clocks. For the sake of simplicity, we will assume that a suitable single clock signal is provided and that any clock issue (skew, power, noise, jitter, start-up) will not be a concern. But these are certainly concerns in high-performance designs.

4.2 Basic memory device: the D Flip-Flop

We begin with the most basic memory device: the positive edge-triggered D flip-flop represented by the symbol in Figure 4.2. This component remembers a 1-bit value. The lower pin on the left

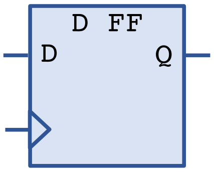


Figure 4.2: The positive edge-triggered D Flip Flop.

is the clock input. The triangle on this pin indicates that this input is sensitive to a rising edge (the transition from 0 to 1) rather than either logic level. The `D` input pin provides the value to be loaded at the next rising clock edge and the `Q` output pin is the current stored value: the state of this flip-flop.

The timing diagram in Figure 4.3 illustrates the operation of a D FF. Specifically,

- The `Q` output will remain unchanged until **after** a positive clock edge.
- On or near the rising edge of the clock input, the logic value of the `D` pin will be loaded into the flip-flop, and after some delay the output `Q` will transition from the previous to the new value (if it is different) and this happens shortly **after** the clock edge.[2] In our timing diagrams, the time between the clock edge and `Q`'s transition is exaggerated to clarify that the transition is after the clock edge.

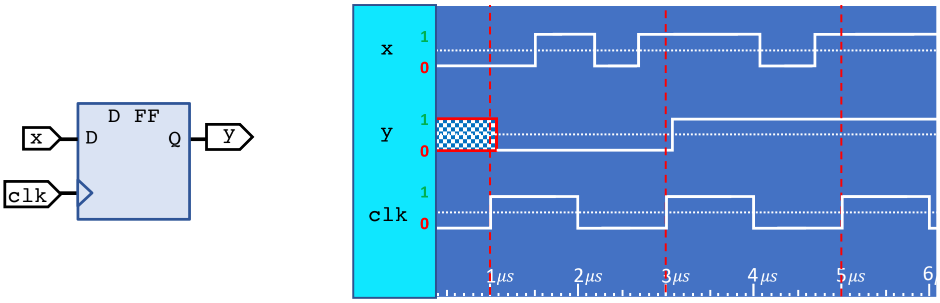


Figure 4.3: Basic operation of the positive edge-triggered D Flip Flop.

In Figure 4.3 the value of the `Q` output is "unknown" before $1\mu s$ since the value stored in this flip-flop before the first load could be either 0 or 1. There is one more thing important to know about D FFs:

WARNING:
To correctly operate the D FF, the `D` input must be valid and stable for a specified small window of time before and after the rising edge of the clock input. Failure to meet this condition can result in unpredictable behavior of the output `Q` even possibly instability.

Following the rules in Section 4.2.1 allow us to check that this requirement is met by our design.

4.2.1 Rules for synchronous design

The following rules will result in a sequential circuit that is synchronous with the input signal `clk`.

1. Each component has no more than one input pin designated as its clock input, and it should be labeled `clk`.

[2] When the circuit contains only devices sensitive to one type of clock edge the term clock edge refers to that type.

2. At the top level and within each component the `clk` net should be connected directly to the clock input pin of all components with clock pins (including all D FFs).
3. The signal `clk` should not be an input to any logic gate or a component pin other than the component's clock pin.
4. The external inputs to the circuit should be synchronous with `clk`. (When the intended external inputs are asynchronous, they need to be synchronized as discussed in Section 4.8.)
5. There are no loops in the logic diagram that do not traverse at least one D FF. That is, removing the D FFs would leave a combinational logic circuit.

Adhering to these rules ensures that the timing constraints for the D FFs can be easily verified. These constraints and how to satisfy them are discussed in Section 4.9. Furthermore, the outputs of such a circuit are synchronized with `clk` and can be used directly as inputs to other components sharing `clk` as their clock. Until Section 4.9 we will assume that the FF input values are stable and valid around the clock edge as required.

4.3 A simple shift register

Our first example is the 3-bit register shown in Figure 4.4. The three D FFs store three bits. At the rising clock edge, the logic values of the flip-flops shift to the right with the leftmost flip-flop assuming the value of the input `a`. The timing diagram in Figure 4.5 shows the operation of the 3-bit shift register with a 0.2 MHz clock. The flip-flops are not initialized, so their values are unknown until they load a known value. This happens at the first clock edge for the leftmost flip-flop, the second for the middle flip-flop, and the third for the rightmost. The external input `a` is stable around the rising clock edges but can change between them. Only its value at the rising clock edges will be reflected in the later outputs of the flip-flops. The waveforms for the three flip-flop outputs, `b`, `c`, and `d` are the same but shifted one clock cycle.

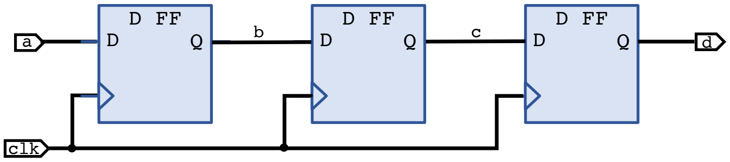


Figure 4.4: A 3-bit shift register.

With this 3-bit shift register, we can detect a 3-bit pattern on consecutive inputs by adding combinational logic to recognize the pattern. In Figure 4.6 AND gates have been added to detect two simple patterns. The output `g` will be 1 when the three stored bits are two 1's followed by a 0. The output `e` will be 1 when the last two stored bits are both 1 and the current input is 0. The two outputs `g` and `e` are detecting the same pattern, but as seen in Figure 4.7 they are different since they are based on the values of `a` at different times. The pattern will be detected one clock cycle earlier on the output `e`, but any changes on `a` while `b` and `c` are 1 will be reflected on the output `e`. There are three of these changes indicated by the yellow vertical dashed lines in the 15μs to

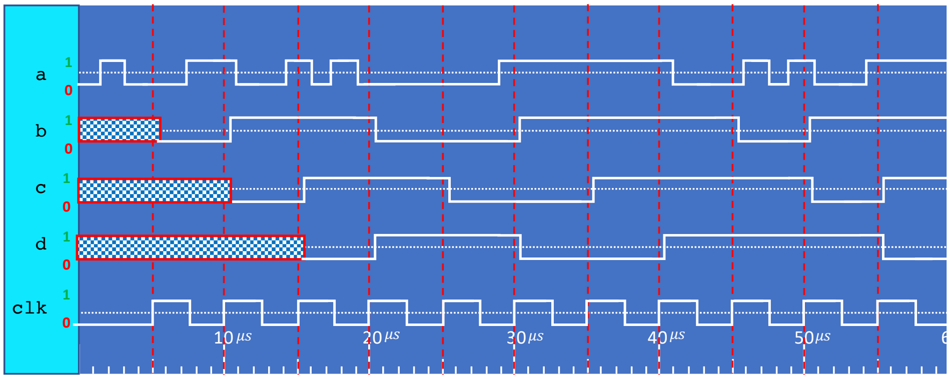


Figure 4.5: Simulation of a 3-bit shift register with D FFs not initialized.

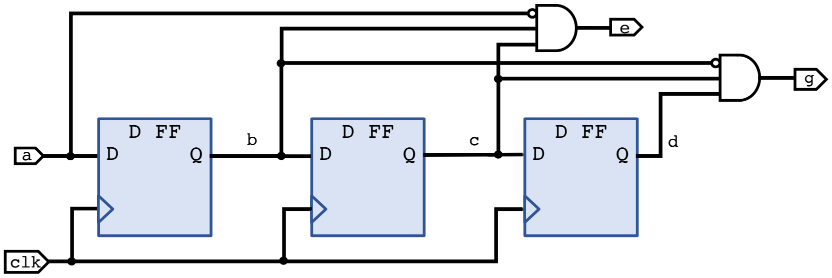


Figure 4.6: Using a 3-bit shift register to detect patterns on the synchronous input a.

$20\mu s$ clock cycle. Since g depends only on the FF outputs it will not be affected by changes in a within the same clock period. We will refer to an output that depends only on the stored bits as a *Moore output* while an output whose value depends on the value(s) of the current input(s) is a *Mealy output*.[3] The behavior of a synchronous sequential design can be represented by the values of its input/outputs at the clock edges. In Figure 4.8 this stream is shown below the timing diagram. We can use functions to represent the behavior of our sequential circuits by introducing time as a variable. Each net's value is represented by a function of the clock edge ordinal (1,2,3,...etc). The output of a D FF at the t^{th} clock edge, $Q(t)$ for $t \geq 1$. For $t > 1$, $Q(t)$ will is the value that was loaded at the $t-1^{th}$ clock edge since this value is retained until after the t^{th} clock edge:

$$\text{for } t > 1 \text{ we have } Q(t) = D(t-1) \text{ and } \; Q(1) \; \text{ is unknown.}$$

Using this notation we can obtain the following equations for each of the nets from the logic diagram.

$$\begin{aligned} b(t) &= a(t-1) \\ c(t) &= b(t-1) \\ d(t) &= c(t-1) \end{aligned}$$

[3] The Moore/Mealy terminology is usually applied to entire designs rather than individual outputs. A Moore machine is a circuit with only Moore outputs, and just one Mealy output makes a circuit Mealy.

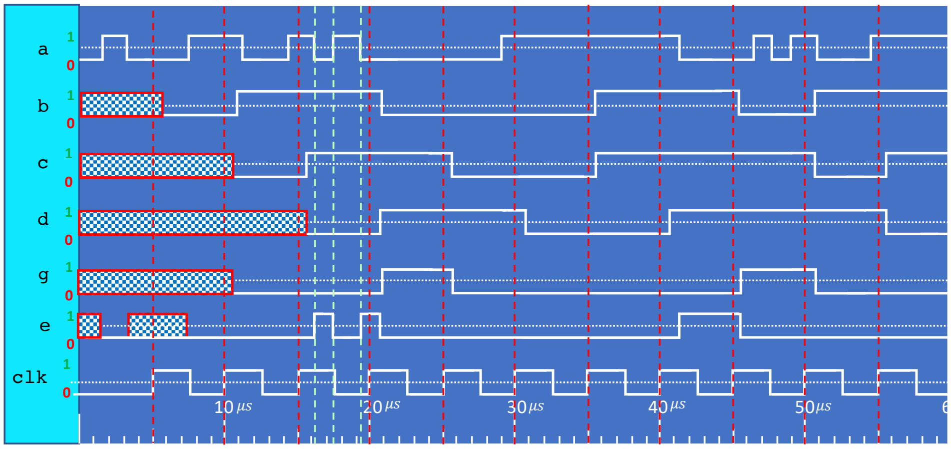
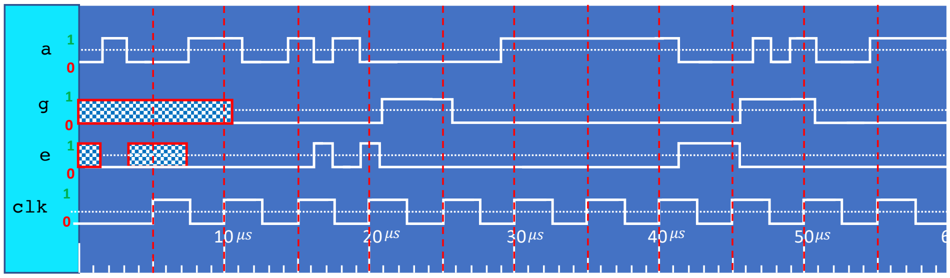


Figure 4.7: Simulation of the pattern detector circuit.



clk t	1	2	3	4	5	6	7	8	9	10	11
a	0	1	1	0	0	1	1	1	0	1	1
g	X	X	0	0	1	0	0	0	0	1	0
e	X	0	0	1	0	0	0	0	1	0	0

Figure 4.8: Input/output stream for the pattern detector.

$$
\begin{aligned}
e(t) &= \overline{a(t)} * b(t) * c(t) \\
g(t) &= \overline{b(t)} * c(t) * d(t)
\end{aligned}
$$

By substitution, we obtain the following equations for the external outputs in terms of the external input:

$$
\begin{aligned}
e(t) &= \overline{a(t)} * a(t-1) * a(t-2) \\
g(t) &= \overline{a(t-1)} * a(t-2) * a(t-3)
\end{aligned}
$$

4.4 Parity checker

Recall that the parity function is 1 when there is an odd number of inputs with the value 1. Here we desire a component with one synchronous input (this implies there is a clock input as well) whose output is 1 when the current input and all the previous inputs together have an odd number of 1's. If we only needed to consider the last K inputs we could use a shift register and remember the last K, but there is no limit on how many of the past inputs can affect the current output. Fortunately, we don't need to remember the last K inputs exactly: we just need to know the parity of all of these previous inputs together. We can write an equation for the current output $y(t)$ in terms of the current and previous inputs ($x(t)$, $x(t-1)$, $x(t-2), \ldots, x(2)$, $x(1)$) and then substitute $y(t-1)$ for the parity of the previous inputs.

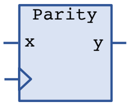


Figure 4.9: Symbol for parity checker.

$$\begin{aligned} \text{If} \quad y(t) &= \quad x(t) \oplus x(t-1) \oplus x(t-2) \oplus \cdots \oplus x(2) \oplus x(1) \\ \text{then} \quad y(t) &= \quad x(t) \oplus y(t-1) \end{aligned}$$

Using this equation we can build the circuit in Figure 4.10. The output `Q` of the D FF holds $y(t-1)$ since this is the value that was loaded into the FF at the previous clock edge. The simulation of the circuit in Figure 4.10 is shown in Figure 4.11.

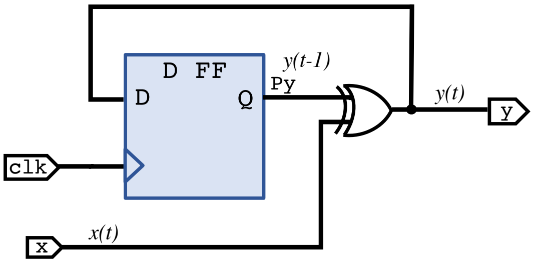


Figure 4.10: Sequential circuit for parity checker.

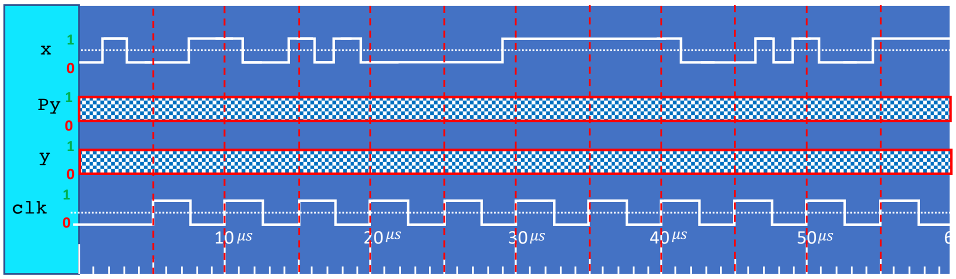
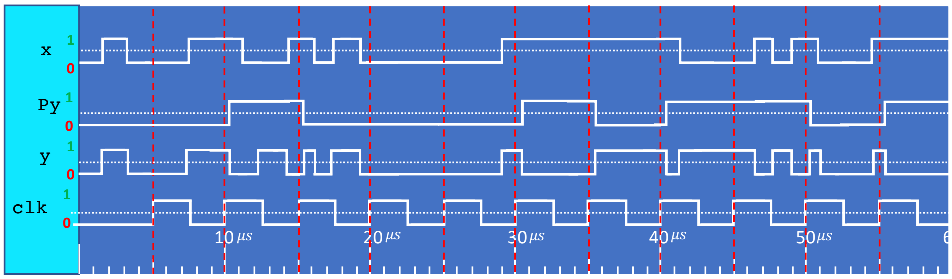


Figure 4.11: Simulation of uninitialized parity checker circuit.

Because the D FF was not initialized, the output of the XOR gate will be unknown even when the input `x` is 0 or 1. This unknown value will be loaded into the D FF and this will continue. The D

FF will never have a known value if it is initially unknown. For this design, the D FF needs to be initialized to 0 since this represents the parity of the number of 1's at the start.[4] In Figure 4.12. the circuit in Figure 4.10 is simulated assuming that the D FF is initially 0.



t	1	2	3	4	5	6	7	8	9	10	11
x	0	1	1	0	0	1	1	1	0	1	1
Py	0	0	1	0	0	0	1	0	1	1	0
y	0	1	0	0	0	1	0	1	1	0	1

Figure 4.12: Simulation of parity checker when the D FF is initially 0.

Just as providing power, the initialization of memory elements in a sequential design is an issue that must be considered. Depending on the implementation technology, initialization of memory elements may or may not be provided. In FPGAs, some of the memory elements may be initialized to 0 and there is built-in circuitry that your design can access to re-initialize during operation. Pay attention to instructions and/or descriptions of devices to ensure that the initialization of memory elements is addressed and that any simulation tools reflect this initialization.

If you must handle the initialization yourself, you will need additional logic in your design. In Figure 4.13(a) an input R has been added to the symbol for the parity checker to indicate when it should be initialized. An AND gate has been added to the circuit as shown in Figure 4.13(b) so that the D FF will load a 0 when the R input is 1 at the clock edge. With this additional circuitry, the parity checker will operate correctly as expected beginning with the first clock edge after the R input returns from 1 to 0 and remains 0.

In some cases, you might need the D FF to be initially 1 rather than 0. You can achieve this by replacing the D FF with the circuit in Figure 4.14.

⚠**WARNING:**
The initialization of memory elements should be synchronized with the clock. Follow the instructions provided with your implementation technology. Under no circumstances should the asynchronous reset/set pins of devices be used either for initialization or as part of the design of a synchronous sequential circuit.

[4] The shift register in the previous section will have known values after 3 clock edges. But if the output value in the first few cycles needs to be 0 then the D FFs in the shift register should be initialized.

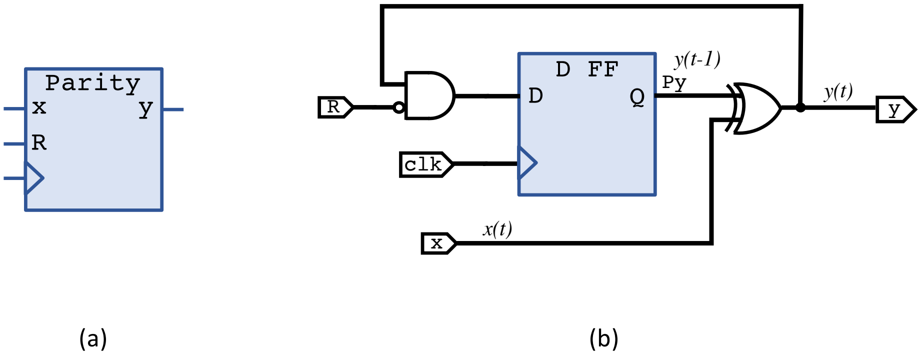


Figure 4.13: (a) Addition of `R` input to the Parity Checker. (b) Additional logic to initialize the D FF to 0 when input `R` is 1.

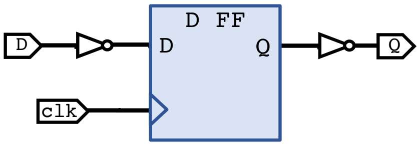


Figure 4.14: Modification of a D FF to be initially 1 rather than 0.

4.5 Counters

One of the most basic and frequent components in digital systems is the counter. There are many varieties (binary, binary-coded decimal, Johnson, ring) with different control options (counting up, counting down, reset, load, etc.). The instruction counter is at the heart of a programmable controller and the speed at which it can increment can limit the clock frequency of the entire system.

Here we will assemble a simple binary counter `BCNT4` with one input, `Inc`, that controls whether the counter increments at the clock edge or keeps its current value. In Chapter 5 we will mechanically synthesize this logic from a truth table, but here the logic will reflect the calculation needed to increment a binary number.

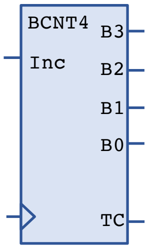


Figure 4.15: Symbol for the simple 4-bit binary counter.

Our 4-bit Binary Counter counts from 0 up to 15. Counting up from 15 returns the counter to 0. The counter will hold a 4-bit value `Q3 Q2 Q1 Q0` representing its current value. Its outputs are `B3 B2 B1 B0` which is its current value and a fifth output, `TC` for terminal count, that will be 1 when the counter is at its highest value (15). The `TC` output will be useful for assembling larger counters and for use in logic controlling the counter.

To understand the logic needed for a counter the sequence of values the counter takes on as it advances is shown in Figure 4.16. A bit is shaded if its value has changed from the previous count. For example, `Q2` changes to 0 at 8 and is shaded since it was 1 at 7. The first bit `Q0` changes value on every increment. The next bit, `Q1` changes when `Q0` is 1 in the previous count. For `Q2` to change, both `Q0` and `Q1` have to be 1. And `Q3` changes when the other three bits are 1. Incrementing is the operation of adding 1 shown in Figure 4.17. For the $\mathtt{Q}_i$ bit to change, we need a carry to make its way from $\mathtt{Q}_0$ all the way to $\mathtt{Q}_i$. A carry will only reach $\mathtt{Q}_i$ if all of the bits to $\mathtt{Q}_i$'s right are 1. If any bit to its right is 0, the carry will be absorbed at the first 0 bit and no bit to the left will change. Similarly, a bit will change from 1 to 0 if there is a carry that reaches it and all the bits to its right are 1. Hence $\mathtt{Q}_i$ changes value when `Inc` is 1 and $\mathtt{Q}_{i-1} * \mathtt{Q}_{i-2} * \cdots * \mathtt{Q}_1 * \mathtt{Q}_0$ is 1. The following equation provides the next value of $\mathtt{Q}_i$.

CNT	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2
Q3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0
Q2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0
Q1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1
Q0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

Figure 4.16: Counting sequence for the simple 4-bit binary counter.

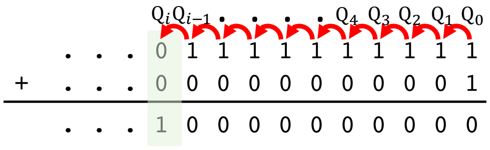

Q_i Q_{i−1} • • • • Q₄ Q₃ Q₂ Q₁ Q₀
. . . 0 1 1 1 1 1 1 1 1 1 1
+ . . . 0 0 0 0 0 0 0 0 0 0 1
. . . 1 0 0 0 0 0 0 0 0 0 0

Figure 4.17: Counting up calculation.

$$\mathtt{D}_i = \mathtt{Q}_i \oplus (\mathtt{Inc} * \mathtt{Q}_{i-1} * \mathtt{Q}_{i-2} * \cdots * \mathtt{Q}_1 * \mathtt{Q}_0)$$

The logic diagram based on this equation is shown in Figure 4.18.[5] The AND gate required grows in number of inputs with each additional bit. Constructing a 16-bit counter in this manner would require a 17-input AND gate. Since each AND gate is the same product as for the previous bit with one more bit, we could also *chain* these AND gates as in Figure 4.19. While the AND gates are all 2-input gates there is now a path with 4 gates (in blue) to the last FF. This is the *carry-chain* of the counter. To build larger counters we can combine smaller counters as in Figure 4.20 where a 16-bit counter has been assembled from four 4-bit counters. Each 4-bit counter advances when `Inc`

[5] We have used the `BUF` symbol here to relabel the FF outputs `Q3 Q2 Q1 Q0` to the external outputs `B3 B2 B1 B0`. The `BUF` symbol does not affect the logic. It may be used in logic diagrams to permit connect nets that have different labels.

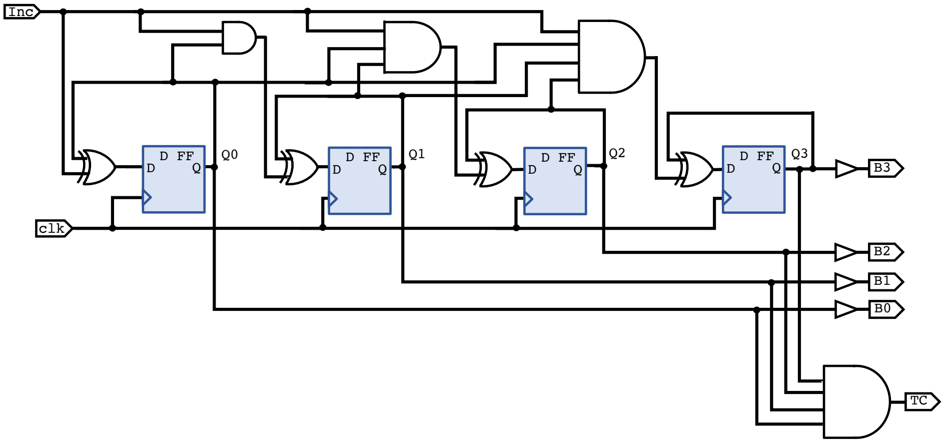


Figure 4.18: Logic diagram for a 4-bit binary counter.

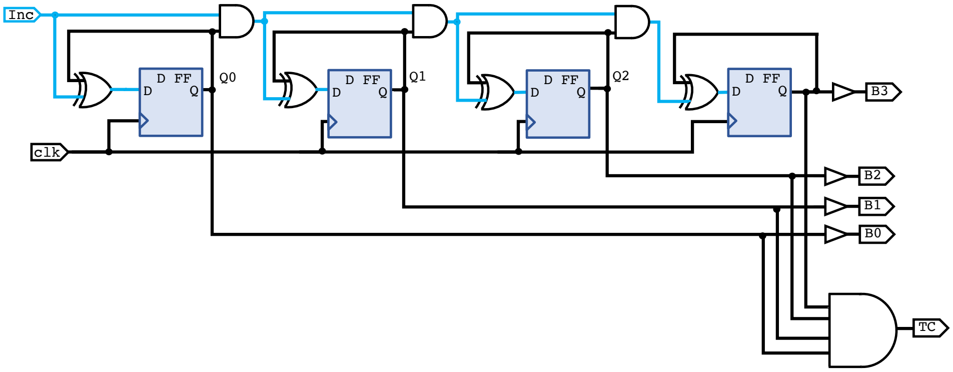


Figure 4.19: Alternative logic diagram for a 4-bit binary counter.

is 1 and the `TC`'s of the previous counters as all 1. Note that the `TC` output of the 4-bit counters in Figures 4.18 and 4.19 did not depend on their `Inc` input. As a result, there is no combinational path through the 4-bit counters. The carry chain of our 16-bit counter is organized as a tree rather than a long chain.

⚠ **WARNING:**
Ripple counters use fewer gates and hence may seem attractive, but they are not synchronous and may cause timing issues. Do not be tempted.

4.6 Registers

In Section 4.3 three FFs were arranged in series to form a simple 3-bit shift register. To build more complex registers, FFs with an additional input, an enable `E`, are handy. The symbol for such a flip-flop is shown in Figure 4.21(a). The additional `E` input controls whether or not the `D FF` loads at the rising clock edge. When `E` is 1 the operation is the same as a `D FF`, but while `E` is 0 the D FF does not load the `D` input when there is a rising clock edge. The `DE FF` may be available

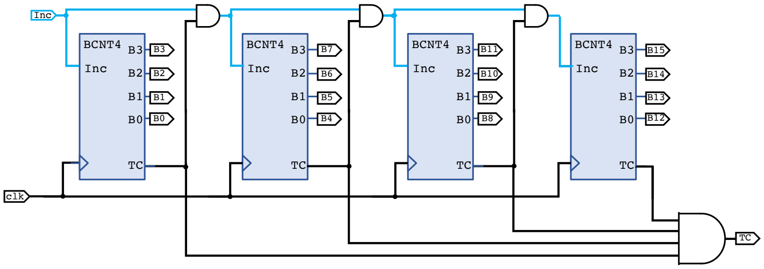


Figure 4.20: Logic diagram for a 16-bit binary counter constructed from 4-bit counters.

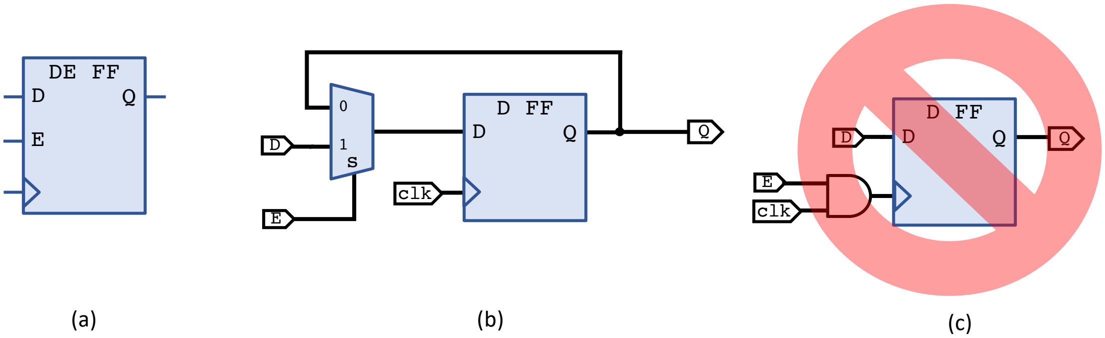


Figure 4.21: The `DE FF` flip-flop: (a) symbol, (b) implementation, and (c) misguided implementation.

as a basic component.[6] If not, it can be implemented by adding a 2-input multiplexer to the `D` input of a `D FF` as shown in Figure 4.21(b). The `D FF` in Figure 4.21(b) will load at every rising clock edge, but when `E` is 0, it will reload its current value rather than the `D` input. So in effect, it will appear that the FF holds its current value when `E` is 0. Figure 4.21(c) might be the first idea for an implementation since the `AND` gate blocks the clock edge when `E` is 0, **but this design is incorrect!** It is a violation of the rules of synchronous sequential circuit design: there is logic on the clock input of the FF. Consider what will happen if `E` transitions from 0 to 1 while `clk` is 1. This would result in a rising edge on the clock input of the `D FF` in the middle of a clock cycle, loading whatever value happened to be present on `D` at the time.

A `DE FF` can be thought of as a 1-bit register with the `E` input serving as the load control. Various `N`-bit registers can be built with `N DE FFs`. In Figure 4.22 the `DE FF` is used to construct three different components.[7] Figure 4.22(a) stores an `N`-bit value when the `Load` input is 1 at the clock edge. Figure 4.22(b) is an `N`-bit shift register. The bits are shifted right when the `Shift` input is 1 at the clock edge and the value of `In` is loaded into the leftmost FF. Figure 4.22(c) is a ring counter. It is constructed from the shift register by connecting the rightmost output of the shift register to the input `In`. The FFs of the ring counter must be initialized to 0 since the values in the ring counter circulate. A 5-bit ring counter will have the values `10000`, `01000`, `00100`, `00010`,

[6] In this case the enable input may be labeled `CE` for clock enable.

[7] Design entry of these registers will benefit from the use of bus structures rather than instantiating individual FFs.

00001, advancing to the next value when the Inc input is 1 at the clock edge. This is an example of one-hot state encoding that we will see in Section 5.4.

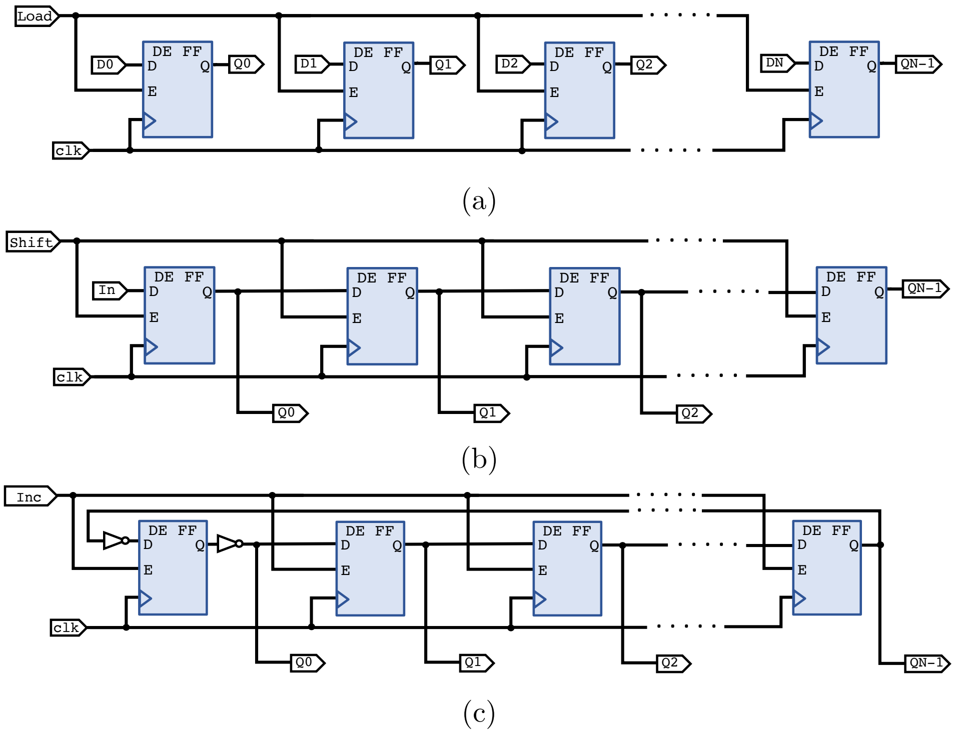


Figure 4.22: Registers: (a) basic N-bit register, (a) N-bit shift register, (c) N-bit ring counter. The FFs in the ring counter must be initialized to 0.

4.7 Analysis of Synchronous Sequential Circuits

Without labels on the inputs, outputs, or internal nets, a synchronous sequential circuit is a spaghetti of gates, FFs, and wires. When it is a synchronous design, the external clock input can be identified since it will be the net connected to all of the FF clock inputs and nothing else. After extracting the FFs and the clock net, what is left is combinational logic (the cloud). The external inputs and FF outputs are inputs to the combinational cloud. The output of the combinational cloud provides the external outputs and the D inputs to the FFs. Figure 4.23 shows this arrangement.

We will refer to the value of the outputs of the FFs (Q's) as the present state of the circuit. The state does not change before the next clock edge. The inputs of the FFs (D's) will be loaded into the FFs at the next clock edge: the value of the D's are the next state. By analyzing the logic in the combinational cloud we can determine for each possible present state and value of the inputs, the next state, and the value of the outputs.

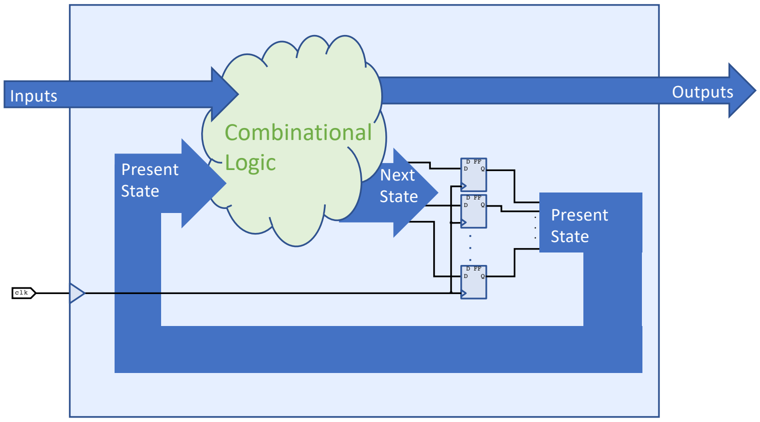


Figure 4.23: Model of synchronous sequential circuits.

4.7.1 Example 1 sequential circuit analysis

We will use the circuit in Figure 4.24(a) as an example for our analysis. As mentioned we can think of the FF outputs as inputs to the combinational logic cloud and the FF inputs are outputs of the combinational logic cloud. In Figure 4.24(b) we have sliced the FFs in half to reflect this.

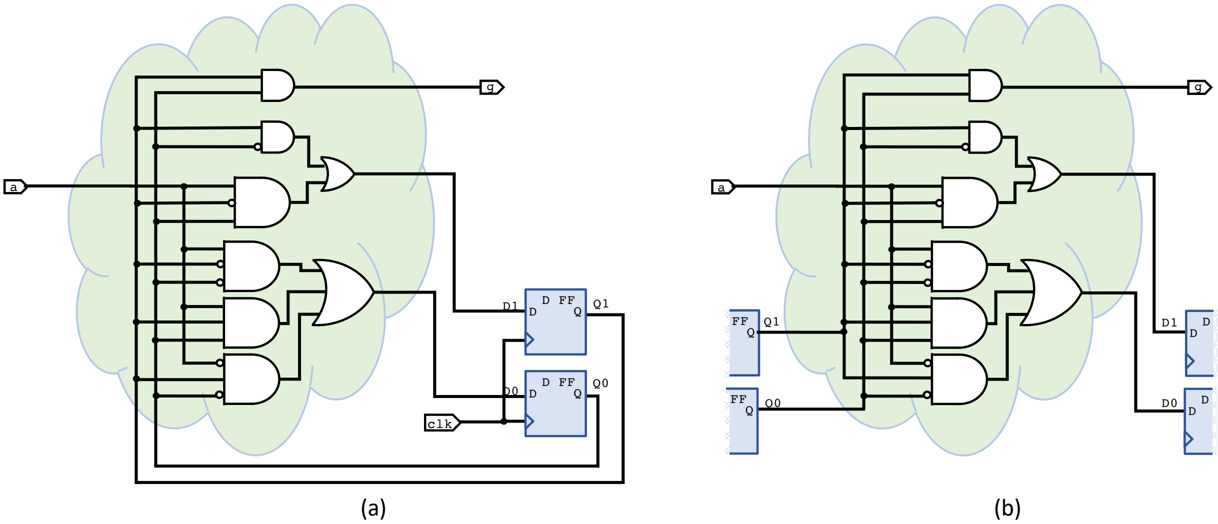


Figure 4.24: Example 1 sequential circuit analysis.

From the combinational cloud, we obtain the following equations for the outputs of the combinational logic cloud:

$$\begin{aligned} g &= Q_1\,Q_0 \\ D_1 &= Q_1\,\overline{Q}_0 + a\,\overline{Q}_1\,Q_0 \\ D_0 &= a\,\overline{Q}_1\,\overline{Q}_0 + a\,Q_1\,Q_0 + \overline{a}\,Q_1\,\overline{Q}_0 \end{aligned}$$

From these equations, the truth table shown in Table 4.1(a) is obtained. For a sequential circuit, it is more convenient to present this table in the format shown in Table 4.1(b): a State Transition Table. In the State Transition Table, each row corresponds to a value of the state variables. The next state and output values are listed in columns for the separate input values. In this example, the equation for the output g does not depend on the input a so only one column is needed.

Q_1	Q_0	a	D_1	D_0	g
0	0	0	0	0	0
0	1	0	0	0	0
1	0	0	1	1	0
1	1	0	0	0	1
0	0	1	0	1	0
0	1	1	1	0	0
1	0	1	1	0	0
1	1	1	0	1	1

(a)

		$D_1\ D_0$				Output
Q_1	Q_0	$a = 0$		$a = 1$		g
0	0	0	0	0	1	0
0	1	0	0	1	0	0
1	0	1	1	1	0	0
1	1	0	0	0	1	1

(b)

Table 4.1: (a) Truth table and (b) State Transition Table for analysis Example 1.

Table 4.1(b) is an *encoded* State Transition Table. That is, the state is represented by boolean variables. By assigning labels to the possible values of the state variables as in Table 4.2(a) and using these labels we obtain the *symbolic* State Transition Table in Table 4.2(b). From the symbolic State Transition Table, the *state diagram* in Figure 4.25 is obtained. Because each state is represented by a large circle, a bubble, this diagram is often referred to as a bubble diagram. The arrows between the circles correspond to the transitions that occur on the clock edge. Specifically, there is an arrow from state X to state Y labeled with the boolean expression E if the next state will be Y when the present state is X and E is 1. Here, the value of the output g is given in the states. The state diagram has the same information as the symbolic state transition table, but in a format where the flow between states is easier to grasp. The state diagram is often useful in understanding a sequential circuit.

Q_1	Q_0	State
0	0	A
0	1	B
1	0	C
1	1	D

(a)

Present	Next State		Output
State	$x = 0$	$x = 1$	g
A	A	B	0
B	A	C	0
C	D	C	0
D	A	B	1

(b)

Table 4.2: State encoding and symbolic State Transition Table for analysis Example 1.

From the diagram in Figure 4.25, we can see that the output is 1 only in state D. State D can only be reached from state C with a 0 input. State C can be reached with two 1's. In fact, from any state, two 1s followed by a 0 will result in state D. This sequential circuit detects two 1's followed by a 0 similar to the circuit in Figure 4.6. The simulation using the same input as Figure 4.7 is

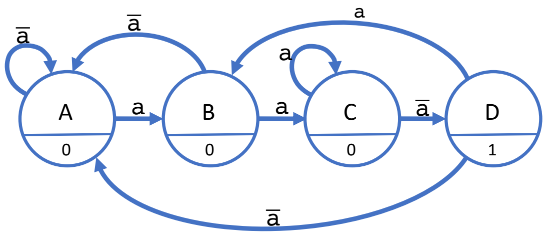


Figure 4.25: State diagram for analysis Example 1.

shown in Figure 4.26. The state corresponding to the value of `Q1 Q0` is shown on a separate line. Here we have assumed the FFs were initially 0 corresponding to state A. Although this circuit would function correctly after two clock edges regardless of whether the FFs are initialized, most simulators would continue to show the state variables as unknown.[8]

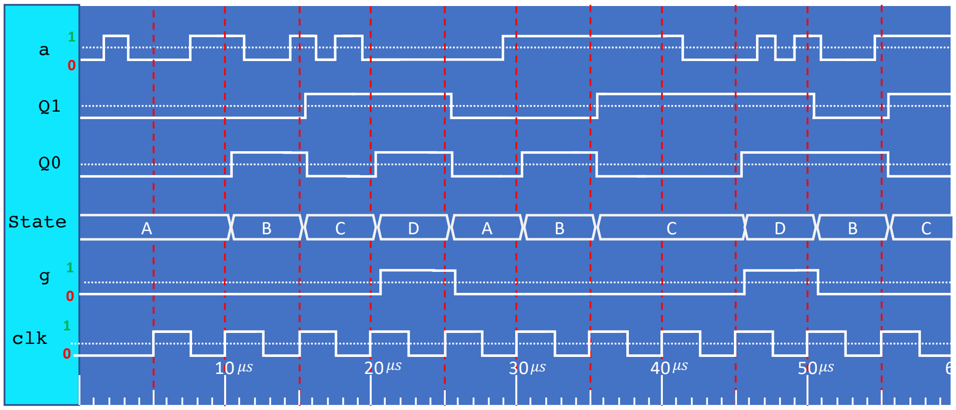


Figure 4.26: Simulation of analysis Example 1.

4.7.2 Example 2 sequential circuit analysis

Figure 4.27 is the logic circuit for a second sequential circuit analysis example. For this example, we obtain the following equations from its combinational logic:

$$\begin{aligned} y &= x\,\overline{Q}_1 \\ D_1 &= x\,Q_0 + \overline{x}\,Q_1 \\ D_0 &= Q_1\,Q_0 + \overline{x}\,Q_0 + x\,\overline{Q}_1\,\overline{Q}_0 \end{aligned}$$

In this example, the output `y`, depends on the input `x`, as well as one of the state bits. Two columns, one for each of the input values, are needed in the output section of the state transition table. The state diagram for Example 2 is shown in Figure 4.28. Here the value of the output

[8] Although the value of the state variables may remain unknown after the first clock edge, the number of possible values for the state variables will be reduced and as a result, their values would be known after the second clock edge (for this example). Most simulators do not provide the necessary level of detail to capture this result.

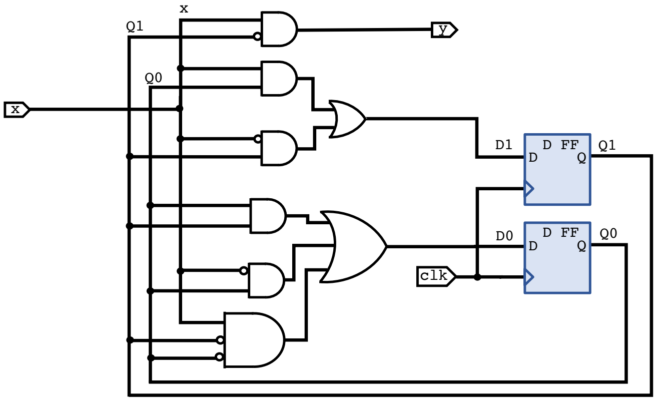


Figure 4.27: Example 2 of sequential circuit analysis.

(a)

	D_1 D_0		y	
Q_1Q_0	$x=0$	$x=1$	$x=0$	$x=1$
0 0	0 0	0 1	0	1
0 1	0 1	1 0	0	1
1 0	1 0	0 0	0	0
1 1	1 1	1 1	0	0

(b)

Q_1Q_0	State
0 0	A
0 1	B
1 0	C
1 1	D

(c)

Present	Next State		Output	
State	$x=0$	$x=1$	$x=0$	$x=1$
A	A	B	0	1
B	B	C	0	1
C	C	A	0	0
D	D	D	0	0

Table 4.3: (a) State transition table, (b) state encoding, and (c) Symbolic STT for Example 2.

depends on the value of the input as well as the state, so the output values are indicated on the arrows. There is an arrow from state X to state Y labeled $E/$`b` if the next state will be Y and output will be `b`, when the present state is X and the boolean expression E is 1. In this state diagram, state D is unreachable from the other states, and there is no way to leave state D. If the FFs are not initialized, it is possible that state D will be the initial state and hence the only state the circuit will visit. But it is also possible that the circuit could enter state D due to a timing issue as discussed in Section 4.8.

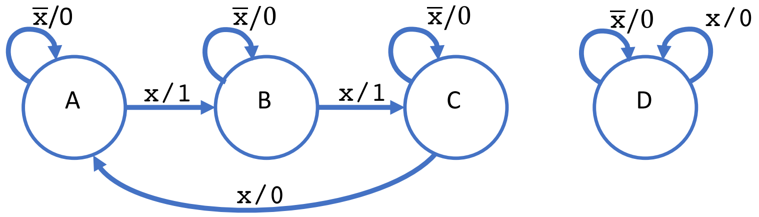


Figure 4.28: State diagram for analysis Example 2.

4.8 Synchronizing external inputs

The D FF component introduced in Section 4.2 came with the following warning regarding its D input:

> ⚠ **WARNING:**
> To correctly operate the D FF, the D input must be valid and stable for a specified small window of time before and after the rising edge of the clock input. Failure to meet this condition can result in unpredictable behavior of the output Q, even instability.

But even when the D FFs operate as expected, there could be an issue when there are multiple D FFs in the design and an external input changes at or near the clock edge. Figure 4.29 is a simulation of the circuit in Figure 4.27 from Example 2 (Section 4.7.2). At time 30 μs the input x transitions from 0 to 1. This occurs at or very near the clock edge. The simulation shows the circuit entering state D. But from the state diagram in Figure 4.28, it should not be possible to enter state D from any of the other states. Since there is no transition leaving state D, the circuit is now stuck in state D (unless there is another anomaly).

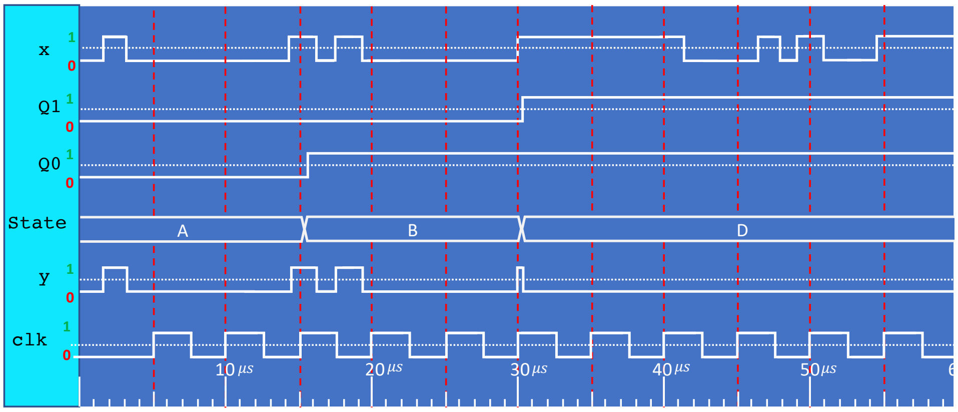
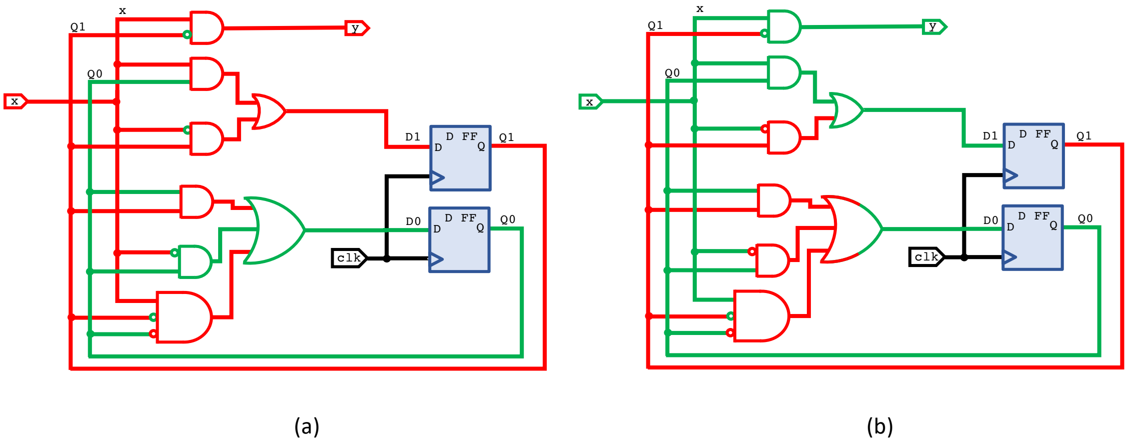


Figure 4.29: Simulation of Example 2 circuit with input transition on clock edge at 30μs.

The two snapshots of this circuit in Figure 4.30 show how state D was reached. The wires and gates are in red/green according to their logic value 0/1. In both snapshots, the circuit is in state B (Q1Q0 = 01). On the left in (a) x=0 and the logic has settled. On the right in (b) x=1 but not all of the logic has settled with their new values. D1 has transitioned from 0 to 1 but D0 is still 1. A clock edge at this point would load both FFs with 1 resulting in state D.

In general if an input changes near a clock edge, there is a possibility that some but not all of the combinational logic affected by this input will have settled to expected values before the next clock edge. Some of the gates will have transitioned to new values while others might take longer. An *asynchronous input* is a signal from a device that does not share your system's clock. This input can change value at any time, possibly near the clock edge. To avoid combinational logic using inconsistent input values, asynchronous inputs should affect only one path through the logic. The simplest way to ensure this is to connect an asynchronous input directly to one D FF and then use this D FF's output as the input to your circuit. This D FF is called a *synchronizer*. Its sole purpose



(a)

(b)

Figure 4.30: Snapshots of Example 2 circuit in state B: (a) with `x`=0 (b) just after `x` transitions to 1.

is to ensure that the rest of your circuit receives a new input value sufficiently early in the clock cycle. It does add a clock delay to the external input, but since this external input is asynchronous it was not required to be present at any specific clock edge.[9] When the asynchronous input changes near a clock edge the synchronizer may or may not capture the new value on that clock edge, but if not, it will capture the new value on the following clock edge. Using a synchronizer ensures that the combinational cloud has a full clock cycle to settle.

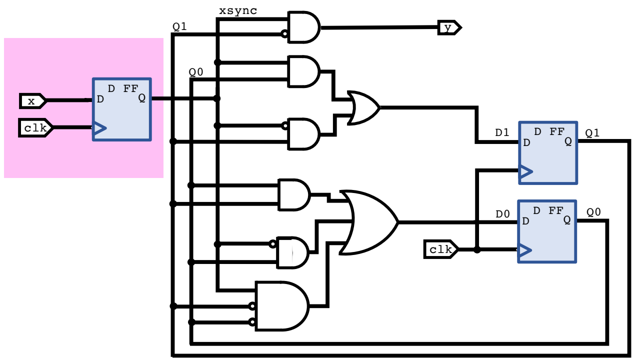


Figure 4.31: Synchronizer (shaded background) added to the external input of the Example 2 circuit.

[9] Adding FFs on inputs involved in feedback loops can cause problems.

4.9 Timing Constraints

When the D FF was introduced earlier in Section 4.2 it came with the following instruction:

> ⚠ **WARNING:**
> To correctly operate the D FF, the D input must be valid and stable for a specified small window of time before and after the rising edge of the clock input. Failure to meet this condition can result in unpredictable behavior of the output Q even possibly instability.

In this section, we shall see how the rules for synchronous sequential design from Section 4.2.1 will allow us to check this requirement. We will need upper and lower bounds on the delay of the combinational cloud of the synchronous sequential circuit as well as the D FF's timing information. The size of the window around the clock edge for which the D input to D FF must be stable is defined by two numbers, w_{SE} and w_H. In Figure 4.32 this window is in light blue. The width of the window before the clock edge is the *setup time* and the width after the clock edge is the *hold time*. These two values, w_{SE} and w_H, are properties of the FF.

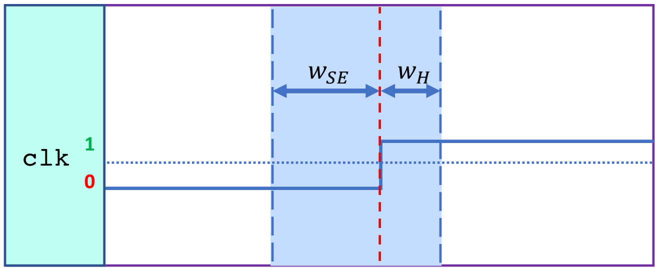


Figure 4.32: The D input of the D FF should be stable within the light blue regions.

To check whether a design meets the setup and hold time requirements of its FFs, the timing of changes on their D inputs need to be examined. The D inputs of the FFs come from the combinational cloud shown in Figure 4.33. Here we have sliced our model of synchronous sequential circuits through the FFs holding the state bits. Synchronizers have been added to the external inputs as discussed in Section 4.8.[10]

Using the method from Section 2.10 we can obtain bounds on when the outputs of the combinational cloud will settle after a change in its inputs. Suppose we know that its outputs will settle by time T_{MaxC}. The inputs of the combinational cloud are from the FFs, either the state bit FFs or synchronizers. Their logic values will change only in response to a rising clock edge after the *propagation* delay of the FF, $D(\texttt{D FF})$. If there is a rising clock edge at time 0, the values of the outputs of the combinational cloud will have settled by time $D(\texttt{D FF}) + T_{MaxC}$, and they will not change until after the next clock edge. The next clock edge will be at time $T_{\texttt{clk}}$ and we need the FF inputs to be stable w_{SE} before this clock edge. The inequality we need to satisfy is:

$$D(\texttt{D FF}) + T_{MaxC} \leq T_{\texttt{clk}} - \mathrm{w}_{SE}$$

[10] Synchronizers are used for asynchronous external inputs which (as the name implies) can change at any time. There is no way to ensure that they do not violate the setup/hold times of the D FFs used as synchronizers.

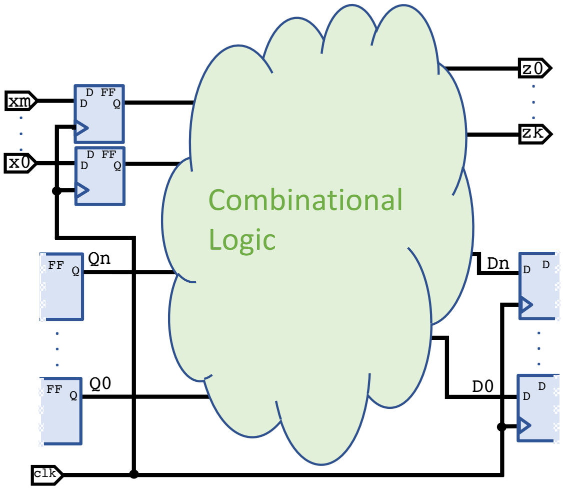


Figure 4.33: Sliced model of synchronous sequential circuits.

Figure 4.34 illustrates the setup time requirement. In Figure 4.34(a) the setup time constraint is met while in Figure 4.34(b) there could be a setup time violation. There are several ways to resolve a possible setup time violation:

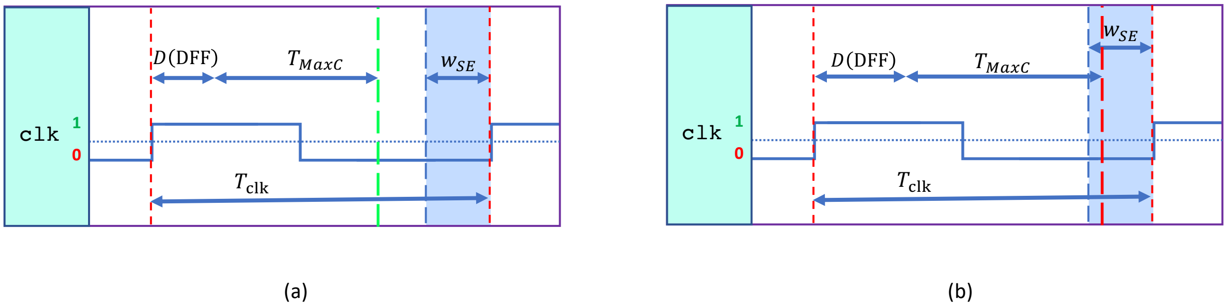


Figure 4.34: Setup time requirement: (a) satisfied, (b) violated.

- Increase the clock period ($T_{\texttt{clk}}$). Increasing the clock period will reduce the performance of the system and may not be an option if our design must operate at a certain speed.
- Use a different FF with smaller propagation delay (D(D FF)) or smaller setup time (w$_{SE}$).
- Redesign the circuit to reduce the delay of the combinational cloud. This might involve improving the combinational logic or even decomposing the design into separate Moore machines (e.g. pipelining).

To meet the hold time requirement we need to be sure that the inputs to FFs must also remain unchanged past the clock edge for the hold time. If the setup time is met, then the outputs of the combinational cloud have settled before the clock edge. They can only change again as a result of new values entering the combinational cloud after the clock edge. The earliest that new values entering the cloud could reach the FF inputs is $D(\texttt{D FF}) + T_{minC}$ where $D(\texttt{D FF})$ is the delay of the FF and T_{minC} is the minimum delay through the combinational cloud. T_{minC} could be very small as in a simple shift register where the output of one FF is directly connected to the input of another. To avoid a hold time violation we need this fastest delay to exceed the hold time. That is, we need

$$D(\texttt{D FF}) + T_{minC} \geq \mathrm{w}_H$$

Figure 4.35 illustrates the hold time requirement. In Figure 4.35(a) the hold time constraint is met while in Figure 4.35(b) there could be a hold time violation. It may seem surprising, but combinational logic that is too fast can be a problem for the FFs. The solutions for resolving a hold time violation include:

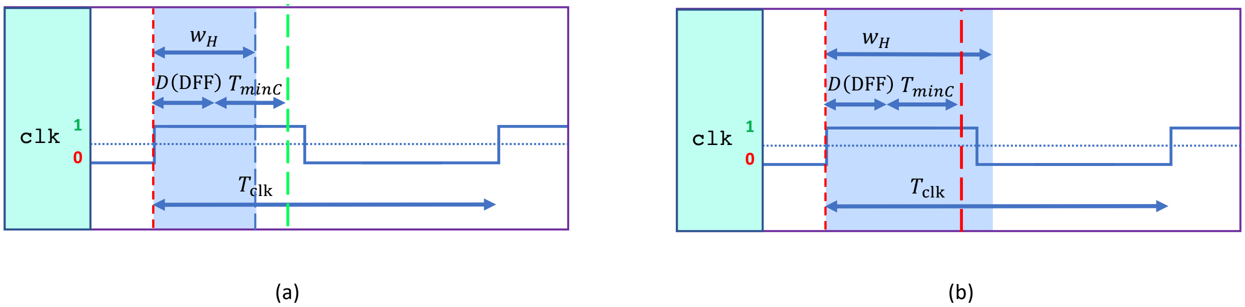


Figure 4.35: Hold time requirement: (a) satisfied, (b) violated.

- Use a different FF with a smaller hold time (w_H).
- Add delay (e.g. two inverters) to increase the delay for the fastest paths through the combinational cloud. Care should be taken to ensure that any delay added does not cause, possibly other paths, to exceed T_{MaxC}, and cause setup time violations.

Changing the clock speed will not resolve a hold time violation, but adding a delay may require increasing the clock period.

Clock skew is an additional consideration in meeting the setup and hold time requirements. The clock inputs of the FFs will not receive the clock edge at exactly the same time due to the physical properties of the clock net wiring. The maximum difference in time for the clock edge to reach the FFs is the *clock skew*. The setup/hold time window is anchored to the clock edge a FF receives. When the clock edge shifts, a hold or setup time violation may result since the FF's input may not shift by the same amount. Using the specialized wiring resources intended for clock distribution will minimize clock skew in FPGAs. Integrated circuit designers will carefully design clock distribution networks to minimize skew as well as achieve performance requirements and reduce power consumption.

Chapter 5

Synthesis of Synchronous Sequential Circuits

Chapter 4 introduced sequential circuits using a basic memory device: the D FF. Several circuits were introduced and a general method for analyzing these circuits was developed. Figure 5.1 (from Section 4.7) captures the structure of a synchronous sequential circuit. In this chapter, we will present the steps needed to obtain a synchronous sequential circuit for a given state diagram or other description of the circuit's desired behavior. This will be the reverse of the analysis process described in Section 4.7. We will design the combinational logic cloud, and decide how to encode the states, including the number of state bits (FFs) used.

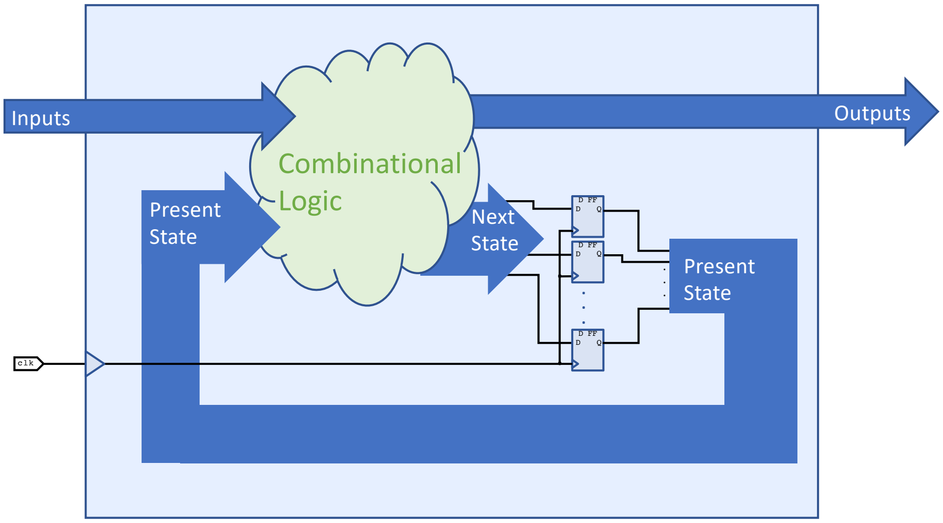


Figure 5.1: Model of synchronous sequential circuits.

The result of the analysis in Section 4.7 was a state diagram. State diagrams describe the behavior of *finite state machines*. Our goal will be to produce a logic diagram for a synchronous sequential circuit from a description of the desired behavior. When this description is a state diagram the process is straightforward, but otherwise, the first step will be to obtain a state diagram. Selecting the encoding of the states is one of the decisions to consider (unless it is specified).

The examples in this chapter are necessarily simple to keep their size small. We will begin with the simple binary counter for which the state diagram is clear-cut.

5.1 Counters revisited

In Section 4.5 a 4-bit binary counter was assembled based on equations capturing the effect of incrementing a bit vector. Here we will use a state diagram to represent the desired operation of a counter, but with 3-bits rather than 4-bits to keep the size manageable. The symbol for the 3-bit counter is shown in Figure 5.2(a) and its state diagram is on the right (Figure 5.2(b)). The 3-bit counter counts from 0 to 7, incrementing when its input `Inc` is 1. The output (indicated in each state below the line) is the value of the count and `TC`. Recall that `TC` should be 1 when the counter is at its maximum value: 7 for a 3-bit counter.

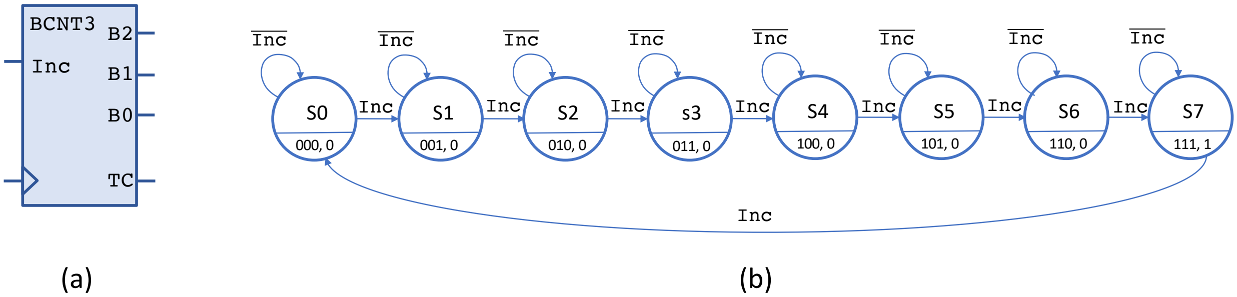


Figure 5.2: (a) Symbol and (b) state diagram for a 3-bit counter.

From the state diagram, we can obtain the Symbolic State Transition Table shown on the left below in Table 5.1(a). On the right one possible encoding for the states is given in Table 5.1(b). Each state has been encoded with the bit vector corresponding to its binary value. This encoding is convenient because the encoding of the state is also the value for the outputs `B2 B1 B0`. We have used 3 bits to encode 8 states which is a *minimum length encoding* since 3 is the smallest number of bits that can have 8 values. When there are N states at least $\log_2 N$ will be needed. The next step is to obtain the Encoded State Transition Table shown in Table 5.2 below. Table 5.2 is essentially a truth table. To obtain equations for D_2 D_1 D_0 and the four outputs we use Kmaps as described in Section 3.3. Since the values of `B2 B1 B0` are the same as the state encoding these outputs will come directly from the FF outputs.

$$\begin{aligned} D_2 &= Q_2\,\overline{Q}_1 + Q_2\,\overline{Q}_0 + Q_2\,\overline{\text{Inc}} + \overline{Q}_2\,Q_1\,Q_0\,\text{Inc} \\ D_1 &= Q_1\,\overline{Q}_0 + Q_1\,\overline{\text{Inc}} + \overline{Q}_1\,Q_0\,\text{Inc} \end{aligned}$$

Present	Next State		Output	
State	Inc = 0	Inc = 1	B2 B1 B0	TC
S0	S0	S1	0 0 0	0
S1	S1	S2	0 0 1	0
S2	S2	S3	0 1 0	0
S3	S3	S4	0 1 1	0
S4	S4	S5	1 0 0	0
S5	S5	S6	1 0 1	0
S6	S6	S7	1 1 0	0
S7	S7	S0	1 1 1	1

(a)

State	Q_2	Q_1	Q_0
S0	0	0	0
S1	0	0	1
S2	0	1	0
S3	0	1	1
S4	1	0	0
S5	1	0	1
S6	1	1	0
S7	1	1	1

(b)

Table 5.1: State encoding and Symbolic State Transition Table for the 3-bit binary counter.

			D_2 D_1 D_0						Outputs	
$Q_2Q_1Q_0$			Inc = 0			Inc = 1			B2B1B0	TC
0	0	0	0	0	0	0	0	1	0 0 0	0
0	0	1	0	0	1	0	1	0	0 0 1	0
0	1	0	0	1	0	0	1	1	0 1 0	0
0	1	1	0	1	1	1	0	0	0 1 1	0
1	0	0	1	0	0	1	0	1	1 0 0	0
1	0	1	1	0	1	1	1	0	1 0 1	0
1	1	0	1	1	0	1	1	1	1 1 0	0
1	1	1	1	1	1	0	0	0	1 1 1	1

Table 5.2: Encoded STT for the 3-bit binary counter.

$$\begin{aligned}
D_0 &= Q_0\,\overline{\text{Inc}} + \overline{Q}_0\,\text{Inc}\\
TC &= Q_2\,Q_1\,Q_0
\end{aligned}$$

These may appear different from the equations we obtained in Section 4.5. However, by regrouping their terms and using the identity $a \oplus b = a\overline{b} + \overline{a}b$ we see that they are the same.

$$\begin{aligned}
D_2 &= Q_2(\overline{Q}_1 + \overline{Q}_0 + \overline{\text{Inc}}) + \overline{Q}_2(Q_1\,Q_0\,\text{Inc})\\
&= Q_2\overline{(Q_1\,Q_0\,\text{Inc})} + \overline{Q}_2(Q_1\,Q_0\,\text{Inc})\\
&= Q_2 \oplus (Q_1\,Q_0\,\text{Inc})\\
D_1 &= Q_1(\overline{Q}_0 + \overline{\text{Inc}}) + \overline{Q}_1(Q_0\,\text{Inc})\\
&= Q_1\overline{(Q_0\,\text{Inc})} + \overline{Q}_1(Q_0\,\text{Inc})\\
&= Q_1 \oplus (Q_0\,\text{Inc})\\
D_0 &= Q_0 \oplus \text{Inc}\\
TC &= Q_2\,Q_1\,Q_0
\end{aligned}$$

Using these equations for the combinational logic we obtain the circuit in Figure 5.4. This is the same as the circuit in Figure 4.18 for the first three bits, but here the FFs are displayed vertically.

The FFs of the counter should be initialized to the desired starting value, usually 0. If they are not, then the count will start at an unknown value.

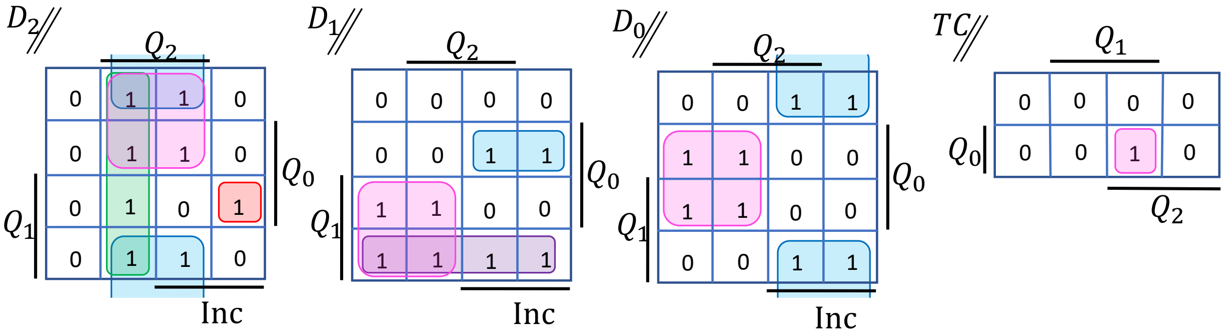


Figure 5.3: Kmaps for combinational logic of 3-bit counter.

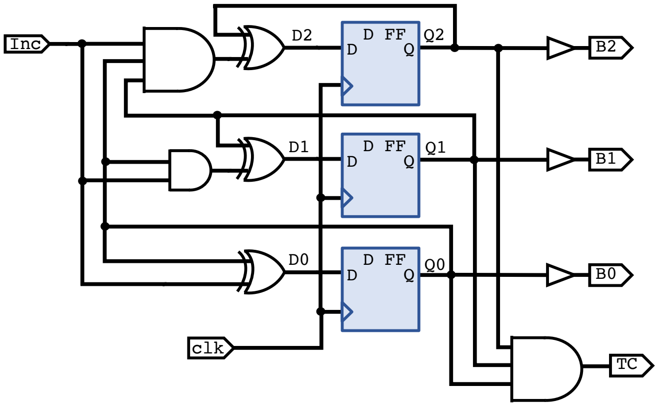


Figure 5.4: Logic diagram for 3-bit counter. The FFs are assumed to be initially 0.

5.1.1 Gray Code Counter

In an n-bit Gray code, the bit vectors are ordered so that only one bit will change between consecutive bit vectors. For example, 001 is followed by 011 rather than 010. Below is the sequence of the vectors in a 3-bit Gray code:

```
000 001 011 010 110 111 101 100
```

An n-bit Gray code can be obtained from the $(n-1)$-bit Gray code, by appending two copies of the $(n-1)$-bit Gray code with the second copy reversed, adding a leading 0 to the first half, and adding a leading 1 to the second half.

0000 **0**001 **0**011 **0**010 **0**110 **0**111 **0**101 **0**100 **1**100 **1**101 **1**111 **1**110 **1**010 **1**011 **1**001 **1**000

Gray codes are useful when two or more outputs must change value, and we need to ensure that the output values are never incorrect. For example, when a binary counter increments from 1 (01) to 2 (10), for a brief moment the two output bits are either 00 or 11 depending on which of the two bits changes first. The outputs should never be incorrect when they are asynchronous inputs to another device since they could be sampled at any time. It is not possible to guarantee that two outputs will transition to new values at the same exact time and so for a short time, the output values will be incorrect.[1] By using a Gray counter there is no possibility for the outputs to be incorrect during any transition since only one output bit is changing when it increments. Two approaches to the implementation of a 3-bit Gray Code counter are shown in Figure 5.5. In Figure 5.5(a) combinational logic is used to re-encode the output of a 3-bit binary counter to its corresponding Gray code. Instead, in Figure 5.5(b) the Gray code is used to encode the states to achieve the goal of ensuring that the `G2 G1 G0` outputs are never incorrect since they are outputs of the FFs, and only one of them changes in each increment. Approach (a) does not provide this assurance.

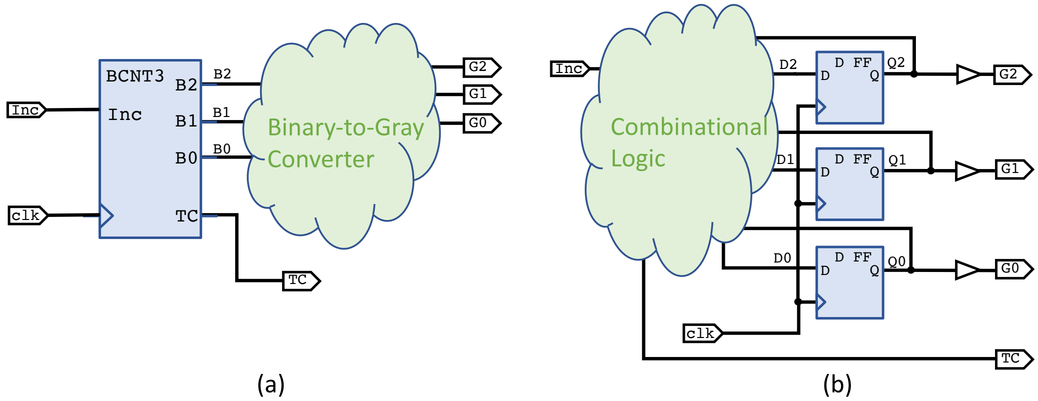


Figure 5.5: Two Gray code counter implementations: (a) encoding the output of a binary counter; and (b) using Gray Code for the state encoding of a 3-bit counter.

To implement the Gray Code counter in Figure 5.5(b) we use the encoding for the states given in Table 5.3(a) to obtain the encoded state transition table in Table 5.3(b).

State	Q_2	Q_1	Q_0
S0	0	0	0
S1	0	0	1
S2	0	1	1
S3	0	1	0
S4	1	1	0
S5	1	1	1
S6	1	0	1
S7	1	0	0

(a)

		$D_2\ D_1\ D_0$		Outputs	
State	$Q_2Q_1Q_0$	Inc = 0	Inc = 1	G2G1G0	TC
S0	0 0 0	0 0 0	0 0 1	0 0 0	0
S1	0 0 1	0 0 1	0 1 1	0 0 1	0
S3	0 1 0	0 1 0	1 1 0	0 1 0	0
S2	0 1 1	0 1 1	0 1 0	0 1 1	0
S7	1 0 0	1 0 0	0 0 0	1 0 0	1
S6	1 0 1	1 0 1	1 0 0	1 0 1	0
S4	1 1 0	1 1 0	1 1 1	1 1 0	0
S5	1 1 1	1 1 1	1 0 1	1 1 1	0

(b)

Table 5.3: 3-bit Gray counter (a) Gray Code state encoding and (b) encoded state transition table.

[1] With a great deal of engineering effort taking into account delays of wires and components, this time can be reduced, but never completely eliminated.

5.2 Even Blocks Machine

In this section, our task is to produce a circuit given the following description.

> Design a sequential circuit with one synchronous input and one synchronous output. The output should be 1 for one clock cycle after an even length block of 1's followed by a 0 is observed on the input.

Since the input/output is synchronous, this circuit will have a clock input in addition to one input and one output. So the symbol for our circuit will be as shown in Figure 5.6.

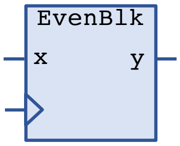


Figure 5.6: Symbol for the Even Blocks machine.

A "block of 1's" is a group of consecutive 1's that cannot be made longer. This means that the block must start with the first input or there must be a 0 input before the block of 1's.[2] The steps below explain how a state diagram might be constructed. These steps are shown in Figure 5.7.

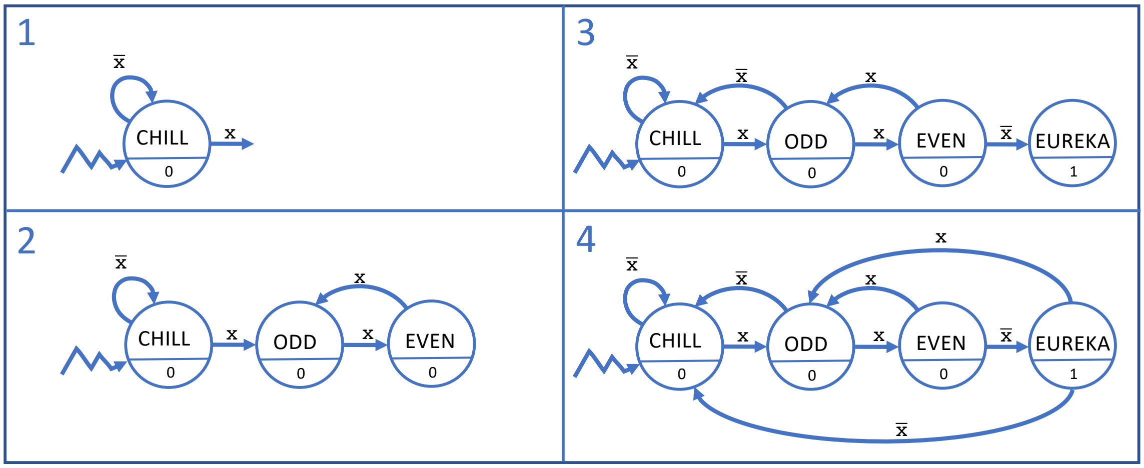


Figure 5.7: Construction of state diagram for the Even Blocks machine.

1. Initially the machine is in state `CHILL` and this is indicated by the jagged arrow into `CHILL`. As long as the input `x` is 0 the machine will remain in `CHILL` awaiting the start of a block of 1's. When `x` is 1 the machine will leave `CHILL` since a block of 1's has begun. The output should be 0.

[2] The description is vague on this point. In practice, it might be wise to verify whether the block must have a 0 before it.

2. After leaving CHILL the machine keeps track of whether the number of 1's it has seen so far is odd or even. It will alternate between ODD and EVEN while x is 1. The output should still be 0.

3. If x is 0 in state ODD then we have an odd length block of 1's followed by a 0 and the machine should return to CHILL. But if x is 0 in state EVEN then we have an even length block of 1's followed by a 0 and the machine should enter a state where it will output 1: EUREKA.

4. To complete the state machine we need to indicate what happens in state EUREKA. The output is 1 in state EUREKA and the machine should leave this state on the next clock edge since the output should only be 1 for one clock cycle. If x is 0 the machine returns to CHILL awaiting the start of the next block of 1's. But if x is 1 then this is the start of a new block of 1's so the the machine returns to ODD.

From the state diagram, we obtain the state transition table:

Present	Next State		
State	x = 0	x = 1	Output
CHILL	CHILL	ODD	0
ODD	CHILL	EVEN	0
EVEN	EUREKA	ODD	0
EUREKA	CHILL	ODD	1

Table 5.4: Symbolic state transition table for Even Blocks machine.

Below is a state encoding of minimum length and the resulting encoded state transition table is shown.

State	Q_1Q_0
CHILL	0 0
ODD	0 1
EVEN	1 0
EUREKA	1 1

(a)

		D_1 D_0		Output
State	Q_1Q_0	x = 0	x = 1	y
CHILL	0 0	0 0	0 1	0
ODD	0 1	0 0	1 0	0
EVEN	1 0	1 1	0 1	0
EUREKA	1 1	0 0	0 1	1

(b)

Table 5.5: Even Blocks implementation: (a) state encoding and (b) encoded state transition table.

The next step is to obtain logic equations for the next state variables, D1 and D0, and the output y.

$$\begin{aligned} D_1 &= x\,\overline{Q}_1\,Q_0 + \overline{x}\,Q_1\,\overline{Q}_0 \\ D_0 &= x\,Q_1 + Q_1\,\overline{Q}_0 + x\,\overline{Q}_0 \\ y &= Q_1\,Q_0 \end{aligned}$$

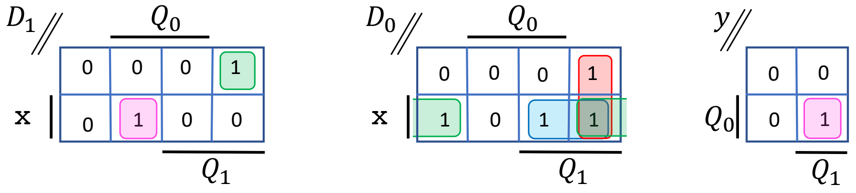


Figure 5.8: Kmaps for combinational logic of Even Blocks machine.

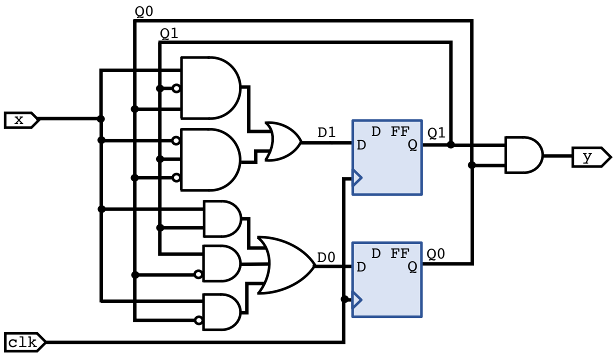


Figure 5.9: Logic diagram for Even Blocks machine. The FFs are assumed to be initially 0.

With these equations for the combinational logic, we obtain the circuit in Figure 5.9. The FFs should be initialized to 0 so that the machine wakes up in the initial state, CHILL.

A simulation of this circuit is shown in Figure 5.10. This design is a Moore machine. Its output value depends only on the present state (the FF values). As a consequence, the output y will be 1 only after x is 0 on the clock edge following an even block of 1's. The 1 output will be on the clock edge after the 0 input, not the same clock edge.

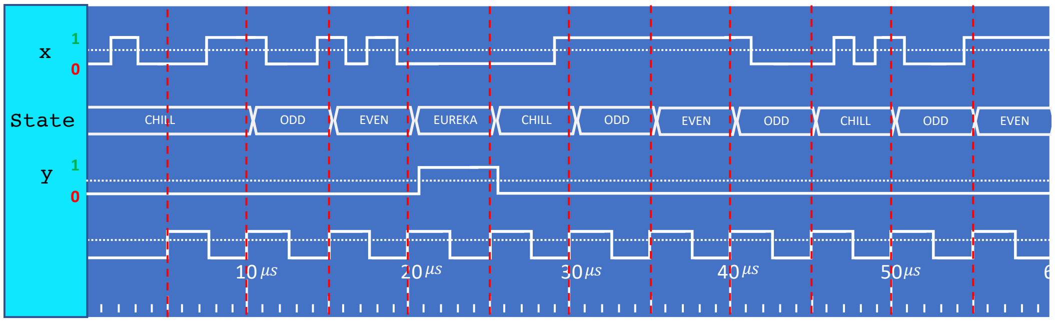


Figure 5.10: Simulation of Even Blocks machine.

5.2.1 Even Blocks Machine Mealy Version

The Even Blocks machine above had a Moore output. Suppose that instead, the circuit should respond as soon as `x` is 0 rather than after the clock edge. This is a different machine. Its state diagram is shown in Figure 5.11. The values for the output `y` are shown on the arcs(transitions) since the value of the output now depends on the value of the input (in state `EVEN2`). The main difference here is that in state `EVEN2` the output is 1 if `x` is 0. There is no need for the `EUREKA` state: in state `EVEN2` the machine transition back to `CHILL2` if `x` is 0.

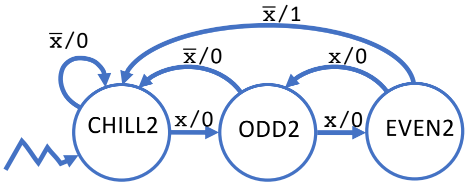


Figure 5.11: State diagram for Mealy version of the Even Blocks machine.

From the state diagram, we obtain the state transition table:

Present	Next State		Output	
State	x = 0	x = 1	x = 0	x = 1
CHILL2	CHILL2	ODD2	0	0
ODD2	CHILL2	EVEN2	0	0
EVEN2	CHILL2	ODD2	1	0

Table 5.6: Symbolic state transition table for Mealy version of the Even Blocks machine.

Below is a state encoding of minimum length, and the resulting encoded state transition table is shown. Since this machine has three states, a minimum length encoding will require two state bits `Q1 Q0`, and so one of the four possible values will not be used. Since the unused value for the state bits does not occur, the values for the next state and output can be indicated as don't care (?).

State	Q_1Q_0
CHILL2	0 0
ODD2	0 1
EVEN2	1 0

(a)

		D_1 D_0		y	
State	Q_1Q_0	x = 0	x = 1	x = 0	x = 1
CHILL2	0 0	0 0	0 1	0	0
ODD2	0 1	0 0	1 0	0	0
EVEN2	1 0	0 0	0 1	1	0
unused	1 1	? ?	? ?	?	?

(b)

Table 5.7: Even Blocks mealy machine implementation: (a) state encoding and (b) encoded state transition table.

The next step is to obtain logic equations for the next state variables, `D1` and `D0`, and the output `y`.

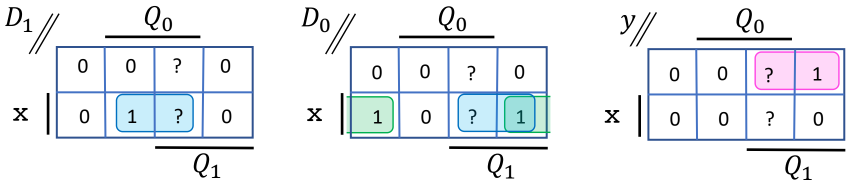


Figure 5.12: Kmaps for combinational logic of Even Blocks Mealy machine.

$$
\begin{aligned}
D_1 &= x\,Q_0 \\
D_0 &= x\,\overline{Q}_0 \\
y &= \overline{x}\,Q_1
\end{aligned}
$$

With these equations for the combinational logic, we obtain the circuit in Figure 5.9.

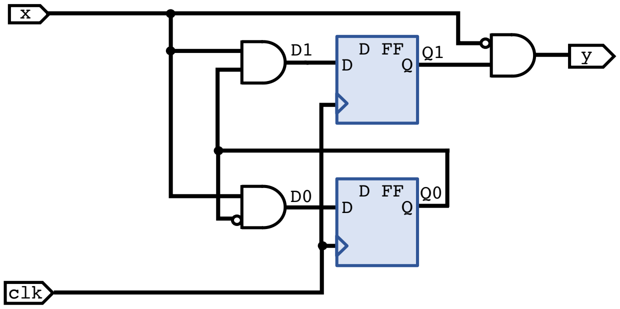


Figure 5.13: Logic diagram for Even Blocks Mealy machine. The FFs are assumed to be initially 0.

The simulation of this circuit on the same input as the Moore version is shown in Figure 5.14 where both the Moore and Mealy state and output are shown. Note that the output `y2` is 1 at the 20μs clock edge, one cycle earlier than in the Moore version `y` (25μs clock edge). In the Mealy version, the output is also 1 in between clock edges such as at 16μs. In addition, the output will not be 1 for a full clock cycle. In Section 5.6 the trade-offs between Mealy and Moore machines explored. But when the specification requires the output to react to the current input on the same clock edge a Mealy machine is needed. Table 5.8 shows the inputs and outputs of both machines at clock edges.

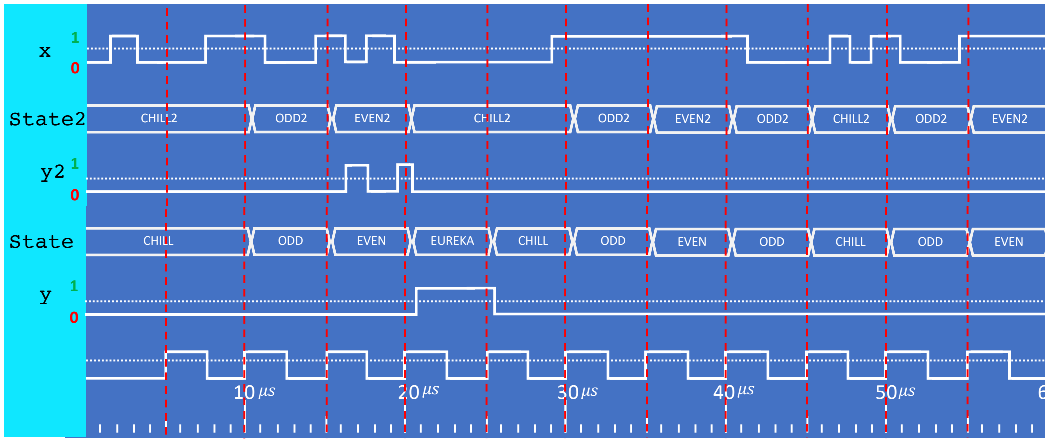


Figure 5.14: Simulation of both Even Blocks machines. Mealy version state and output are labeled `State2` and `y2`.

	t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	x	0	1	1	0	0	1	1	1	0	1	1	1	1	0	0	0	1	0	1	1
Moore	y	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
Mealy	y2	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

Table 5.8: Comparison of input/output of Moore and Mealy versions of the Even Blocks machines.

5.3 Message Checker

In serial communication, messages are transmitted one bit at a time on a single wire rather than on multiple wires in parallel. Logic circuits are needed to pack and unpack these messages. Here we will consider a small example.

Design a sequential circuit with two synchronous inputs, `s` and `d`, and two synchronous outputs, `e` and `p`. This device will receive a message, one bit at a time on the clock edge.

- The value of the bit sent will be on the input `d`.
- The input `s` will mark the beginning and end of the message, since there may be several clock edges between messages. Specifically, `s` will be 1 at the same clock edge as the first bit of the message and 1 at the clock edge with the last bit of the message.
- The output `e` should identify the end bit of a message: `e` should be 1 on the same edge that `s` is 1 but only at the end of messages, not the start.
- The output `p` is 1 at the end of a message if the number of 1 bits in the message is 2 plus a multiple of 3 (aka 2 (mod 3)).

Since the inputs are synchronous, the circuit will have a clock input in addition to `s` and `d`. The symbol for the circuit is shown in Figure 5.15. Below is a sample input/output sequence. Since the outputs are 1 on the same clock edge as the last bit of the message, they will both be Mealy.

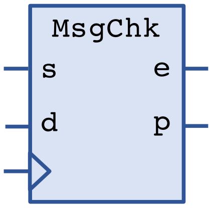


Figure 5.15: Symbol for the Message Checker.

t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
s	0	1	0	0	0	1	0	1	1	0	1	0	1	0	0	1	0	0	0	1
d	0	1	1	0	0	1	1	1	1	0	1	0	1	1	0	0	0	1	0	0
e	0	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	1
p	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0

Table 5.9: Sample input/output of Message Checker machine.

The steps below explain how a state diagram might be constructed. These steps are illustrated in Figure 5.16.

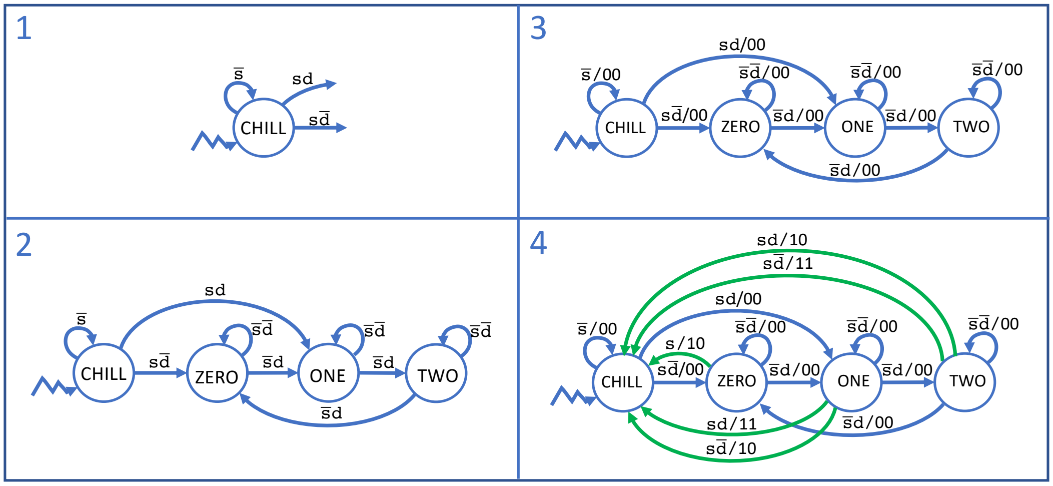


Figure 5.16: Construction of state diagram for the Message Checker machine.

1. Initially the machine is in state CHILL and this is indicated by the jagged arrow into CHILL. As long as the input s is 0 the machine will remain in CHILL awaiting the start of a message. When s is 1 the machine will leave CHILL since a message has begun. The next state when s is 1 will depend on the value of d so there are separate arcs.

2. At the start of a message (in CHILL with s=1) the machine keeps track of the number of 1's in the message. Much like a counter, the machine cycles between ZERO, ONE, and TWO when d is 1 but stays put when d is 0. This continues while s remains 0 since the message has not

ended.

3. Pairs of values for the outputs (`ep`) have been added to the arcs. Both `e` and `p` will be 0 until the last bit of the message.

4. To complete the state machine we add the arcs (in green) for `s`=1 from `ZERO`, `ONE`, and `TWO` to `CHILL` since this is the end of the message. In `ZERO` the number of 1's in the message cannot be 2 (mod 3) regardless of the value of `d` so `p` will be 0 in both cases. In states `ONE`, and `TWO` the value of `p` will depend on `d` so there are two separate arcs.

From the state diagram, we obtain the state transition table:

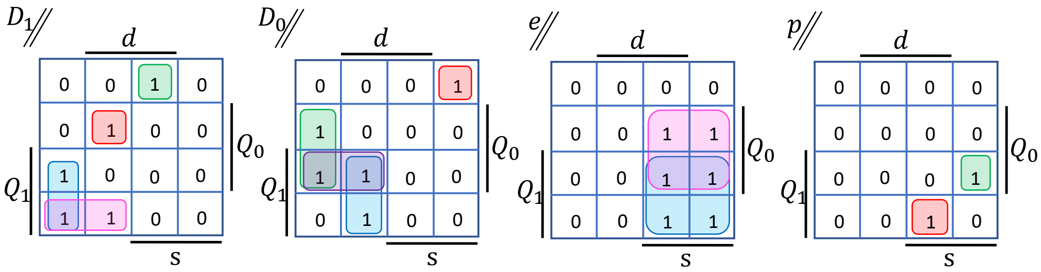
Present	Next State				Output			
State	`sd` = 00	`sd` = 01	`sd` = 10	`sd` = 11	`sd` = 00	`sd` = 01	`sd` = 10	`sd` = 11
`CHILL`	`CHILL`	`CHILL`	`ZERO`	`ONE`	00	00	00	00
`ZERO`	`ZERO`	`ONE`	`CHILL`	`CHILL`	00	00	10	10
`ONE`	`ONE`	`TWO`	`CHILL`	`CHILL`	00	00	10	11
`TWO`	`TWO`	`ZERO`	`CHILL`	`CHILL`	00	00	11	10

Table 5.10: Symbolic state transition table for Message Checker machine.

Below a state encoding of minimum length is used for the states to produce an encoded state transition table.

		`sd`=	D_1D_0			`sd`=	Output		
State	Q_1Q_0	00	01	10	11	00	01	10	11
`CHILL`	0 0	00	00	01	10	00	00	00	00
`ZERO`	0 1	01	10	00	00	00	00	10	10
`ONE`	1 0	10	11	00	00	00	00	10	11
`TWO`	1 1	11	01	00	00	00	00	11	10

Table 5.11: Message Checker encoded state transition table.



0	0	1	0
0	1	0	0
1	0	0	0
1	1	0	0

0	0	0	1
1	0	0	0
1	1	0	0
0	1	0	0

0	0	0	0
0	0	1	1
0	0	1	1
0	0	1	1

0	0	0	0
0	0	0	0
0	0	0	1
0	0	1	0

Figure 5.17: Kmaps for combinational logic of Message Checker machine.

The Kmaps for the next state variables, `D1` and `D0`, and the outputs `e` and `p` are in Figure 5.17 followed by the logic equations.

$$
\begin{aligned}
D_1 &= s\,d\,\overline{Q}_1\,\overline{Q}_0 + \overline{s}\,d\,\overline{Q}_1\,Q_0 + \overline{s}\,\overline{d}\,Q_1 + \overline{s}\,Q_1\,\overline{Q}_0 \\
D_0 &= s\,\overline{d}\,\overline{Q}_1\,\overline{Q}_0 + \overline{s}\,d\,Q_1 + \overline{s}\,\overline{d}\,Q_0 \\
e &= s\,Q_0 + s\,Q_1 \\
p &= s\,d\,Q_1\,\overline{Q}_0 + s\,\overline{d}\,Q_1\,Q_0
\end{aligned}
$$

The FFs should be initialized to 0 so that the machine wakes up in the initial state, CHILL.

5.4 One-hot State Encoding

In the previous sections, the state encodings were minimum-length encodings. The smallest number of bits possible was used to minimize the number of FFs. When FPGAs are used to implement the design, minimizing the number of FFs may be less efficient. This is because the resources (chip area) have already been pre-allocated for the FFs and combinational logic. Using more FFs can result in smaller output and next state equations providing better performance. In a *one-hot* state encoding there is one state bit for each state, and that bit is 1 only for its associated state. A key advantage of one-hot encoding is the simple correspondence between the state diagram (or table) and the next state logic equations.

Figure 5.18 illustrates the correspondence between the state diagram and the logic for the next state equations. The states are labeled with their state bits. That is, the machine is in the state labeled Q_{23} exactly when variable Q23 is 1 and all the other state variables are 0. There are three transitions into Q_{23}. The next state will be Q_{23} when one of these transitions occurs. The variable D23 should be 1 when the next state will be Q_{23}. The transition from state Q_{13} will occur if cd is 1 and Q13 is 1 (the machine is in state Q_{13}). Combining the logic for the three transitions into Q_{13} gives the equation for D_{23}:

$$D_{23} = (a+b)\,Q_{23} + c\,d\,Q_{13} + (a+c)\,\overline{d}\,Q_7.$$

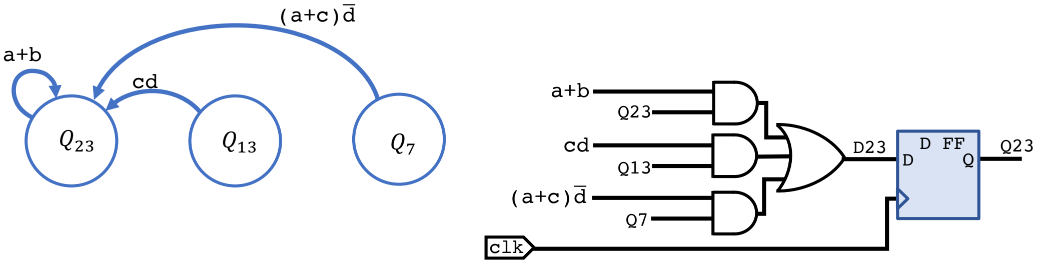


Figure 5.18: One-hot state encoding: from state diagram to logic.

In one-hot encoding, the FF corresponding to the initial state of the machine should be initialized to 1.

⚠**WARNING:**
If all of the FFs initialize to 0, then all of the next state equations will evaluate to 0 and this will continue resulting in a very boring machine. The FF corresponding to the initial state should initialize to 1.

The Even Blocks machine implemented with one-hot encoding is shown in Figure 5.19. The state bit associated with each state is shown in the state diagram on the left and the logic diagram is on the right. Note that inverters are added to the FF for the initial state so that the machine will wake up in state `CHILL`. The equations

$$\begin{aligned}
D_3 &= \overline{x}\,Q_2 \\
D_2 &= x\,Q_1 \\
D_1 &= x\,Q_0 + x\,Q_2 + x\,Q_3 = x(Q_0 + Q_2 + Q_3) \\
D_0 &= \overline{x}\,Q_0 + \overline{x}\,Q_1 + \overline{x}\,Q_3 = \overline{x}(Q_0 + Q_1 + Q_3) \\
y &= Q_3
\end{aligned}$$

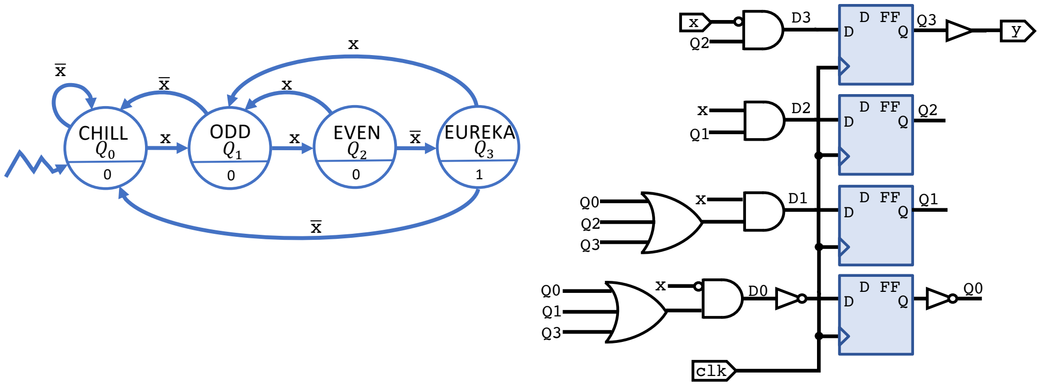


Figure 5.19: One-hot implementation of the Even Blocks machine. The FFs are initialized to 0.

5.5 Well-defined State Machines

Constructing the state diagram (or the equivalent symbolic state transition table) is the step that requires the most thought. Several modifications to the state diagram may be needed, or even a complete redesign from scratch may be a good idea. Identifying the inputs/outputs of the state machine should be the first task. Once they are known, two rules must be met to ensure that the state machine is well defined.

Rule 1 : mutually exclusive transitions For each state S, the conditions on the arcs leaving S should be mutually exclusive. It should not be possible for more than one condition to evaluate to 1.

Rule 2 : exhaustive transitions For each state S, the condition on at least one arc leaving S must be possible. It should not be possible for all of the conditions to simultaneously evaluate to 0.

Rule 1 prevents multiple states from simultaneously being possible as the next state. If two arcs can be 1 simultaneously then there are two possible next states. In this case, the output of the next state combinational logic may even be a different or unused state value. For example, if one-hot state encoding is used, then there will be two state bits that are 1. Verifying this rule can become complicated as the number of inputs increases. Suppose we have a state S with the arcs as shown in Figures 5.20(a) and (b). The color of the cells of the Kmaps correspond to the conditions on the arcs out of state S that they satisfy. In Figure 5.20(a) the cell where a=0, b=1, and c=0 is both red and green since it satisfies both the red and green conditions. In Figure 5.20(b) the green condition was changed so that it no longer includes this case and now there is no overlap in the Kmap. The desired behavior of the state machine should determine how the overlap is resolved.

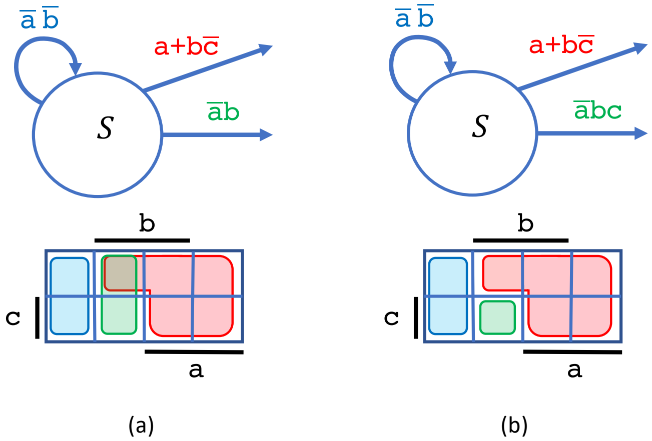


Figure 5.20: Rule 1 mutually exclusive transitions: (a) rule not satisfied, (b) rule satisfied.

Rule 2 ensures that the next state is spelled out in all cases. If a value for the inputs is not covered by any of the conditions, then there is no next state defined when that value occurs. If this happens the combinational logic may provide an unexpected value for the next state. When one-hot state encoding is used, all of the state bits will be 0 and they will remain that way until the state machine FFs are re-initialized. Again, verifying this rule becomes complicated as the number of inputs increases. Suppose we have a state S with the arcs as shown in Figures 5.21(a) and (b). The colors of the cells of the Kmaps correspond to the color of the arc condition they satisfy. In Figure 5.21(a) the cell where a=1, b=0, and c=1 does not satisfy any condition. If this input value occurs the next state will not be defined. This has been fixed in Figure 5.21(b) by changing the red condition so that it now covers this case.

In some cases, there may be input combinations that are not expected to occur. For example, a counter may have two inputs `Inc` and `Dec` that indicate that an increment/decrement of the value should occur at a clock edge. If both inputs are 0, the counter holds its value and if one of the two inputs is 1, then the requested change occurs. But if both inputs are 1 simultaneously and no provision has been made for this case, the value of the counter assumes may be quite unexpected. Both `Inc` and `Dec` could be 1 as a result of a logic error in the surrounding circuit or simply a misconception of how the counter will behave in this case. The design of the counter should take

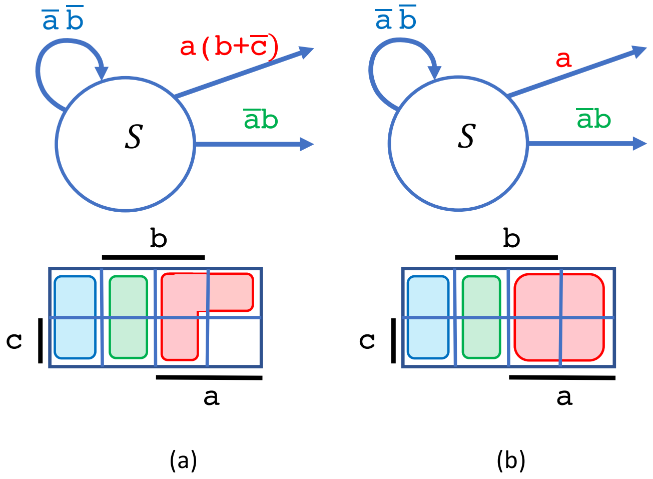


Figure 5.21: Rule 2 exhaustive transitions: (a) rule not satisfied, (b) rule satisfied.

into consideration the possibility that these inputs could both be 1 and ensure that the outcome (its value) in this case is not completely unexpected. Options might include not changing the value or deciding that one of the two inputs, `Inc` or `Dec`, takes priority over the other. In general, even when an input value should not occur, it is best to define a next state for it that is friendly. With one-hot encoding, if no next state is defined for a particular input value, then all of the state bits will be 0 if this input value occurs.

5.6 Mealy versus Moore

Earlier in this chapter two versions of the Even Blocks Machine were implemented. The first was a Moore version with 4 states and the second a Mealy with 3 states. The outputs of these two versions will not always agree. In general, the requirements for the design will determine whether an output should react to the current input or only past inputs.

When designing state machines as part of a larger sequential system (as in Chapter 6), there may be a choice as to whether an output of a state machine is Mealy or Moore. The advantages/disadvantages are discussed here.

Fewer states Delaying the output till after the clock edge will often require an additional state. For example, the Moore version of the Even Blocks Machine needed a separate state, `EUREKA`, in which the output `y` was 1. A Mealy output can always be changed to a Moore output by passing it through a FF. However, this might affect the timing of the interaction of the machine with other components, so it is best to design the machine with the output as a Moore output directly.[3]

Shorter logic paths A digital system composed of multiple state machines may have paths of

[3] There in an example in Section 6.5.

combinational logic that cross between state machine boundaries. Since a Mealy output depends on the input to the machine, there is a path of combinational logic from an input to that output. When two Mealy machines are connected as in the top half of Figure 5.22 there is a combinational logic path through both machines from an input `xi` to some output `zj`. This is not the case with Moore machines since the outputs will only depend on the FF outputs. In the bottom half of Figure 5.22, two Moore machines are connected, but there is no combinational path from any of the inputs `xi` to any output `zj`. In effect, using several

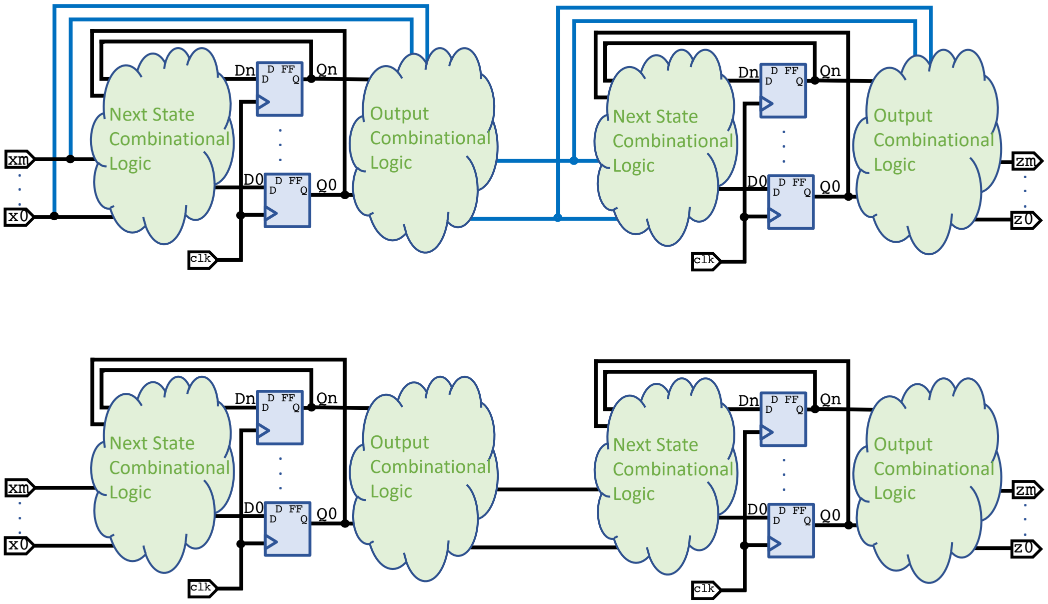


Figure 5.22: Combinational logic paths created by connecting Mealy versus Moore machines.

smaller Moore machines is akin to pipelining. The task is divided into stages, each with a short delay, The clock can then be faster, but more clock cycles will be required to produce the final output.

Avoiding combinational loops As noted composing Mealy machines can create longer combinational paths. There is also the possibility of inadvertently creating a *combinational loop*. In Figure 5.23 the blue wires could be part of a combinational loop. Depending on the type

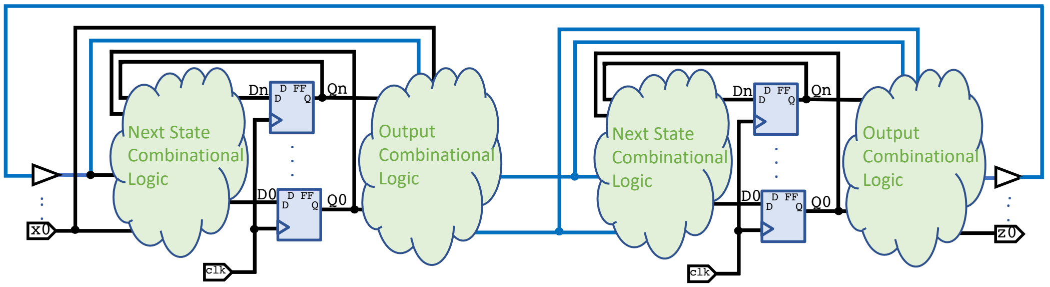


Figure 5.23: Connecting Mealy machines may create combinational loops.

of simulation, a combinational loop may cause the simulator to stall since the evaluation of the logic values of nets in the loop will never settle. If the simulation takes into account gate delays, then the outputs of the gates on the loop will appear to be oscillating.

5.7 State Minimization

An edge detector is a circuit that will recognize a rising edge on its input. Its output will be 1 for one clock cycle when the inputs at the last two clock edges were a 0 followed by a 1. This component can be implemented with a 2-bit shift register as shown in Figure 5.24 and AND to recognize the pattern 10. Its state transition table is in Table 5.12.

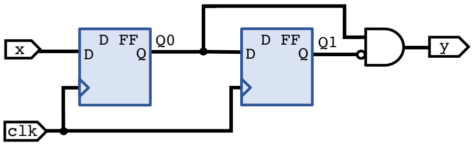


Figure 5.24: Edge Detector logic diagram .

	D_0D_1		Output
Q_0Q_1	x = 0	x = 1	y
0 0	0 0	1 0	0
0 1	0 0	1 0	0
1 0	0 1	1 1	1
1 1	0 1	1 1	0

Table 5.12: State transition table of the Edge Detector.

The state diagram for this Edge Detector is in Figure 5.25(a) below. The shaded states have the same output and their transitions for either input value take them to the same state. This can also be seen from the state transition table in Table 5.24 where the rows for these two states, 00 and 01, have the same next states and output. In Figure 5.25(b), the states 00 and 01 have been merged into one state, 0- reducing the number of states from four to three.

A sequence of input values (possibly empty) that will result in different sequences of output values when the machine is started in state A versus state B is said to *distinguish* states A and B. Two states are *distinguishable* if there is a sequence that can distinguish between them, and otherwise they are *indistinguishable* or *equivalent*. For example in the Edge Detector, 10 is distinguishable from the other states since it is the only state with an output value of 1. States 00 and 11 are distinguishable since when the input is 1, their next states will be 10 and 11 which have different output. However, in the unreduced Edge Detector, states 10 and 00 are indistinguishable. This was established by observing that their output and next states were the same. Unfortunately, this method is not enough to identify all equivalent states. Figure 5.26 contains a third state diagram for yet another Edge Detector and Table 5.26 contains the state transition table.

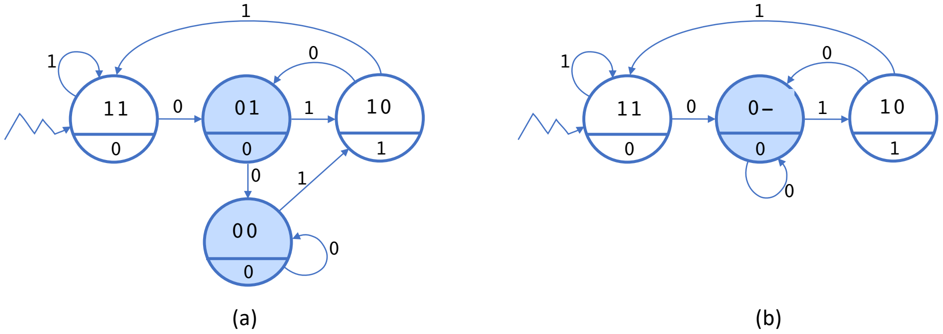


Figure 5.25: Edge detector state diagrams: (a) original and (b) reduced.

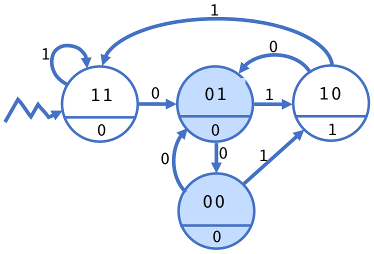


Figure 5.26: Yet another Edge Detector state diagram.

	D_0D_1		Output
Q_0Q_1	x = 0	x = 1	y
0 0	0 1	1 0	0
0 1	0 0	1 0	0
1 0	0 1	1 1	1
1 1	0 1	1 1	0

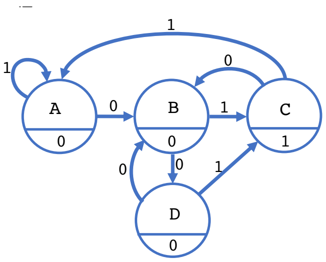
Table 5.13: State transition table of yet another Edge Detector.

This version differs from the shift register version only in that the transition from 00 with input 0 is to state 01 rather than to state 00. States 00 and 01 are still indistinguishable, even though their next states are not the same (though their next states are equivalent).

Moore's method for identifying equivalent states is outlined below.

1. Group states by output.
2. For each group and each state in that group
   Generate a label consisting of the group labels of the next states for all input values.
3. For each group
   Subdivide each group by the generated labels
4. If at least one new group was created repeat step 2

In this method, states are first partitioned into groups based on their output. At any time, states that are in separate groups are known to be distinguishable. The algorithm proceeds by examining states that are still potentially equivalent: states in the same group. If two states in the same group have next states that are already known to be distinguishable, then these two states become distinguishable and they are moved to separate groups. This process repeats until no new groups are formed.



```
                 1             2
             ( A B D )       ( C )   Step 1. Group states by output
x=0            1 1 1           1     Step 2. Generate next-group labels
x=1            1 2 2           1

          1         2          3
        ( A )    ( B D )     ( C )   Step 3: Sub-divide groups
x=0       2        2 2         2     Step 2. Generate next-group labels
x=1       1        3 3         1

          1         2          3
        ( A )    ( B D )     ( C )   Step 3: Sub-divide groups
No change: Done
```

Figure 5.27: Moore's algorithm applied to yet another Edge Detector.

In Figure 5.27, Moore's method is applied to the state machine from Figure 5.26. The states are divided into two groups based on their output. States `A`, `B`, and `D` have outputs 1, while state `C` has output 1. The two groups are labeled using script numbers (any symbol would do). The states are then labeled with the groups of their next states. For example, state `A` is labeled with 1 1 because its next states are `A` and `B` which are in group 1. State `B` is labeled with 1 2 because its next states are `D` and `C` which are in groups 1 and 2. After labeling all 4 states, group 1 is split into two groups since state `A` has a different label from `B` and `D`. States `B` and `D` remain together since they have the same label. State `C` is already by itself and hence is distinguishable from all other states. It is not necessary to determine the label of a state that is in a group by itself. This process is then repeated with the 3 groups. Since the labels within each group are the same, no new groups are created and the process is complete. States `B` and `D` are the only two equivalent states.

An application of Moore's algorithm is shown in Figure 5.28. The state transition table of a Mealy machine is on the left, and the application of the method is on the right. To be grouped in Step 1, two states should have the same output value for all input values. In general, there could be as many as four groups at first, but in this state machine, only two output value combinations occur.

The minimized version of the machine is obtained by removing all but one state from each group and redirecting transitions to removed states to the remaining state for its group. This process is shown in Figure 5.29. On the left, in the original table, state C is replaced by state A, and state D is replaced by B in the NS section. On the right, the rows corresponding to states C and D have been removed to produce the final state transition table.

	NS		Output	
PS	x=0	x=1	x=0	x=1
A	E	D	0	1
B	F	D	0	0
C	E	B	0	1
D	F	B	0	0
E	C	F	0	1
F	B	C	0	0

```
              1            2
          ( A C E )    ( B D F )         Step 1. Group states by output
x=0         1 1 1        2 2 2           Step 2. Generate next-group labels
x=1         2 2 2        2 2 1

              1          2        3
          ( A C E )   ( B D )   ( F )    Step 3: Sub-divide groups
x=0         1 1 1       3 3       -      Step 2. Generate next-group labels
x=1         2 2 3       2 2       -

            1       2       3       4
          ( A C )  ( E )  ( B D )  ( F ) Step 3: Sub-divide groups
x=0         2 2      -      4 4      -   Step 2. Generate next-group labels
x=1         3 3      -      3 3      -

          ( A C )  ( E )  ( B D )  ( F ) Step 3: Sub-divide groups
          No change: Done
```

Figure 5.28: Moore's algorithm applied to a Mealy machine.

	NS		Output	
PS	x=0	x=1	x=0	x=1
A	E	~~D~~ B	0	1
B	F	~~D~~ B	0	0
C	E	B	0	1
D	F	B	0	0
E	~~C~~ A	F	0	1
F	B	~~C~~ A	0	0

	NS		Output	
PS	x=0	x=1	x=0	x=1
A	E	B	0	1
B	F	B	0	0
E	A	F	0	1
F	B	A	0	0

Figure 5.29: Result of Moore's algorithm.

Chapter 6

Sequential System Design

6.1 Introduction

Any digital design can be viewed as a “flattened” network of gates and FFs. Viewing a large design in this way is complicated and tedious due to the number of gates and FFs involved. Applying different design strategies in different parts of the design may be advantageous. For example, using one-hot encoding would not be advisable for a 16-bit counter: 2^{16} FFs would be needed! Instead of a flattened network, designs can be assembled from smaller sequential components, with a state machine controlling their interaction. We begin with a very simple (somewhat contrived) design of a system to count bicycles crossing a sensor. Our designs will be synchronous in that all of their sequential components will share the same clock. In Section 6.5 we will discuss how to provide outputs to a component that is not synchronous (does not share the clock).

6.2 Bike Counter

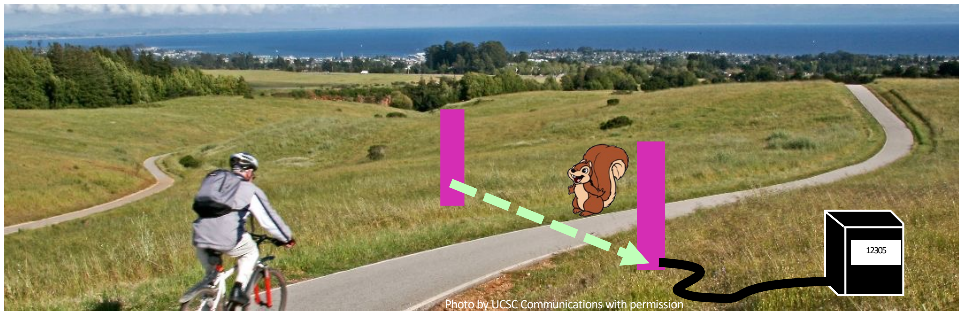


Photo by UCSC Communications with permission

Figure 6.1: UCSC bike path with its planned bike counter system.

The UC Santa Cruz transportation department has requested help implementing a device to count the number of bicycles using the bike path. The proposed system is shown in Figure 6.1. It consists of a light aimed across the path with a sensor on the other side detecting when the light is blocked by an object.

The sensor provides a signal that is 1 when the light is blocked and 0 when it is not blocked. Extensive studies have been conducted to determine the "signature" on the sensor output of a bicycle crossing the light beam. This is complicated by the large squirrel population living in the meadow that has been interfering with the sensor in protest of planned development on the meadow. Figure 6.2 shows the signal expected when a bicycle crosses the sensor. Based on the studies, it has been determined that when the sensor is blocked twice with at most 16 cycles of the provided clock in between, a bicycle has crossed.[1]

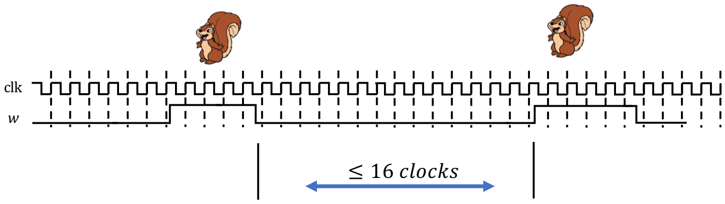


Figure 6.2: Signature on the sensor output of a bicycle crossing the light sensor.

The symbol for the desired component is shown in Figure 6.3. It has the sensor signal `w` as an input and the provided clock signal. The output of the component is the number of bicycles detected.

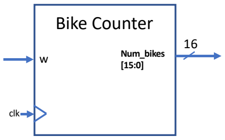


Figure 6.3: Symbol for the component counting bicycles crossing the sensor.

Since we will be counting up every time a bicycle is detected, using a counter to store this number is an obvious choice. In Figure 6.4 the 16-bit counter `Bcounter` will be used to record the number of bikes. We assume this counter is reset to 0 on start-up. The `CE` input of this counter should be 1 for each bike detected. To generate the `bike` input that increments `Bcounter`, the input from the sensor `w` needs to be monitored by a state machine. To detect a bicycle, the input `w` needs to be 1 and then 0 and then 1 again with no more than 16 clock cycles between the blocks of 1's. Our state machine could use 16 states to count the 0's, but instead a 4-bit counter `Time_Counter` will be used to keep track of the number of 0's. This design choice makes the state machine smaller, but it now needs to interact with `Time_Counter`. We will need to start `Time_Counter` after the first block of 1's and see if it reaches 15 before another block of 1's begins. We have chosen to have `Time_Counter` continuously count up by setting its `CE` input to 1 and resetting the counter when we need to start the count. The `TC` output will serve as the signal that 16 clock cycles have elapsed.

With the block diagram complete, the components can now be designed. In the process, you may

[1] ... as opposed to angry squirrels, pedestrians, or two unicycles.

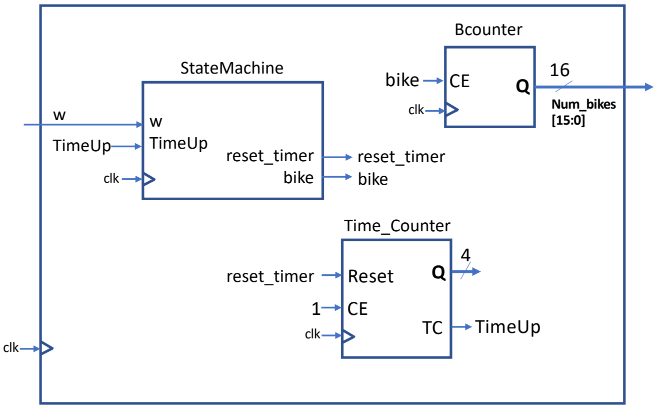


Figure 6.4: Block diagram for Bike Counter.

discover that additional component inputs/outputs are needed or even additional components. This is an iterative process, but completing as much of the block diagram as possible will reduce the revision(s).

The counters have been designed in Sections 4.5 and 5.1. The state machine is the remaining component to design. Its state diagram is in Figure 6.5. In the initial state IDLE, we wait for w to become 1. When w is 1 the machine transitions to state FW and stays there until w is 0. This is the first block of 1's that may correspond to the front wheel of a bicycle. Time_Counter is reset on the transition out of state FW since the number of 0's must now be monitored. In state BTW the machine waits until either w or TimeUp becomes 1. If w becomes 1 before TimeUp then the second block of 1's has occurred within 16 clock cycles. In this case the machine transitions to state RW where it will wait for the second block of 1's representing the rear wheel to end. On the transition back to IDLE the Bcounter is incremented since the bike should not be counted until it has completely past the sensor. If bike was 1 while in state RW the counter would increment at each clock edge while w remained 1. In state BTW, if TimeUp becomes 1 before w then we transition back to state IDLE and await the next bicycle.

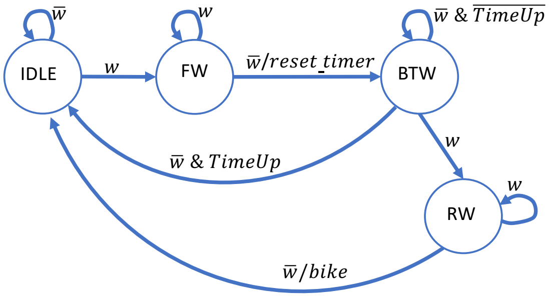

$\overline{w}$
IDLE
w
w
FW
$\overline{w}/reset\_timer$
BTW
$\overline{w}$ & $\overline{TimeUp}$
$\overline{w}$ & $TimeUp$
w
RW
w
$\overline{w}/bike$

Figure 6.5: State diagram for state machine of Bike Counter.

The state diagram in Figure 6.5 does not completely specify the values of the two outputs, `reset_time` and `bike`. The two cases where these outputs need to be *asserted* (in this case have value 1) are indicated, but their values elsewhere are not. The output `bike` should be 0 at all other transitions since the `Bcounter` should not increment. But for `reset_time`, there is more flexibility. We need `reset_time` to be 1 on the transition from `FW` to `BTW` and it should be 0 in `BTW`. But elsewhere it can be either 0 or 1. Though in the equations we have chosen to make it 0.

The next state and output equations for the state machine are below. Here we have chosen to use the names of the state (in upper case) as the state variables, rather than Q_i's. The corresponding next state variables have $NEXT\_$ as a prefix.

$$\begin{aligned}
Next\_IDLE &= \overline{w} * IDLE + \overline{w} * TimeUp * BTW + \overline{w} * RW \\
Next\_FW &= w * IDLE + w * FW \\
Next\_BTW &= \overline{w} * \overline{TimeUp} * BTW + \overline{w} * FW \\
Next\_RW &= w * RW + w * BTW \\
bike &= \overline{w} * RW \\
reset\_timer &= \overline{w} * FW
\end{aligned}$$

The state machine is implemented in Figure 6.6. Since the input `w` has multiple loads it would be wise to verify that this input is synchronous with the clock. If not it should be first passed through as D FF before connecting it to any components. The input `TimeUp` is synchronous since it is from a component with the same clock.

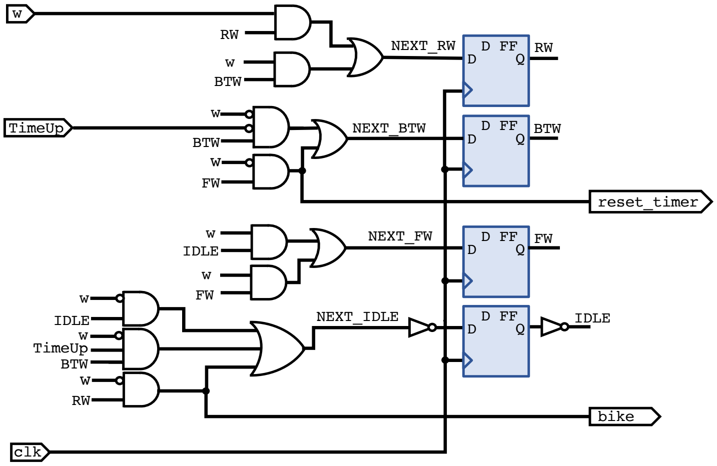


Figure 6.6: Logic diagram for state machine of Bike Counter.

6.2.1 Bike Counter revisited

More often than not, the first design is not the final one. Beyond design errors, there could be misunderstanding of the desired behavior or the specification may have been revised. It is tempting to attempt a patch to the existing design by inserting or removing components, but often a better approach is to redesign the state machine.

Here we will consider a change request from the transportation department. They have requested an additional output, `error`, that they plan to use for an error indicator light on the device. The output `error` should become 1 when more than 16 clocks have elapsed since the first block of 1's on `w`. This corresponds to the transition from state `BTW` back to `IDLE`: $setError = \overline{w} * TimeUp * BTW$. The output `error` should remain 1 while `w` is 0 and return to 0 when `w` is 1 again in state IDLE: $resetError = w * IDLE$. In Figure 6.7, a D FF has been added to provide the output `error`.

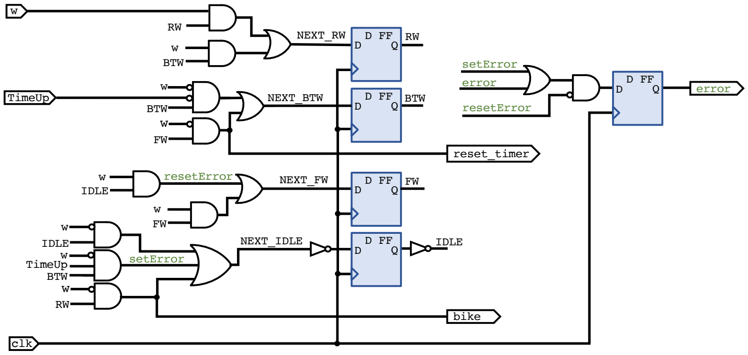


Figure 6.7: Patch applied to original Bike Counter logic diagram to provide the output `error`.

The signals `seterror` and `reseterror` control this new D FF.

Consider instead the revised state diagram in Figure 6.8. A fifth state `ERR` has been added and the output `error` will be 1 in this state and 0 elsewhere. The machine enters state `ERR` from `BTW` when time expires and `w` is still 0. It stays in state `ERR` until `w` is 1 again. Since this would be the start of a block of 1's the machine goes directly from `ERR` to `FW` when `w` is 1. The next state and output logic equations for the revised state machine are below.

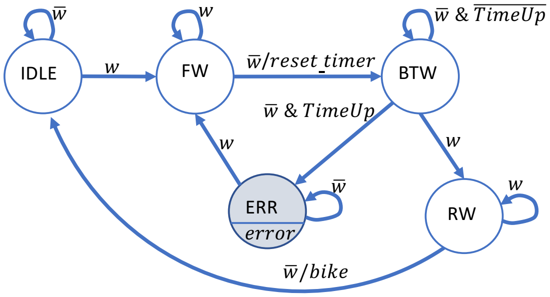


Figure 6.8: Revised state diagram with additional state `ERR`.

$$
\begin{aligned}
Next\_IDLE &= \overline{w} * IDLE + \overline{w} * RW \\
Next\_FW &= w * IDLE + w * ERR + w * FW \\
Next\_BTW &= \overline{w} * \overline{TimeUp} * BTW + \overline{w} * FW \\
Next\_RW &= w * RW + w * BTW \\
Next\_ERR &= \overline{w} * TimeUp * BTW + \overline{w} * ERR
\end{aligned}
$$

$$\begin{aligned} bike &= \overline{w} * RW \\ reset\_timer &= \overline{w} * FW \end{aligned}$$

The corresponding logic diagram is in Figure 6.9.

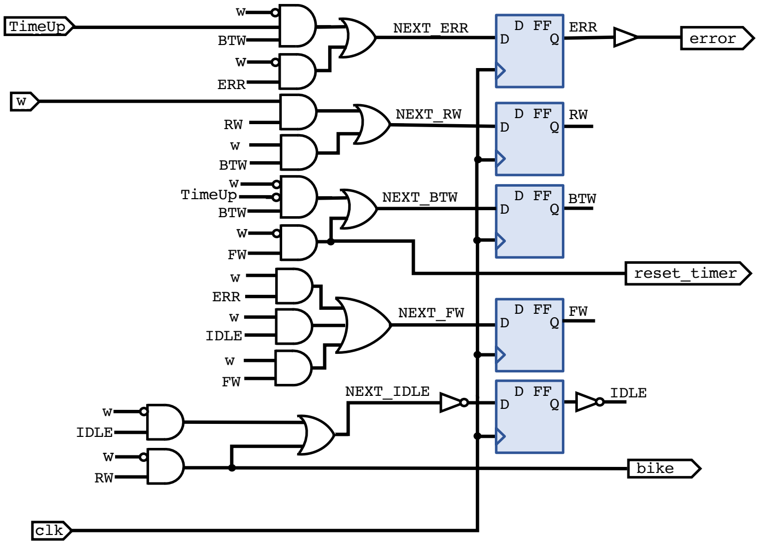


Figure 6.9: Logic circuit for the revised state diagram with additional state ERR.

The two designs in Figures 6.7 and 6.8 both have 5 FFs. If we consider the error FF to be part of the state machine in Figure 6.7, then this design has a state machine with 5 states using the encodings 10000, 01000, 00100, 00010, and 00011. There are two IDLE states, one with error having value 1 and the other with error having value 0. The difference between the two designs is essentially how the 5 states were encoded, and conceptually for the designer the separation between the *control* and supporting components.

6.3 Shift and Add Multiplier

The multiplication of integers can be implemented either as a combinational or sequential circuit. There are a variety of approaches, trading off time and resources. In this section, a sequential multiplier for two n-bit unsigned integers is constructed with a simple shift and add approach. The symbol for the multiplier is shown in Figure 6.10. Because the number of clock cycles needed to complete the multiplication will vary, there is an output Ready that signals when the multiplier is available or busy. The Ready output also indicates that the output bus P has the result from the most recent multiplication. When the multiplier is available, the input Go will begin the multiplication of the two integers on the inputs A and B. These two inputs should be valid on the same clock edge where Go becomes 1, but need not remain afterward. The Ready output will be 0 on the next clock edge. When the Ready output is 1 again, this indicates that the output bus P has the result of the multiplication of A and B. The result of a multiplication of two n-bit unsigned integers will require at most $2n$ bits.

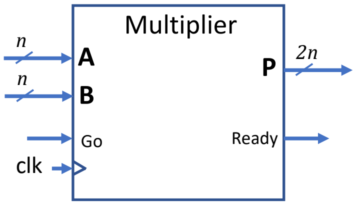
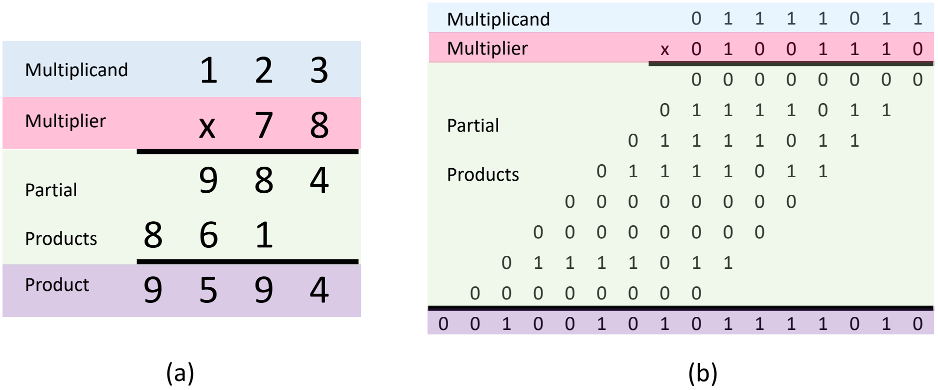


Figure 6.10: Symbol for multiplier.



Multiplicand		1	2	3
Multiplier		x	7	8
Partial		9	8	4
Products	8	6	1	
Product	9	5	9	4

(a)

Multiplicand									0	1	1	1	1	0	1	1
Multiplier								x	0	1	0	0	1	1	1	0
									0	0	0	0	0	0	0	0
Partial								0	1	1	1	1	0	1	1	
							0	1	1	1	1	0	1	1		
Products						0	1	1	1	1	0	1	1			
					0	0	0	0	0	0	0	0				
				0	0	0	0	0	0	0	0					
			0	1	1	1	1	0	1	1						
		0	0	0	0	0	0	0	0							
	0	0	1	0	0	1	0	1	0	1	1	1	1	0	1	0

(b)

Figure 6.11: Multiplication of 123 and 78: (a) Decimal and (b) Binary.

The steps in the multiplication of 123 and 78 are shown in Figure 6.11 for both decimal and binary representations. In each step, a partial product is obtained by multiplying a single digit of the multiplier (either 7 or 8) with the multiplicand (123). The same procedure is used for binary multiplication, but since the multiplier digits are either 0 or 1, each partial product is either 0 or the multiplicand. The partial products are shifted to the left lining up with the position of their multiplier digit.

In a shift and add multiplier, the partial products are generated one at a time by shifting the multiplicand to the left. Rather than collecting all the partial products first and then summing them, they can be added to a running total as they are generated. In Figure 6.12, a $2n$-bit register is used to hold the running total.[2] The adder sums each partial product with the output of the register and the result is loaded into the register using the control input `LD`. The register should be 0 initially. It can be reset to 0 with the control input (`R` input). As mentioned the partial products are either 0 or shifted versions of the multiplicand. They are generated using the $2n$-bit shift register shown in Figure 6.13. The multiplicand `A` is extended by adding n 0's on the left. Its extended value will be loaded into the shift register when the control input `LD` is 1 at the clock edge. The shift register will shift its contents (`Q`) to the left when the control input `SHL` is 1 at the clock edge. The input `SIn` is the new value of the rightmost bit when a shift occurs. In this case, it is set to 0 so that 0's will fill in as the multiplicand is shifted right.

When the multiplier bit is 0, the partial product should be 0 rather than the shifted multiplicand. Rather than generating a bit vector of 0's for the adder, we will skip adding the multiplicand by

[2] A register filling this type of role is often referred to as an *accumulator*.

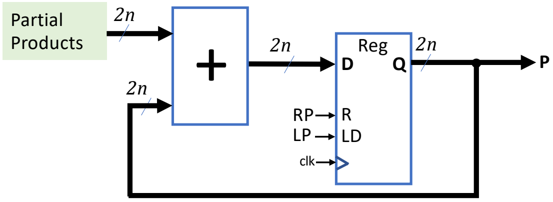


Figure 6.12: Summing the partial products.

not loading the result of the adder into the register. Not adding anything has the same effect as adding 0.

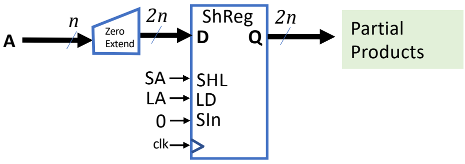


Figure 6.13: Generating the partial products.

The bits of the multiplier B will be "served up" one by one to determine whether the shifted multiplicand should be added. We need to store B since this input may no longer be valid after the initial Go signal. The shift register in Figure 6.14 should be loaded with the B input at the start and then shifted right so that the next multiplier bit is available as the rightmost bit (NB[0] of the output NB. The 0 on the SIn input will fill in the bits from the left with 0's as the bits of B are shifted out. When all of the bits in the shift register are 0 the multiplication is complete. There will be no further additions. The NOR gate output will be 1 when all, except possibly the rightmost bit, of NB is 0.

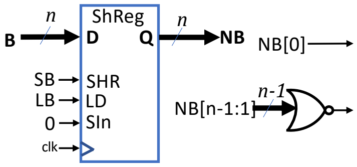


Figure 6.14: Shift register for providing multiplier bits.

The block diagram in Figure 6.15 assembles the three components. The missing piece is the controller that will provide values to control their inputs as well as the Ready external output. The state diagram for the controller is shown in Figure 6.16. Only two states are needed. In the initial state CHILL. the machine waits for a Go signal while indicating that it is ready. When Go becomes 1 at the clock edge, the registers are initialized by asserting RP, LA, and LB. The machine transitions to state SHADD, where both shift registers will shift at clock edges. The P register is loaded when m, the current multiplier bit is 1. When the input z is 1, the machine transitions back to state CHILL

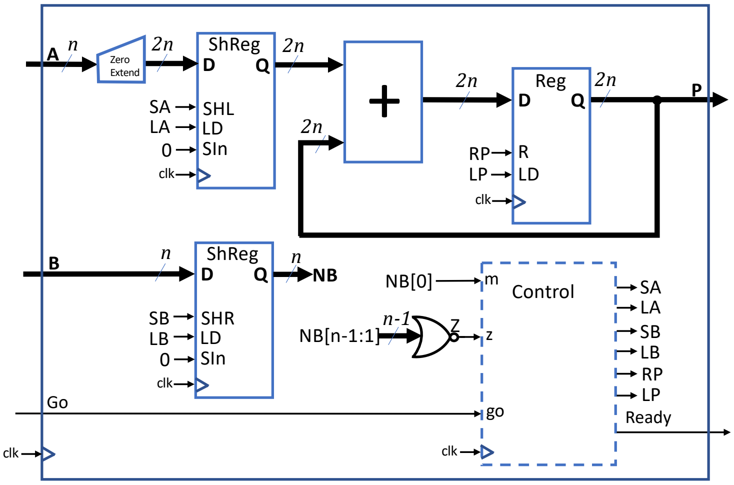


Figure 6.15: Block diagram for Shift and Add multiplier.

since there will be no more additions after the clock edge. The outputs that need to be *asserted* have been indicated on the arcs (Mealy outputs) or in the state (Moore output). Table 6.1 is

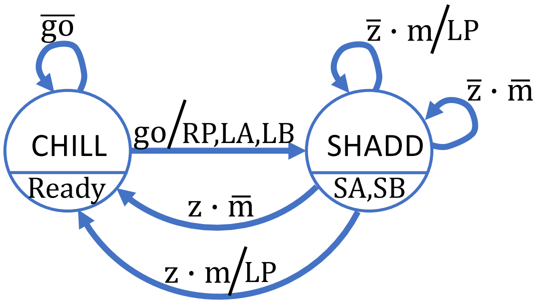


Figure 6.16: State diagram for multiplier control.

the state transition table. Here a different format is used. Instead of a column for each possible value of the three inputs (8 columns!) there is a column for each input, and each row corresponds to a transition in the state diagram. Each row identifies the current state and the combination of inputs for a specific transition and provides the next state and values of the outputs.[3] From the state diagram, 1's have been entered when the outputs need to be asserted. The remaining entries could all be 0, but there may be cases where there is a choice. For example, the outputs SA and SB can either be 0 or 1 in state CHILL while go is 0. This is not the case for the LP since register P must keep its value, the result of the multiplication, until the next go signal. In Table 6.1 don't cares (?'s) have been entered for output values that can be either 0 or 1. Depending on the shift register design, it might be possible for their LD and SH inputs to be simultaneously 1. But without knowing their specific behavior, we need to assume that the shift registers' LD and SH inputs should not be simultaneously 1 when their contents need to be valid. There are 7 outputs shown in

[3] Computer architecture students may note the similarity with microcode.

Present	Inputs			Next	Outputs						
State	go	z	m	State	Ready	RP	LA	LB	LP	SA	SB
CHILL	0	–	–	CHILL	1	0	?	?	0	?	?
CHILL	1	–	–	SHADD	1	1	1	1	0	0	0
SHADD	–	0	0	SHADD	0	0	0	0	0	1	1
SHADD	–	0	1	SHADD	0	0	0	0	1	1	1
SHADD	–	1	0	CHILL	0	0	0	0	0	?	?
SHADD	–	1	1	CHILL	0	0	0	0	1	?	?

Table 6.1: State transition table for Shift and Add multiplier.

Table 6.1, but they can be reduced to 4. For example, RP, LA, and LB can all be one signal which we will name initR, by resolving LA, and LB's don't cares to agree with RP. SA and SB can be the same signal, sh. The output LP is not compatible with any of the other outputs. Since there are only two states, only one FF is needed. The state and output equations are:

$$\begin{aligned}
Next\_CHILL &= \overline{go} * CHILL + z * SHADD \\
SHADD &= \overline{CHILL} \\
initR &= go * CHILL \\
sh &= SHADD \\
loadP &= m * SHADD \\
ready &= CHILL
\end{aligned}$$

The completed block diagram is in Figure 6.17. The 6 outputs needed to control the registers have been reduced to 3 (SH, LP, and InitR) and connected to the registers.

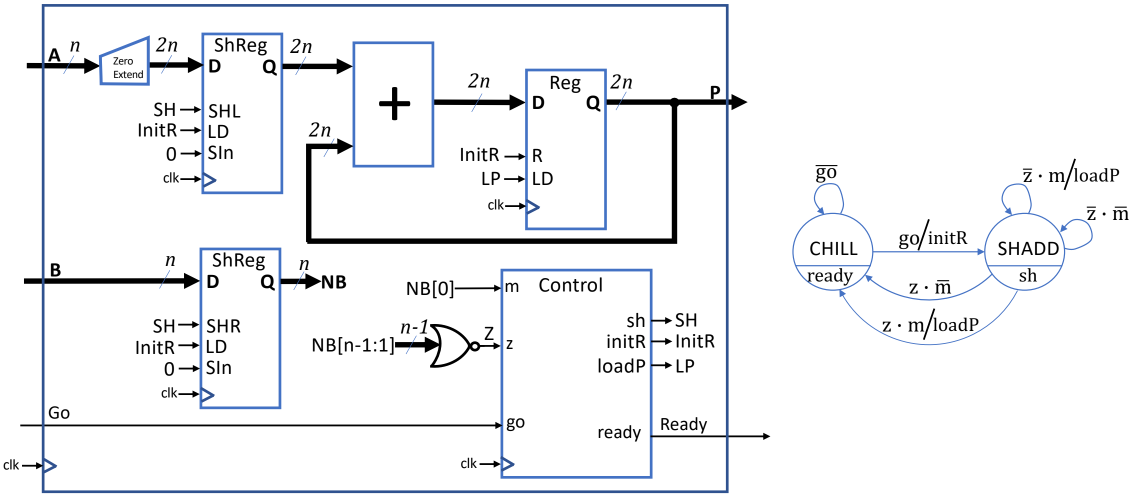


Figure 6.17: Completed block diagram and state diagram for the Shift and Add Multiplier.

The design in this section mimics the familiar multiplication method taught in grade schools. This particular design has two obvious inefficiencies. First, at each stage, only an n-bit addition is taking place. In all but n of the columns 0 is being added. Hence the $2n$-bit adder could be replaced with an n-bit adder by shifting the accumulated sum `P` rather than the multiplicand `A`. Second, checking whether the remaining multiplier bits are all 0 as the termination condition (an n-bit NOR) will eventually, as n grows, become less efficient than using a counter to track the number of iterations. In any event, more efficient multiplier designs exist that use faster adders, higher radix, and take direct advantage of the realization technology.

6.4 Shift and Subtract Divider

To divide an integer `A` by `D` we will reverse the process used in the Shift and Add multiplier, essentially attempting to subtract potential partial products. The symbol for the divider is shown in Figure 6.18. The result of the division is two integers, the quotient `Q` and the remainder `Rem`.

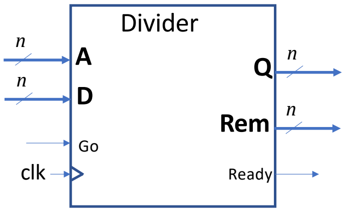


Figure 6.18: Symbol for Divider.

Although the number of clock cycles needed to complete the division will not vary, the divider still provides an output `Ready` that signals when the divider is available or busy. The `Ready` output also indicates that the output buses `Q` and `Rem` have the result from the most recent division. When the divider is available, the input `Go` will begin the division of the two integers on the inputs `A` and `D`. These two inputs should be valid on the same clock edge where `Go` becomes 1, but need not remain afterward. The `Ready` output will be 0 on the next clock edge. When the `Ready` output is 1 again, this indicates that the output buses `Q` and `Rem` have the result of the division of `A` by `D`.

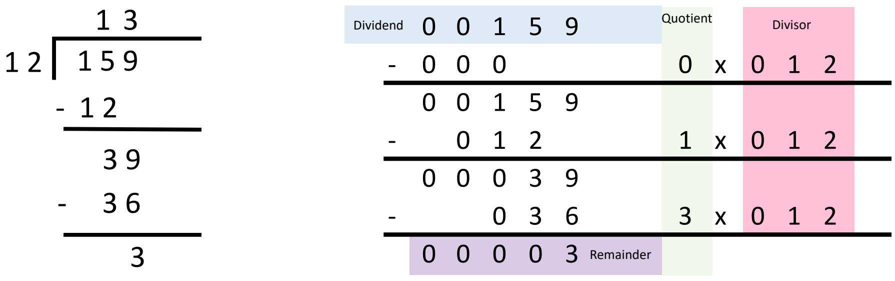


Figure 6.19: Decimal long division of 159 by 12.

Figure 6.19 shows the paper-pencil long division of 159 by 12 on the left, with the process on the right. In each step, a shifted version of the divider is considered. The number of multiples of this shifted version that can be subtracted from what remains of the dividend determines the next digit of the quotient. The initial remainder is the dividend. In the paper-pencil method, the most significant digits of the dividend and divisor were aligned, while a general process for 3-digit integers would zero-extend the dividend by two digits so that the least significant digit of the divisor would align with the most significant digit of the dividend.

In binary, the only multiples of the divisor to consider are 0 or 1 so we merely need to check whether we can subtract the divisor from the remainder. The division of 159 by 12 assuming 8 binary digits is shown in Figure 6.20(a). Our divider will align the rightmost bit of the divisor with the leftmost bit of the dividend, In Figure 6.20(b), the process for the division of 8-bit integers D into A is shown. Each step requires a 15-bit subtraction.

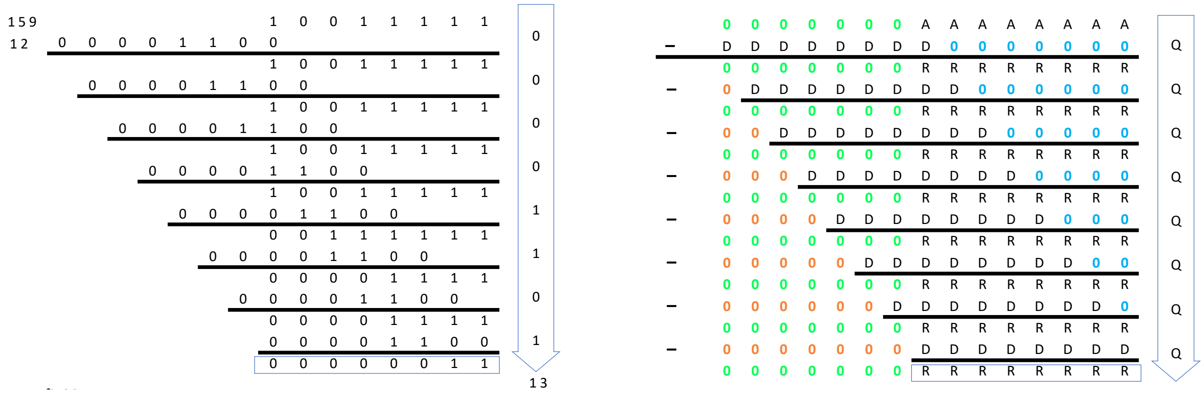

159 1 0 0 1 1 1 1 1
12 0 0 0 0 1 1 0 0
1 0 0 1 1 1 1 1
0 0 0 0 1 1 0 0
1 0 0 1 1 1 1 1
0 0 0 0 1 1 0 0
1 0 0 1 1 1 1 1
0 0 0 0 1 1 0 0
1 0 0 1 1 1 1 1
0 0 0 0 1 1 0 0
0 0 1 1 1 1 1 1
0 0 0 0 1 1 0 0
0 0 0 0 1 1 1 1
0 0 0 0 1 1 0 0
0 0 0 0 1 1 1 1
0 0 0 0 1 1 0 0
0 0 0 0 0 0 1 1
0
0
0
0
1
1
0
1
13
0 0 0 0 0 0 0 A A A A A A A A
− D D D D D D D D 0 0 0 0 0 0 0 Q
0 0 0 0 0 0 0 R R R R R R R R
− 0 D D D D D D D D 0 0 0 0 0 0 Q
0 0 0 0 0 0 0 R R R R R R R R
− 0 0 D D D D D D D D 0 0 0 0 0 Q
0 0 0 0 0 0 0 R R R R R R R R
− 0 0 0 D D D D D D D D 0 0 0 0 Q
0 0 0 0 0 0 0 R R R R R R R R
− 0 0 0 0 D D D D D D D D 0 0 0 Q
0 0 0 0 0 0 0 R R R R R R R R
− 0 0 0 0 0 D D D D D D D D 0 0 Q
0 0 0 0 0 0 0 R R R R R R R R
− 0 0 0 0 0 0 D D D D D D D D 0 Q
0 0 0 0 0 0 0 R R R R R R R R
− 0 0 0 0 0 0 0 D D D D D D D D Q
0 0 0 0 0 0 0 R R R R R R R R

Figure 6.20: (a) 8-bit binary division of 159 by 12; (b) 8-bit binary division of A by D.

Figure 6.21 contains the design of the divider for n-bit unsigned integers corresponding to the process in Figure 6.20(b). Here the inputs, unsigned n-bit integers, are converted to $2n$-bit signed integers to support the subtraction of unsigned $2n-1$-bit integers. Three registers are used to hold the remainder (**R**), the divisor (**D**), and the quotient (**Q**). The **R** register is loaded with the zero-extended A at the start and then will be loaded with the result of the subtraction when needed. The **D** shift register is loaded with the shifted divisor at the start and then shifted right at each iteration. The **Q** shift register holds the resulting quotient at the end of the process by capturing each bit as it is produced bit and shifting left. The Q shift register is also used to detect the last iteration: it is initially loaded with the n-bit vector representing "1" ($n-1$ zero's and 1 in the rightmost position) and then Q's leftmost bit will become 1 after exactly $n-1$ iterations.

The state diagram for the controller is shown in Figure 6.22. As with the Shift and Ad Multiplier, only two states are needed. In the initial state, `CHILL`, the machine waits for the `go` while indicating that it is ready to begin a new division and that outputs Q and R hold the result of the previous division (if any). The machine transitions to the state `TESTSUB` when it receives the `go` signal. While in `TESTSUB` the D and Q registers will be shifted, and the R register will be loaded if D can be subtracted from R (if D is not greater than R). Once $n-1$ bits of the quotient has been shifted into Q, the leftmost bit of Q (`last`) becomes 1, and the machine transitions back to `CHILL` as it shifts in the last bit of the quotient into Q and updates R if needed.

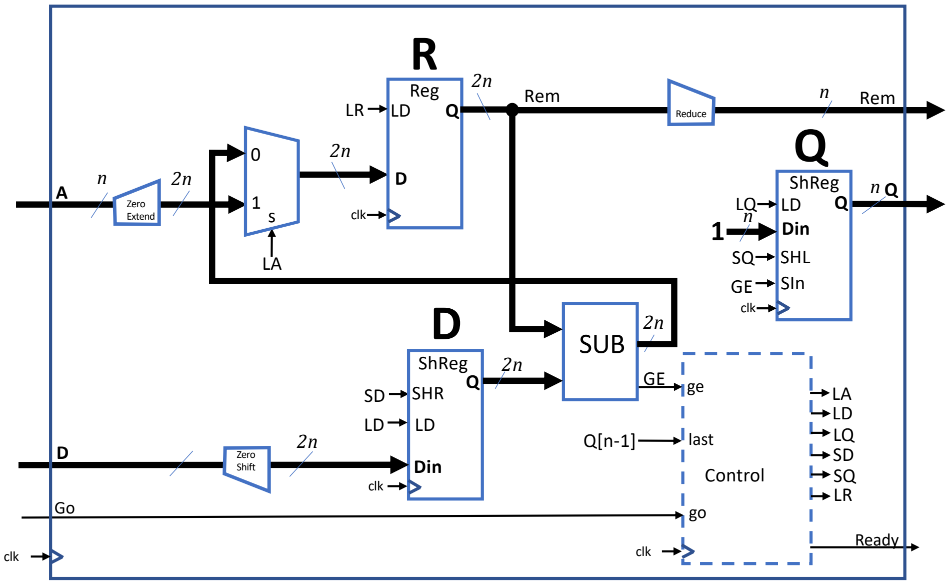


Figure 6.21: Design of divider for n-bit integers.

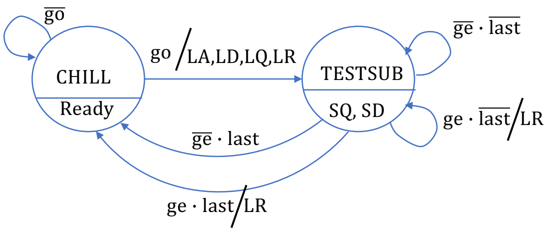

$\overline{go}$
CHILL
Ready
go / LA,LD,LQ,LR
TESTSUB
SQ, SD
$\overline{ge} \cdot \overline{last}$
$ge \cdot \overline{last}$ / LR
$\overline{ge} \cdot last$
$ge \cdot last$ / LR

Figure 6.22: State diagram for divider control.

The state transition table for this machine, Table 6.2, uses the same format as the multiplier's: each row identifies the current state and the combination of inputs for a specific transition, and provides the next state and values of the outputs. From the state diagram, 1's have been entered when the outputs need to be asserted. The remaining entries could all be 0, but there may be cases where there is a choice. For example, the outputs `LA` and `LD` can either be 0 or 1 in state `CHILL` while `go` is 0. This is not the case for the `LQ` and `LR` since registers `Q` and `R` must keep their contents, the result of the division, until the next `go` signal. In Table 6.2 don't cares (?'s) have been entered for output values that can be either 0 or 1. There are 7 outputs shown in Table 6.2, but they can be reduced to 4. Outputs `LA`, `LD`, and `LQ` can all be one signal which we will name `InitD`, by resolving `LA`, and `LD`'s don't cares to agree with `LQ`. `SD` and `SQ` can be the same signal, `SH`. The output `LR` is not compatible with any of the other outputs. Since there are only two states, only one FF is needed. The state and output equations are:

$$
\begin{aligned}
Next\_CHILL &= \overline{go} * CHILL + last * TESTSUB \\
TESTSUB &= \overline{CHILL}
\end{aligned}
$$

Present	Inputs			Next	Outputs						
State	`go`	`last`	`ge`	State	`Ready`	`LA`	`LD`	`LQ`	`LR`	`SD`	`SQ`
CHILL	0	–	–	CHILL	1	?	?	0	0	?	0
CHILL	1	–	–	TESTSUB	1	1	1	1	1	0	0
TESTSUB	–	0	0	TESTSUB	0	?	0	0	0	1	1
TESTSUB	–	0	1	TESTSUB	0	0	0	0	1	1	1
TESTSUB	–	1	0	CHILL	0	?	?	0	0	?	1
TESTSUB	–	1	1	CHILL	0	0	?	0	1	?	1

Table 6.2: State transition table for Divider.

$$\begin{aligned} initD &= go * CHILL \\ sh &= TESTSUB \\ loadR &= go * CHILL + ge * TESTSUB \\ ready &= CHILL \end{aligned}$$

The completed design of the divider is in Figure 6.23.

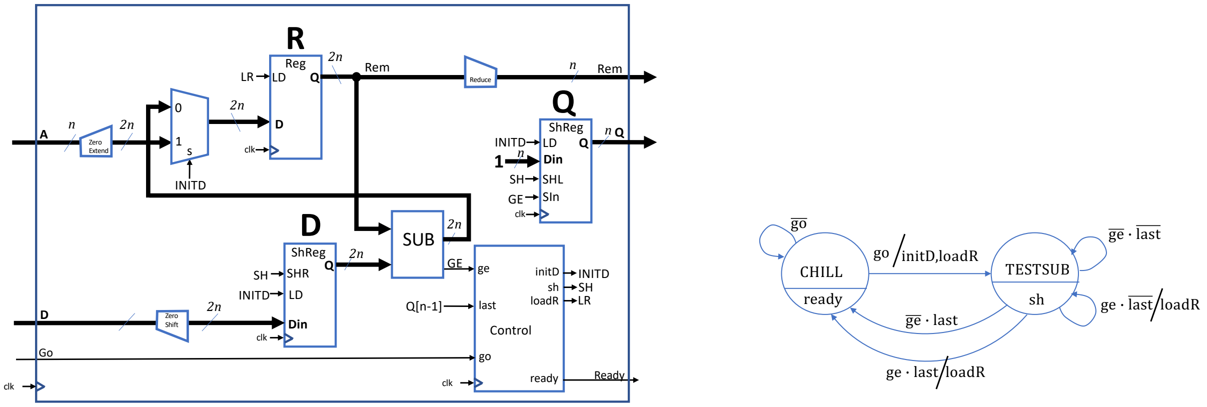


Figure 6.23: Completed design of divider.

As in the multiplier design, only an n-bit subtraction is taking place. Hence the $2n$-bit subtraction (adder) can be reduced to n-bits by shifting the remainder `R` rather than the divisor `D`.

6.5 Asynchronous Outputs: A cautionary tale

In Section 4.8, the use of a D FF as a synchronizer was introduced to synchronize an asynchronous input so to avoid violating timing constraints. Care must also be taken when outputs are connected to devices that do not share the same system clock. Some of these devices may be sensitive to glitches as well as the timing of transitions.

In Section 2.10, we observed a glitch on the output of a 2-input multiplexer as shown in Figure 2.20.

Waveforms for this circuit's inputs and outputs over a longer time frame are shown in Figure 6.24. Here the output `f` has a glitch at time 60. This may or may not affect the component connected to `f`. To remove the glitch `f` can be passed through a D FF as shown in Figure 6.25. This not only removes the glitch but also results in a pulse that is the width of the clock. However, the output `f_synch` is delayed by adding the D FF and this might be an issue .....

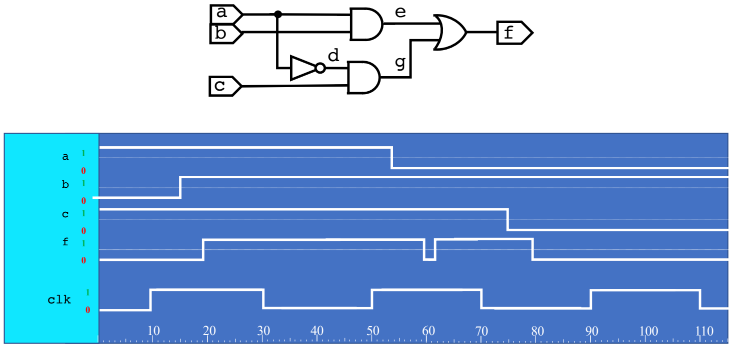


Figure 6.24: A 2-input multiplexer and a timing diagram showing its glitch.

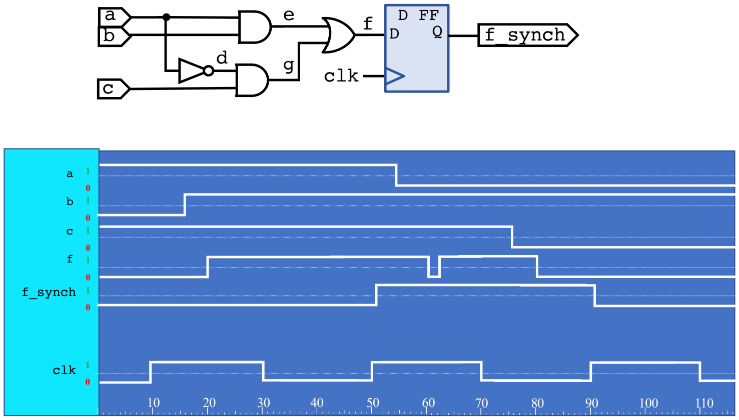


Figure 6.25: Passing the 2-input multiplexer output through a FF.

Consider the following design where the goal is a circuit that retrieves data based on an input key.[4] The desired component, `Dictionary`, is shown in Figure 6.26. This component has two synchronous inputs, a `Go` signal that begins the search, and the 8-bit `Key`. When `Go` is 1 on the clock edge, the search begins with the `Ready` transitioning to 0. After some number of clock cycles, the `Ready` output returns to 1. If the `Found` output is 0, no data corresponding to the `Key`. was found, But when the `Found` output is 1, the 12-bit output `Data` is associated with the `Key`. Both `Ready` and `Found` will keep their values until the next time `Go` is 1.

[4] In computer architecture such a circuit is referred to as an associative memory, but it is not implemented in the manner presented here.

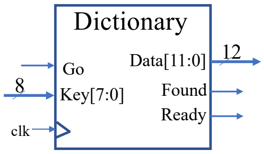


Figure 6.26: The `Dictionary` component.

Our `Dictionary` will be implemented using an asynchronous component, the `Rolodex` shown in Figure 6.27.[5] This component stores the keys and data as 20-bit words.

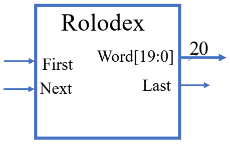


Figure 6.27: The `Rolodex` component storing the keys and data.

The list of words in the `Rolodex` are accessed one at a time using the inputs `First` and `Next`. The operation of the `Rolodex` is illustrated by the timing diagram in Figure 6.28. A high pulse on the `First` input resets the `Rolodex` to its first word. A high pulse on the `Next` input advances the `Rolodex` to the next word in its list. The output `Last` will be 1 when the last word in the list is reached. Note that the `Word` output is unknown while either `First` and `Next` are high. Since the `Rolodex` component has no clock input, the `First` and `Next` inputs must be valid at all times in addition to meeting specific timing requirements.

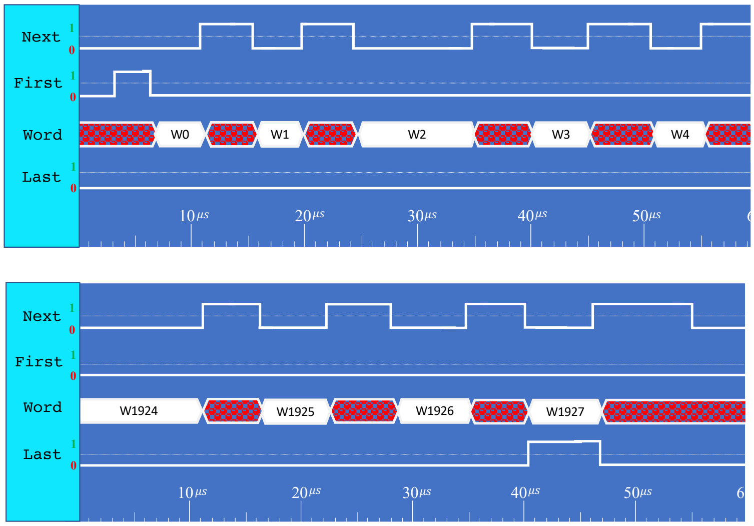


Figure 6.28: The operation of the `Rolodex` component storing the keys and data.

[5] This component has been imagined purely for this example.

To implement the `Dictionary` we create a state machine `FSM` that restarts and then advances the `Rolodex` until the upper 8 bits of the `Word` output matches `Key`. We stop advancing the `Rolodex` if there is a match or the last word is reached. The design, including the state diagram for `FSM`, is shown in Figure 6.29. Here the `Go` input is passed through an edge detector before the state machine so that only one search will be initiated when `Go` goes high. A comparator is used to determine when the top 8 bits of the `Rolodex` `Word` output matches the `Key` input. Note that this design requires the `Key` input to remain stable until the result of the search is no longer needed.

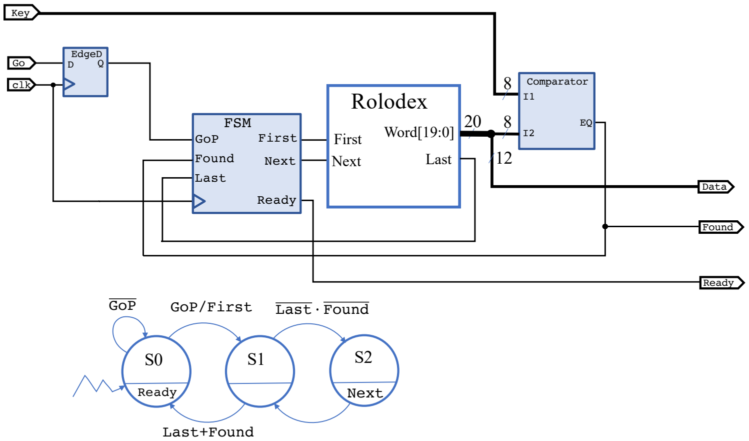


Figure 6.29: The initial design of the `Dictionary` component.

The functional simulation of the design in Figure 6.29 is shown in Figure 6.30.[6] The timing diagram on the left has a simulation where `Key` matches the fourth word (`W3`) and the timing diagram on the right has a simulation where `Key` matches the first word (`W0`). This is the expected behavior of the design.

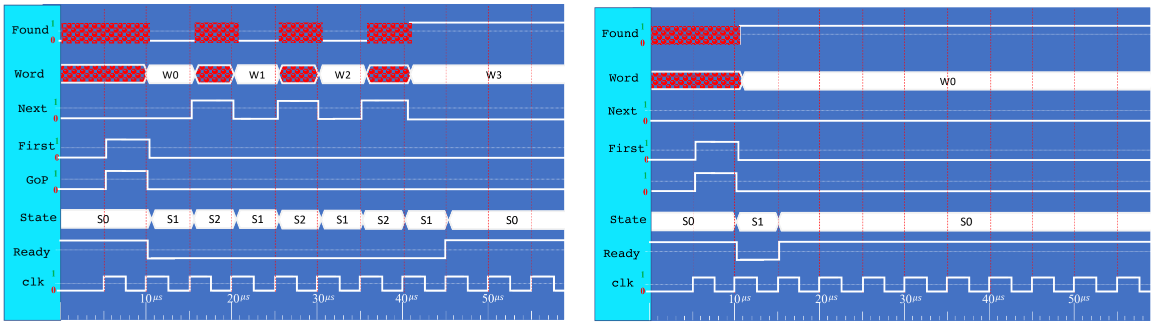


Figure 6.30: The functional simulation of the `Dictionary` design.

Unfortunately once implemented the design is not able to match the key associated with word `W0`. The timing simulation shown in Figure 6.31, may explain why the design is not performing correctly. The output `Next` of the state machine is 1 briefly just after the transition from state `S0`

[6] In a functional simulation there is no timing information for combinational logic.

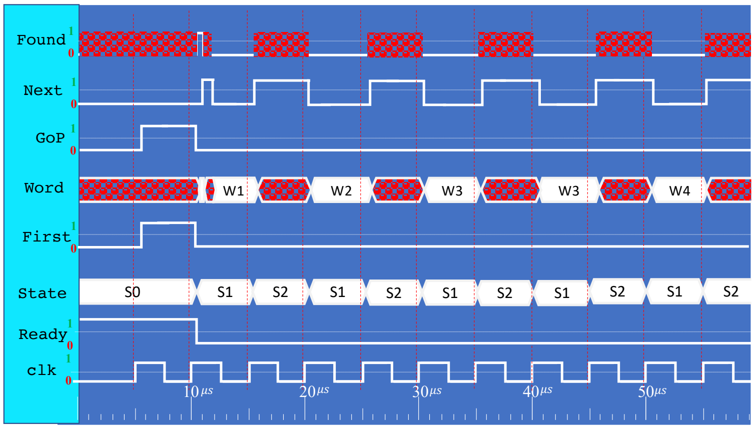


Figure 6.31: The timing simulation of the `Dictionary` design.

to state `S1`. This pulse causes the `Rolodex` to advance, rather than remain at word `W0` for one clock cycle. When the key is associated with word `W0`, the `Rolodex` does not stop at word `W0`. To remove this glitch on the output `Next`, we could add D FFs to the outputs of `FSM` as in Figure 6.32. The resulting timing simulation after introducing the D FFs is shown in Figure 6.33. The outputs of the new D FFs are `DFirst` and `DNext`. Although the glitch on `Next` is no longer reaching the `Rolodex`, `DNext` is delayed by one clock cycle, which delays the `Found` signal as well. Now `Found` is unknown when the state machine is in state `S1`. As a result, the next state becomes unknown.[7] Redesigning the state machine to take into account the one cycle delay of `DFirst` and `DNext` is one approach to solving the problem. A simpler approach is to examine the source of the glitch on `Next` in the original design.

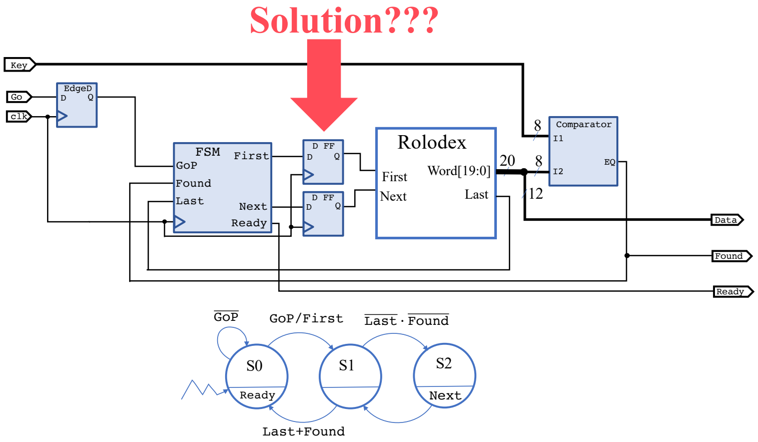


Figure 6.32: The revised design of the `Dictionary` component.

In Figure 6.34, the state diagram and encoding of `FSM` are shown on the top, and the circuit and

[7] The reader can verify that while passing only `Next` and not `First` through D FFs would correctly match a key corresponding to word `W0`, it will result in an unknown state when the key does not correspond to word `W0`.

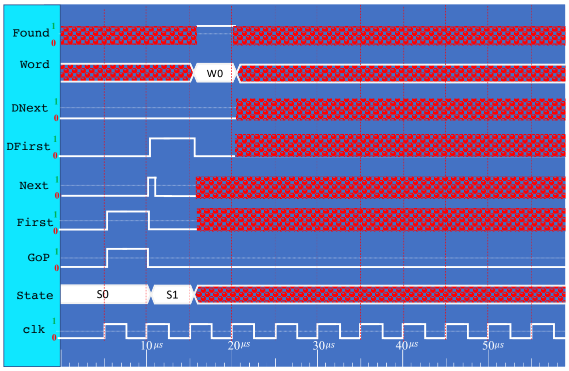
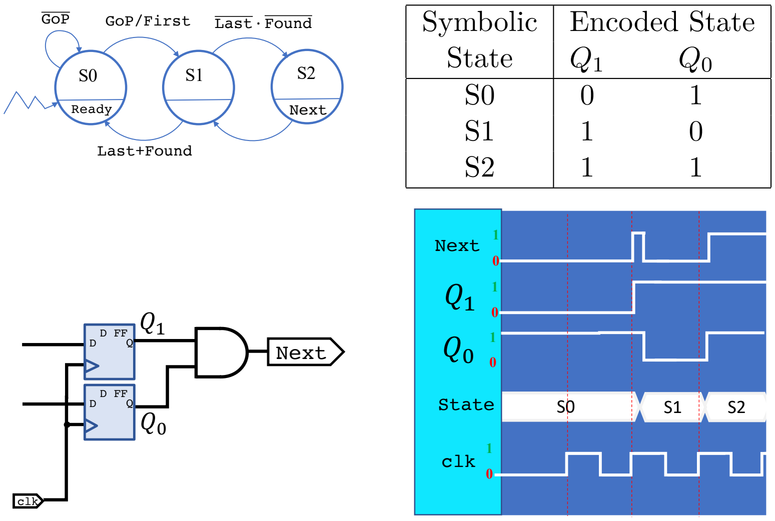


Figure 6.33: The timing simulation of the revised `Dictionary` design.



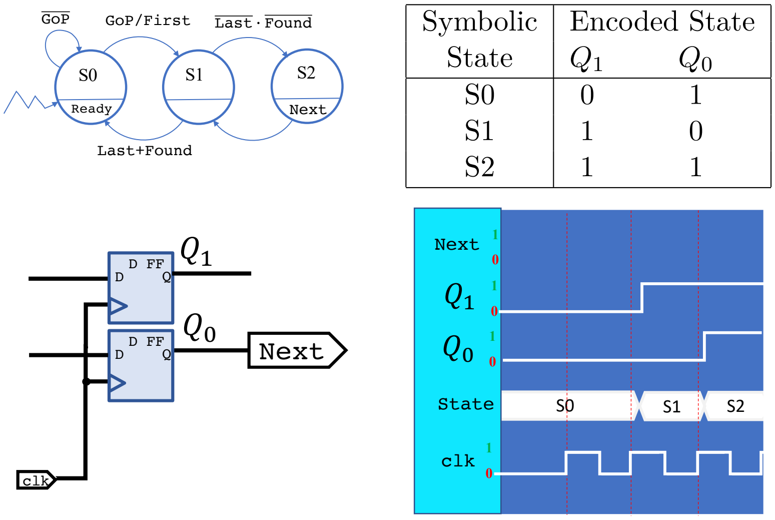
Symbolic State	Encoded State Q_1	Q_0
S0	0	1
S1	1	0
S2	1	1

Figure 6.34: Source of the glitch in the original `Dictionary FSM`'s Next output.

timing simulation of the output `Next` is shown on the bottom. As `FSM` transitions from `S0` to `S1` the outputs of the state FFs, $Q1$ and $Q0$ both change. It appears that $Q1$ transitions from 0 to 1, before $Q0$ transitions from 1 to 0. (This could be due to clock skew. But even without clock skew, wiring delays might result in $Q1$'s transition reaching the AND gate before $Q0$'s.) As a result, the output of the AND gate goes high and then low (a short high pulse). Even though `Next` is a Moore output, (depending only on the state of `FSM`), its output logic can have transients.

Rather than adding D FFs to outputs to remove transients, a state machine can be implemented so that all or some of its outputs come directly from FFs. In Figure 6.35, a different encoding is used for state `S0`. Now the logic for `Next` depends only on $Q0$, so `Next` is the output of the D FF for $Q0$.

In *output encoding*, the states of a state machine are encoded using their output values. Figure 6.36 shows an example. The table on the left in Figure 6.36 has the desired values for the outputs, $O0$, $O1$, and $O2$, of the five states of a state machine. A state encoding is on the right. The



Symbolic State	Encoded State Q_1	Q_0
S0	0	1
S1	1	0
S2	1	1

Figure 6.35: A new state encoding for the `Dictionary` FSM.

first three state bits, $Q0$, $Q1$, and $Q2$ correspond to the values of the outputs $O0$, $O1$, and $O2$. Since $S0$ and $S1$ have the same output values, a fourth state bit, $Q3$, is added to provide different encodings. Similarly, $S2$ and $S3$ are differentiated using the fourth bit. With this state encoding, the outputs will correspond to state bits and will be implemented as direct outputs of FFs. In a large design, it is often advantageous to have a small state machine handle the communication with an asynchronous device (such as a memory) and the output encoding would only be needed for the small state machine.

State	O_1	O_2	O_3
S0	1	0	0
S1	1	0	0
S2	1	1	1
S3	1	1	1
S4	0	0	1

State	Q_0	Q_1	Q_2	Q_3
S0	1	0	0	0
S1	1	0	0	1
S2	1	1	1	0
S3	1	1	1	1
S4	0	0	1	1

Figure 6.36: Output encoding of state machines.

Even with output encoding, two outputs that transition at the same clock edge, cannot be assumed to be exactly simultaneous. To ensure that one output transitions before the other, these changes should not occur on the same clock edge. At most one output should change value at every clock edge, such as in the Gray Code counter in Section 5.1.1.

One additional takeaway from the misguided insertion of D FFs into the `Dictionary` design is that adding FFs on internal connections may introduce clock delays between components that affect their operation. Even within a synchronous design, the effect on communication between components needs to be carefully considered when introducing clock delays on their interconnections. Retiming is a methodology for modifying the timing of a synchronous circuit by inserting positive and negative clock delays.[8] These clock delay insertions are balanced to retain the original communication timing, but can then be repositioned within the circuit.

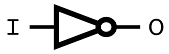
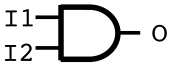
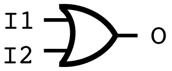
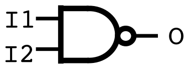
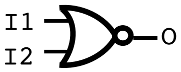
6.6 Gratuitous Advice

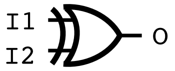
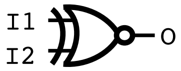
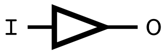
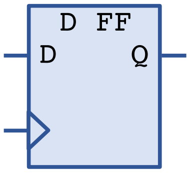
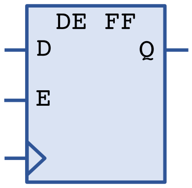
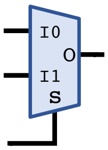
In this chapter, we have designed our circuits by selecting ”off-the-shelf” components and deriving a state machine to control them. The boundary between ”control” and ”data” in a circuit is a view imposed by the designer. The circuit itself is merely a list of gates, flip-flops, and nets. While the entire design can be thought of as a large state machine, decomposing a design allows the reuse of components, and imposes a structure that tools can use to optimize its implementation. Further, imposing a hierarchy can reduce the design’s complexity and thereby enhance correctness. Often there are multiple options in decomposing a design, offering trade-offs between complexity and efficiency. In the bike counter in Section 6.2, instead of a separate timer, the state machine could have included 16 additional states after the front wheel was detected to count the clock edges between wheels. With one-hot state encoding, this amounts to replacing the 4-bit binary counter for the timer with a 16-bit ring counter. Beginning a design by creating the block diagram and symbols for the top-level components is an excellent way to discover and create a hierarchy in the design.

[8] Despite a long and exhaustive search, the implementation of a device with a negative clock delay, a circuit whose output predicts the value of its next input, still eludes us.

Appendix A

Symbols

Name	Symbol	Logic	Page
NOT		I O 0 1 1 0	Page 1
AND		I1 I2 O 0 0 0 0 1 0 1 0 0 1 1 1	Page 1
OR		I1 I2 O 0 0 0 0 1 1 1 0 1 1 1 1	Page 1
NAND		I1 I2 O 0 0 1 0 1 1 1 0 1 1 1 0	Page 24
NOR		I1 I2 O 0 0 1 0 1 0 1 0 0 1 1 0	Page 24

Name	Symbol	Logic	Page
XOR		I1 I2 | O 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0	Page 10
XNOR		I1 I2 | O 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1	Page 30
BUF		I | O 0 | 0 1 | 1	Page 58
D FF		D clock | Q 0 ↑ | 0 1 ↑ | 1 - $\not\uparrow$ | no change	Page 49
DE FF		D E clock | Q 0 1 ↑ | 0 1 1 ↑ | 1 - 0 - | no change - - $\not\uparrow$ | no change	Page 59
2-1MUX		s | O 0 | I0 1 | I1	Page 29

Appendix B

Binary Representations

Many different binary codes assemble strings of “0” and “1” to represent various objects/values. How we interpret them depends on the context and conventions we have established. This appendix describes several standard representations based on “0” and “1” that are used in computer systems.

A *bit* is a single digit that can have two values, 0 or 1. A *bit-vector* is an ordered list of bits, $\vec{A} = \langle a_{n-1} \dots a_1 a_0 \rangle$. The number of bits, n, is the *length* or *width* of the bit vector. A bit-vector of length n has 2^n possible values. How these values are interpreted depends on the *type* we associate with the vector.

B.1 Unsigned Integer

The most common interpretation of a bit-vector is as the integer value it represents in radix 2 (more familiarly base 2). That is, $\vec{A} = \langle a_{n-1} \dots a_1 a_0 \rangle$ is associated with the integer value

$$I(\vec{A}) = a_{n-1} * 2^{n-1} + a_{n-2} * 2^{n-2} + \cdots + a_1 * 2^1 + a_0 * 2^0$$

For example, the bit vector $\langle 00011010 \rangle$ represents the decimal value $0 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 0 + 16 + 8 + 0 + 2 + 0 = 26$. The leftmost bit is the most significant. As an unsigned integer, a bit-vector of length n represents a non-negative integer in the range 0 to $2^n - 1$.[1] Below are the values for $n = 8$.

$I(\vec{A})$	$\vec{A}$
255	11111111
254	11111110
253	11111101
.	. . . .
4	00000100
3	00000011

[1] The integer value 0 is neither positive nor negative.

2	00000010
1	00000001
0	00000000

B.2 2's Complement Integer

In 2's Complement, the integers in the range -2^{n-1} through $2^{n-1}-1$ are represented using the following formula:

$$T(\vec{A}) = \begin{cases} I(\vec{A}) & \text{if } a_{n-1} = 0 \\ I(\vec{A}) - 2^n & \text{if } a_{n-1} = 1 \end{cases}$$

The top half of the unsigned range is used for the negative integers. The leftmost bit, a_{n-1} of $\vec{A}$ tells us whether it is negative or not. For example with $n = 8$, 255 is now -1, 254 is now -2, ..., and 128 is now -128.

$T(\vec{A})$	$I(\vec{A})$	$\vec{A}$
127	127	01111111
126	126	01111110
125	125	01111101
.	.	....
2	2	00000010
1	1	00000001
0	0	00000000
-1	255	11111111
-2	254	11111110
-3	253	11111101
.		....
-128	128	10000000

As before, to find the n-bit vector corresponding to an integer $N \geq 0$, we use the base 2 representation for N. To represent $-N < 0$ we find the base 2 representation of $2^n - N > 0$. For example, for $n = 8$ and $N = 26$ we have $2^8 - 26 = 256 - 26 = 230$ and the base 2 representation of 230 is $\langle 11100110 \rangle$. Hence $\langle 11100110 \rangle$ represents -26 in 2's complement. This may seem complicated, but there is a simpler way to find the 2's complement vector corresponding to $-N$. It's based on observing that 2^n is the n-bit vector of all 1's plus the vector corresponding to 1:

$$2^n = (2^n - 1) + 1 = I(\langle 111 \ldots 11 \rangle) + I(\langle 000 \ldots 01 \rangle).$$

To find the 2's complement representation of $-N$ we are calculating

$$2^n - N = (2^n - 1) + 1 - N = I(\langle 111 \ldots 11 \rangle) - N + I(\langle 000 \ldots 01 \rangle)$$

If $\vec{A}$ is the base 2 representation of N then

$$I(\langle 111 \ldots 11 \rangle) - N = I(\langle 111 \ldots 11 \rangle - \langle a_{n-1} \ldots a_1 a_0 \rangle) = \langle \overline{a}_{n-1} \ldots \overline{a}_1 \overline{a}_0 \rangle = \overline{\vec{A}}$$

Subtracting $\vec{A}$ from $\langle 111 \ldots 11 \rangle$ results in the complement of $\vec{A}$. In our example for -26, the base 2 representation of 26 is $\langle 00011010 \rangle$, and its complement is $\langle 11100101 \rangle$. Adding 1 to $\langle 11100101 \rangle$ gives $\langle 11100110 \rangle$ as before. So to find the 2's complement vector corresponding to $-N$, we can complement the vector corresponding to N and add 1. This works even for $N < 0$! The big advantage of using 2's complement is that all arithmetic operations are the same as with unsigned integers. We only need to consider signs to detect overflow. Counters are the same regardless of whether their output is considered an unsigned or signed integer. Another advantage is that two signed integers with the same sign can be compared in the same manner as unsigned integers. That is, if $\vec{A}$ and $\vec{B}$ have the same leftmost bit, then $T(\vec{A}) < T(\vec{B})$ if and only if $I(\vec{A}) < I(\vec{B})$.

B.3 ASCII Character

The ASCII code associates the characters of the alphabet with 8-bit values. Although arithmetic is not usually performed with characters, their values are compared to determine the alphabetic order of words. The upper case characters "A" to "Z" run sequentially through the 8-bit values from 65 to 90, while the lower case "a" to "z" are from 97 through 122. The difference between the upper case and lower case versions of a letter is exactly 32. This conveniently allows us to compare the alphabetic order of two letters, regardless of case, by ignoring the sixth bit (a_5) of their binary code. The numeric characters, "0" to "9" are also conveniently sequential starting at 48. The ASCII code is now the character code used almost everywhere. The EBCDIC code (from IBM) preceded ASCII but is now seldom used except in legacy data/systems.

Appendix C

Boolean Algebra Identities

This appendix contains the proofs of several boolean algebra identities. Each proof relies only on the laws of boolean algebras or identities earlier in the list.

Idempotent

$a + a \quad = \quad a$

Proof:

$$\begin{aligned} a + a &= (a + a) * 1 && \text{(Identity)} \\ &= (a + a) * (a + \overline{a}) && \text{(Complement)} \\ &= a + (a * \overline{a}) && \text{(Distributive)} \\ &= a + 0 && \text{(Complement)} \\ &= a && \text{(Identity)} \end{aligned}$$

$a * a \quad = \quad a$

Proof:

$$\begin{aligned} a * a &= (a * a) + 0 && \text{(Identity)} \\ &= (a * a) + (a * \overline{a}) && \text{(Complement)} \\ &= a * (a + \overline{a}) && \text{(Distributive)} \\ &= a * 1 && \text{(Complement)} \\ &= a && \text{(Identity)} \end{aligned}$$

Domination

$a + 1 \quad = \quad 1$

Proof:

$$\begin{aligned} a + 1 &= (a + 1) * 1 && \text{(Identity)} \\ &= (a + 1) * (a + \overline{a}) && \text{(Complements)} \\ &= a + (1 * \overline{a}) && \text{(Distributive)} \\ &= a + \overline{a} && \text{(Identity)} \\ &= 1 && \text{(Complements)} \end{aligned}$$

$a * 0 \quad = \quad 0$

Proof:

$$\begin{aligned} a * 0 &= a * 0 + 0 && \text{(Identity)} \\ &= a * 0 + a * \overline{a} && \text{(Complements)} \\ &= a * (0 + \overline{a}) && \text{(Distributive)} \\ &= a * \overline{a} && \text{(Identity)} \\ &= 0 && \text{(Complements)} \end{aligned}$$

Absorption

$a + a * b \quad = \quad a$

Proof:

$$\begin{aligned} a + a * b &= (a * 1) + a * b && \text{(Identity)} \\ &= a * (1 + b) && \text{(Distributive)} \\ &= a * 1 && \text{(Domination)} \\ &= a && \text{(Identity)} \end{aligned}$$

$a * (a + b) \quad = \quad a$

Proof:

$$\begin{aligned} a * (a + b) &= (a + 0) * (a + b) && \text{(Idenity)} \\ &= a + (0 * b) && \text{(Distributive)} \\ &= a + 0 && \text{(Domination)} \\ &= a && \text{(Identity)} \end{aligned}$$

Simplification

$$a + \overline{a} * b = a + b$$

Proof:

$$\begin{aligned} a + \overline{a} * b &= (a + \overline{a}) * (a + b) && \text{(Distributive)} \\ &= 1 * (a + b) && \text{(Complements)} \\ &= a + b && \text{(Identity)} \end{aligned}$$

$$a * (\overline{a} + b) = a * b$$

Proof:

$$\begin{aligned} a * (\overline{a} + b) &= a * \overline{a} + a * b && \text{(Distributive)} \\ &= 0 + a * b && \text{(Complements)} \\ &= a * b && \text{(Identity)} \end{aligned}$$

Uniqueness of complements

If $a + b = 1$ and $a * b = 0$ then $b = \overline{a}$.

Proof:

Suppose $a + b = 1$ and $a * b = 0$ then

$$\begin{aligned} b &= 1 * b && \text{(Identity)} \\ &= (a + \overline{a}) * b && \text{(Complements)} \\ &= a * b + \overline{a} * b && \text{(Distributive)} \\ &= 0 + \overline{a} * b && \text{(Hypothesis)} \\ &= \overline{a} * a + \overline{a} * b && \text{(Complements)} \\ &= \overline{a} * (a + b) && \text{(Distributive)} \\ &= \overline{a} * 1 && \text{(Hypothesis)} \\ &= \overline{a} && \text{(Identity)} \end{aligned}$$

Involution

$\overline{\overline{a}} = a$

Proof: By the Complement Laws we know $a + \overline{a} = 1$ and $a * \overline{a} = 0$. So by the Uniqueness of Complements, a must be the complement of $\overline{a}$: that is, $\overline{\overline{a}} = a$.

DeMorgan's Laws

$$\overline{(a + b)} = \overline{a} * \overline{b}$$

Proof:

$$\begin{aligned} \overline{(a + b)} &= \overline{(a + b)} * 1 && \text{(Identity)} \\ &= \overline{(a + b)} * (a + 1) && \text{(Domination)} \\ &= \overline{(a + b)} * (a + (b + \overline{b})) && \text{(Complements)} \\ &= \overline{(a + b)} * (b + (a + \overline{b})) && \text{(Associative)} \\ &= \overline{(a + b)} * (b + (a + \overline{a} * \overline{b})) && \text{(Simplification)} \\ &= \overline{(a + b)} * ((a + b) + \overline{a} * \overline{b}) && \text{(Associative)} \\ &= \overline{(a + b)} * (\overline{a} * \overline{b}) && \text{(Simplification)} \\ &= \overline{(a + b)} * (\overline{a} * \overline{b}) + 0 && \text{(Identity)} \\ &= \overline{(a + b)} * (\overline{a} * \overline{b}) + (0 + 0) && \text{(Identity or Idempotent)} \\ &= \overline{(a + b)} * (\overline{a} * \overline{b}) + (0 * \overline{b} + 0 * \overline{a}) && \text{(Domination)} \\ &= \overline{(a + b)} * (\overline{a} * \overline{b}) + ((a * \overline{a}) * \overline{b} + (b * \overline{b}) * \overline{a}) && \text{(Complements)} \\ &= \overline{(a + b)} * (\overline{a} * \overline{b}) + ((a * (\overline{a} * \overline{b}) + b * (\overline{b} * \overline{a})) && \text{(Associative)} \\ &= \overline{(a + b)} * (\overline{a} * \overline{b}) + ((a * (\overline{a} * \overline{b}) + b * (\overline{a} * \overline{b})) && \text{(Commutative)} \\ &= \overline{(a + b)} * (\overline{a} * \overline{b}) + (a + b) * (\overline{a} * \overline{b}) && \text{(Distributive)} \\ &= (\overline{(a + b)} + (a + b)) * (\overline{a} * \overline{b}) && \text{(Distributive)} \\ &= 1 * (\overline{a} * \overline{b}) && \text{(Complements)} \\ &= \overline{a} * \overline{b} && \text{(Identity)} \end{aligned}$$

Rather than applying duality to the proof above for the second DeMorgan Law, below we use uniqueness of complements. Note that the proof above essentially uses the same steps for establishing the uniqueness of complements.

$\overline{(a * b)} = \overline{a} + \overline{b}$

Proof:

$$\begin{aligned}
(\overline{a} + \overline{b}) + (a * b) &= (\overline{a} + (a * b))\overline{b} && \text{(Associative, Commutative)} \\
&= (\overline{a} + b) + \overline{b} && \text{(Simplification)} \\
&= \overline{a} + (b + \overline{b}) && \text{(Associative)} \\
&= \overline{a} + 1 && \text{(Complements)} \\
&= 1 && \text{(Domination)}
\end{aligned}$$

$$\begin{aligned}
(\overline{a} + \overline{b}) * (a * b) &= (\overline{a}) * (a * b) + (\overline{b}) * (a * b) && \text{(Distributive)} \\
&= (a * \overline{a}) * b + a * (b * \overline{b}) && \text{(Associative, Commutative)} \\
&= 0 * b + a * 0 && \text{(Complements)} \\
&= 0 + 0 && \text{(Domination)} \\
&= 0 && \text{(Identity)}
\end{aligned}$$

By the uniqueness of complements, $(\overline{a} + \overline{b})$ is the complement of $(a * b)$ hence $\overline{a} + \overline{b} = \overline{(a * b)}$.

C.1 Duality

You may have noticed that the laws of boolean algebra laws come in pairs that have a similar structure. Within each pair, exchanging the +'s and ∗'s, and the 0's and 1's in one equation results in the other equation. This property extends to the identities proved in this appendix. The proofs themselves can be obtained by this transformation.

The *dual* of an expression E is obtained by swapping the two operations $*$ and $+$ as well as 0 and 1. The order of the operations in the original expression should remain the same so parentheses may be needed. For example, the dual of $a + b * c$ is $a * (b + c)$ as opposed to $a * b + c$. Here are more examples of expressions and their duals:

E	dual of E
a	a
$0 + \overline{a}$	$1 * \overline{a}$
$a + bc$	$a * (b + c)$
$ab + \overline{a}\overline{b}$	$(a + b) * (\overline{a} + \overline{b})$

An expression and its dual are not, in general, equivalent. The dual of an expression can be equivalent to the original, the complement of the original, or neither. One result that does follow is:

> If E_1 and E_2 are equivalent expressions, then their dual expressions are equivalent to each other.

For example, after showing that

$$a + \overline{a} * b = a + b$$

we can also conclude that

$$a * (\overline{a} + b) = a * b$$

rather than providing a proof that uses the dual of the law at every step.

Bonus material

As mentioned, the Associative Law can be derived from the other 4 axioms of Boolean Algebras (the Commutative, Identity, Complements, and Distributive Laws). Below is a proof for the disjunctive operator. Note that the Domination, Absorption, and Simplification Identities were all shown without using the Associative Law. The equivalent law for the conjunctive operator follows from the principle of duality.

Associativity
$(a+b)+c = a+(b+c)$
Proof:

$$\begin{array}{rcll}
(a+b)+c & = & a*((a+b)+c)+((a+b)+c) & \text{(Absorption)} \\
 & = & (a*(a+b)+a*c)+((a+b)+c) & \text{(Distributive)} \\
 & = & (a+a*c)+((a+b)+c) & \text{(Absorption)} \\
 & = & a+((a+b)+c) & \text{(Absorption)} \\
 & = & a+\overline{a}*((a+b)+c) & \text{(Simplification)} \\
 & = & a+(\overline{a}*(a+b)+\overline{a}*c) & \text{(Distributive)} \\
 & = & a+(\overline{a}*b+\overline{a}*c) & \text{(Simplification)} \\
 & = & a+\overline{a}*(b+c) & \text{(Distributive)} \\
 & = & a+(b+c) & \text{(Simplification)}
\end{array}$$

Appendix D

Bibliography

[1] George Boole. *An Investigation of the Laws of Thought: On Which Are Founded the Mathematical Theories of Logic and Probabilities.* Cambridge Library Collection - Mathematics. Cambridge University Press, 2009. (Originally published in 1854.).

[2] M. Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5):593–599, 1953.

[3] Edward V. Huntington. Sets of independent postulates for the algebra of logic. *Transactions of the American Mathematical Society*, 5(3):288–309, 1904.

[4] Robert K Brayton, Gary D Hachtel, Lane A Hemachandra, A Richard Newton, and Alberto Luigi M Sangiovanni-Vincentelli. A comparison of logic minimization strategies using espresso: An apl program package for partitioned logic minimization. In *Proceedings of the International Symposium on Circuits and Systems*, pages 42–48, 1982.

[5] Robert K Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R Wang. Mis: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(6):1062–1081, 1987.

[6] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.