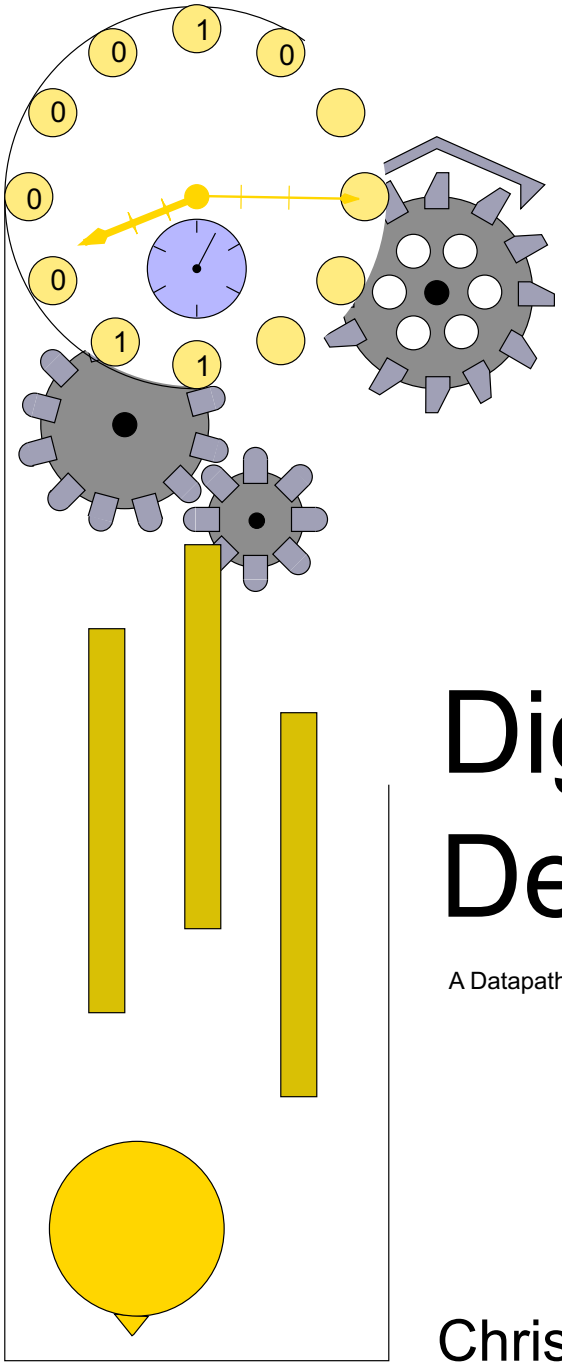


Digital Design

A Datapath and Control Approach

Chris Coulston

May 13, 2025

This document was prepared with LaTeX.

Contents

List of Processes

List of Basic Building Blocks

Chapter 1

Numbering Systems

The goal of this textbook is to teach students how to design digital systems. To understand what this means, it is necessary to understand the behavior of a digital system. A digital system is a device which receives binary numbers as input and generates binary numbers as output. A binary number is a number composed of bits, or binary digits. A bit is equal to 0 or 1. Generally, bits are represented by voltages, a 0 by a ground potential and a 1 by a 3.3v or 5v potential. However, for most of this text, the physical representation of bits will take a back seat to the the logical representation. Figure 1.1 shows a digital system with three bits of input and two bits of output.

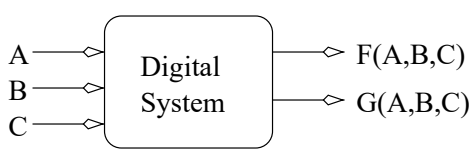


Figure 1.1: An abstract digital system with three bits of input labeled A,B,C and two bits of output, F(A,B,C) and G(A,B,C).

The letters A, B, C in Figure 1.1 represent *Boolean variables*, variables which are only allowed to assume the values 0 or 1. The notation $F(A, B, C)$ means that the output depends on the values of A, B, C.

Unfortunately, there is a disconnect between the normal way of describing quantities using the digits $0 \ldots 9$ and that used by digital systems using the bits $0, 1$. The study of digital systems starts by describing the numbering system used to describe decimal numbers and binary numbers. These systems are *positional numbering systems* - the position of a digit in a number determines its significance. The familiar decimal numbering system is used to illustrate the main concepts in positional numbering system.

1.1 Decimal

The goal of any numbering system (this includes both decimal and binary) is to represent quantities using a fixed set of symbols. One feature which distinguishes numbering systems is

the number of symbols used to represent quantities, the numbering system's *base* .

Decimal is a base-10 numbering system where quantities are expressed using 10 symbols $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. To explicitly denote a number's base, include the base as a subscript after the number. For example, the number of days in a year could be represented as 365_{10}.

Each digit in a number represents a quantity which depends on the digit, the base, and the position of the digit. The value of a digit is determined by the formula $d * b^i$ where d is the digit, i is the position of the digit and b is the base. The position of the digit immediately to the left of the decimal point is 0, every other digit is indexed starting from this position. The value of a multidigit number is the sum of the values of the digits, $\sum_{i=0}^{N} d_i * b^i$, where N is the number of digits in the number. Thus 365_{10} is interpreted as

$$3 * 10^2 + 6 * 10^1 + 5 * 10^0$$

The leftmost digit is called the most significant digit and the rightmost digit is called the least significant digit.

1.2 Binary

Binary is a base-2 numbering system where quantities are expressed using two symbols $\{0, 1\}$. When verbally communicating a binary value like 101_2, do not say "one hundred and one base two". The term *hundred* is a decimal concept and implies the quantity being described is decimal, contradicting the "base two". Instead, say "one zero one base two", with the "base two" part being optional. Enough about the nomenclature used to communicate binary numbers, how to represent values in binary? Since binary is a positional numbering system, then apply the formula $\sum_{i=0}^{N} d_i * b^i$.

Binary to Decimal

The value of 101_2 is interpreted as

$$101_2 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 1 * 4_{10} + 0 * 2_{10} + 1 * 1_{10} = 4_{10} + 0_{10} + 1_{10} = 5_{10}$$

Application of the positional numbering formula to a binary number converts it into a decimal number. As in the decimal case there are special names for two of the bits. The leftmost bit is called the most significant bit (MSB) and the rightmost bit is called the least significant bit (LSB). As a final example, convert 1101_2 to decimal.

$$1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 1 * 8_{10} + 1 * 4_{10} + 0 * 2_{10} + 1 * 1_{10} = 8_{10} + 4_{10} + 0_{10} + 1_{10} = 13_{10}$$

Decimal to Binary

In order to provide inputs to a digital system, real world values represented in decimal need to be converted into binary. While there are several ways to perform this conversion, it makes sense to adapt a familiar procedure, the binary to decimal conversion. The key idea is to represent the decimal number as the sum of distinct powers-of-two. To assist in this procedure, use a power-of-two table.

i	0	1	2	3	4	5	6	7	8	9
2^i	1	2	4	8	16	32	64	128	256	512

The power-of-two table lists the index i and its value when raised as the exponent of 2. This table is used in the following 3-step decimal to binary conversion procedure.

Step 1 Find the largest power of two less than or equal to the number to convert.

Step 2 Subtract this power of two from the number being converted. This is the remaining number to convert.

Step 3 If the reminder is not equal to 0, go to step 1; otherwise stop.

The set of values found in Step 1 are the distinct powers-of-two that when added together equal the number to be converted. The conversion is completed by putting the sum into the positional numbering notation. The procedure is now applied to convert 13_{10} into binary.

Step	Action
1	The largest power-of-two less than or equal to 13 is 8.
2	The remaining number to convert is 13-8=5.
3	The new remainder is not 0.
1	The largest power-of-two less than or equal to 5 is 4.
2	The remaining number to convert is 5-4=1.
3	The new remainder is not 0.
1	The largest power-of-two less than or equal to 1 is 1.
2	The remaining number to convert is 1-1=0.
3	The new remainder is 0 so the conversion is complete.

Thus, $13_{10} = 8 + 4 + 1$. This derivation can also be expressed in the positional numbering notation as follows.

$$13_{10} = 8_{10} + 4_{10} + 1_{10} = 1 * 2^3 + 1 * 2^2 + 1 * 2^0$$

At this point the "missing" powers of two are included in the summation by setting their coefficients to 0. In the final step, all the coefficients are stripped off to form the binary number. Continuing with the previous example,

$$13_{10} = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 1101_2$$

Notice that this derivation is the exact reverse of the binary to decimal conversion shown on page 2.

What range of values can be described by N bits? Clearly, the smallest number to be represented is 0. To determine the largest number calculate how many different binary numbers can be formed with N bits. Each bit can be written in two different ways, 0 or 1. Since these choices are independent events, then the total number of possible outcomes is the product of the individual events. Hence, the number of ways to arrange N bits is equal to $2 * 2 * \ldots * 2$ (N times) which is equal to 2^N. Thus, N bits can be arranged in 2^N different ways. Since 0 is the smallest binary number then the maximum binary number is $2^N - 1$. The range of an N-bit binary number can be represented as $[0, 2^N - 1]$, where the [and] symbols mean that values 0 and $2^N - 1$ are included in the range.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Table 1.1: The first 16 counting numbers represented in decimal, binary, and hexadecimal.

1.3 Hexadecimal

Hexadecimal is a base-16 numbering system where quantities are expressed using 16 symbols $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$. The symbols $\{A \ldots F\}$ represent quantities just like the symbols $\{0 \ldots 9\}$. The symbol A represents the quantity 10, B 11, C 12, D 13, E 14, and F 15. So if humans had adopted the hexadecimal numbering system instead of decimal you might say to a friend, "I had A friends over for a party last night" meaning that 10 people showed up. To show how hexadecimal is used as a shorthand for binary, consider the conversion of hexadecimal numbers into binary.

Hexadecimal to Binary

To convert a number from hexadecimal to binary, *unpack* it. That is, each hexadecimal digit is converted into a 4-bit binary representation. This conversion is possible because four bits exactly represents every hexadecimal digit as shown in Table 1.1.

In order to convert $1DAD_{16}$ to binary, replace each hexadecimal digit with its binary counterpart by consulting Table 1.1. For example $1DAD_{16} = 0001\ 1101\ 1010\ 1101_2$. The spaces between the binary numbers are included to make reading the number easier.

Binary to Hexadecimal

The above procedure can be reversed to convert binary numbers into hexadecimal. Group the bits into sets of four, starting at the least significant bit, then convert each set of four bits into its corresponding hexadecimal digit in Table 1.1. If there are not four digits in the most significant grouping, then just add 0s to make a grouping of three – that is pad the number with zeros. For example, convert 1110101011_2 into hexadecimal. $1110101011_2 = 0011\ 1010\ 1011 = 3AB_{16}$.

To understand why this conversion works, consider the binary number 1110101011_2. Start by writing down this number as the sum of powers of two, group the powers of two into sets

of 4. The remaining steps are shown in the derivation below.

$$
\begin{aligned}
&1110101011_2 = \\
&1*2^9+1*2^8+1*2^7+0*2^6+1*2^5+0*2^4+1*2^3+0*2^2+1*2^1+1*2^0 = \\
&2^8(0*2^3+0*2^2+1*2^1+1*2^0)+2^4(1*2^3+0*2^2+1*2^1+0*2^0)+2^0*(1*2^3+0*2^2+1*2^1+1*2^0) = \\
&2^8(0011_2)+2^4(1010_2)+2^0(1011_2) = \\
&2^{4*2}(0011_2)+2^{4*1}(1010_2)+2^{4*0}(1011_2) = \\
&16^2(0011_2)+16^1(1010_2)+16^0*(1011_2) = \\
&16^2(3_{16})+16^1(A_{16})+16^0*(B_{16}) = \\
&3AB_{16}
\end{aligned}
$$

1.4 Binary Addition

The fact that the addition of binary numbers is similar to the addition of decimal numbers should not come as a surprise as both are positional numbering systems. However, when adding binary numbers with a digital system, it is possible that the result may exceed the digital system's capacity because the digital system can only accommodate a finite number of bit positions. The number of bits to be simultaneously manipulated by a digital system is refereed to as its *word size* . A digital system with a word size of N-bits can represent binary numbers in the range $[0, 2^N - 1]$. *Overflow* occurs when a digital system is forced to represent a value outside the range of its word size.

The process of adding binary numbers is exactly the same as adding decimal numbers: Start in the LSB and work towards the MSB. Each of the bit-positions is called a *bit-slice* . At each bit-slice, the two bits of the sum are added together along with the carry-in generated in the previous bit-slice. This addition in a bit-slice will generate one bit of sum and possibly one bit of carry to the next bit-slice. The addition of bits must be performed in base-2, hence the results may look strange. For example, $1_2 + 1_2 = 10_2$. In this case the sum-bit equals 0 and the carry-bit equals 1. Now, consider a more complex problem, adding $3 + 2$ in binary, assuming a word size of four bits.

		1		
3	0	0	1	1
+2	0	0	1	0
5	0	1	0	1

Notice, one of the additions produced a carry which is propagated to the next significant bit position. The next example demonstrates an addition where the result is larger than the word size can accommodate.

	1	1	1	1	1
13		1	1	0	1
+7		0	1	1	1
20	1	0	1	0	0

The decimal result, 20, cannot be represented in four bits, hence the addition produced overflow. When adding binary numbers, overflow occurs whenever there is a carry-out from the most significant bit-slice. That is, the result requires more bits than are available in the word size.

After being introduced to binary numbering it might be tempting to look at all collections of bits as being binary numbers. In truth, a collection of bits has no implicit meaning. The sequence of bits, 0101, could just as easily represent the value 5 as it could represent the intensity of red on a display. A collection of bits gets its meaning from the interpretation used. When bits are used to represent integer quantities, two main interpretations, binary numbering

and 2's complement, predominate.

1.5 Negative Numbers

The binary numbering systems is often called an *unsigned* numbering representation. The term unsigned arises from the fact that there is no need to write a sign symbol in front of a binary number because all binary numbers are positive – the positive sign is implicit. A *signed* numbering representation, 2's complement, is capable of representing both positive and negative numbers.

Like binary numbering, 2's-complement numbers exist within the confines of a word size. One way to determine the 2's-complement representation of a decimal number x, is to write down the binary representation for the quantity $2^N + x$ using N bits, where N is the word size. For example, assuming a word size of four bits, determine the 2's-complement representation for 6. To do this, compute $2^N + x = 2^4 + 6 = 16 + 6 = 22 = 10110_2$. Taking the least significant four bits yields 0110.

There are two points to note. First, this representation is the same as in binary numbering. Second, the 2's-complement value is written without a subscript 2, because it is not a binary number. Now consider the 2's-complement representation of a negative number.

Assuming a word size of four bits, determine the 2's-complement representation for -6. Compute $2^N + x = 2^4 - 6 = 16 - 6 = 10 = 01010_2$. Taking the least significant four bits yields 1010.

To determine the decimal value of a 2's-complement number, inspect its MSB. If the MSB is 0, then the number is positive, hence can be interpreted as a binary number. If the MSB is 1, then $2^N + x$ must be solved for x. There is, however an easier way to approach this problem.

Negating a 2's-complement number will mean changing the sign of the underlying decimal representation. The negation of a 2's-complement number, x, can be formed by flipping all the bits of x and then adding 1. For example, take the complement of the 4-bit 2's-complement number $x = 0110$ which equals 6. Flipping all the bits of x yields 1001. Adding 1 to this yields 1010, which was previously shown to equal -6.

This technique aids in interpreting negative 2's-complement numbers as follows. Given a 2's-complement number that is negative, form its negation, convert that to decimal, then stick a negative sign in front of the decimal representation. For example, determine the decimal representation for the 4-bit 2's-complement quantity 1010. Since the MSB is 1, this 2's-complement number represents a negative quantity. Flipping the bits, 0101, then adding 1, results in 0110. This is the representation for 6, so the original 2's-complement number 1010 represent -6.

Figure 1.2 shows every combination of four bits and their associated 2's-complement representation.

Clearly, half of the numbers in Figure 1.2 have a leading 0 as their MSB, and are positive; 0 is considered a positive number. The other half of the numbers have their MSB equal to 1 and are negative. Since 0 is considered a positive number, the largest negative number is 1 larger than the largest positive number. Given a word size of N bits the range of 2's-complement numbers is $[-2^{N-1}, 2^{N-1} - 1]$.

2's-complement numbers are added in the same way that binary numbers are added as shown in the following three problems.

6	0	1	1	0
+ -7	1	0	0	1
= -1	1	1	1	1

	1	1	1		
3		0	0	1	1
+ -2		1	1	1	0
= 1		0	0	0	1

	1	1		
6	0	1	1	0
+ 6	0	1	1	0
= 12	1	1	0	0

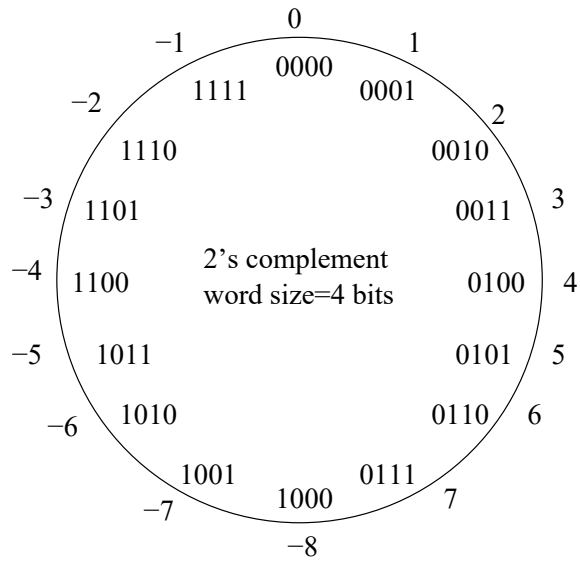


Figure 1.2: All possible combinations of four bits and their 2's-complement interpretations.

The first example, $6 + -7$, shows the addition process working correctly when the operands have different signs. The other two problems illustrate the need for a new definition of overflow for 2's-complement numbers. The general rule for overflow in 2's complement is, if the carry-in and carry-out into and out from the MSB are not equal, then overflow has occurred. The carry-out from the MSB is always thrown away. For example, the carry-in and carry-out of the MSB in second problem are both 1, hence the result is valid. In the third problem, the carry in to the MSB is 1 and the carry-out from the MSB is 0, hence overflow has occurred. This result should not be a surprise because the expected result, 12, cannot be represented as a 4-bit 2's-complement number.

Taking the complement of a 2's-complement number allows subtraction problems to be converted into addition problems. For example, the subtraction problem $3 - 2$ can be rewritten as $3 + (-2)$.

						1	1	1		
3	0	0	1	1	Becomes	3	0	0	1	1
- +2	0	0	1	0		+ -2	1	1	1	0
						= 1	0	0	0	1

Situations will arise which require increasing the number of bits required to represent a number while retaining the value of the number. For example, imagine having a 4-bit binary number that must be stored in a device that holds eight bits. How can this be done while preserving the magnitude of the number? For binary numbers, the answer is easy, add 4 leading zeros, an operation called *padding with 0s*.

For 2's-complement numbers this solution will not work. For example, consider the 4-bit 2's-complement representation for -1 (1111) that needs to be store in a 8-bit device. Adding 4 leading 0s will change the value of the number to 00001111, the value 15. The solution, in this case, is to pad with 1s, yielding 11111111, which represents -1 in 8-bit 2's complement. As with the binary numbering example, positive 2's-complement numbers can be padded with

4-bit 2's complement	8-bit 2's complement
1110=-2	11111110=-2
1010=-6	11111110=-6
1000=-8	11111000=-8
0011=3	00000011=3
0111=7	00000111=7

Table 1.2: Five examples showing how to sign-extend a 4-bit 2's-complement number.

0s and still retain their value. The general rule for padding in 2's complement is called *sign extension*; and involves copying the MSB to fill in the needed space. Table 1.2 shows five examples of sign extension on 2-bit 2's-complement numbers.

1.6 Exercises

1. **(1 pt. each)** Syllabus:

   a) What is the late penalty for homework?

   b) True or False: Calculators can be used during exams.

   c) True of False: University ID is required during exams.

   d) What is my thesis regarding grades?

   e) Bob L. Student has the following grades. Determine his final overall course percentage and grade.

Component	Percentage
Homework	60%
Exam 1	90%
Exam 2	80%
Final	70%

   f) How should you prepare for the 43rd lecture?

2. **(1 pt. each)** Convert the following numbers to decimal. Show work, or receive 1/2 credit.

   a) 100_2

   b) 1000_2

   c) 10000_2

   d) 100000_2

   e) 111111_2

   f) 1000100101000101_2

   g) $3EA_{16}$

3. **(1 pt. each)** Convert the following number to binary. Show work, or receive 1/2 credit.

   a) 44_{16}

   b) 44_{10}

   c) 1023_{10}

4. **(1 pt. each)** Convert the following number to hex. Show work, or receive 1/2 credit.

   a) 101011101_2

   b) 77_{10}

5. **(2 pts. each)** Toughies:

   a) Convert 123_5 to base-12

   b) Convert 789_{12} to base-5

   c) What is the largest base-10 quantity that can be represented using 5 digits in base 12?

6. **(1 pt. each)** Perform the following additions, assume a word size of four bits. Determine if overflow occurs.

a) $0110_2 + 0101_2$
b) $0010_2 + 0110_2$
c) $0111_2 + 0011_2$
d) $0010_2 + 0101_2$
e) $0010_2 + 1010_2$
f) $0101_2 + 1011_2$
g) $0011_2 + 1001_2$

Chapter 2

Representations of Logical Functions

A digital system starts its life deep inside an engineer's mind. In the beginning it is an abstract device; its very far removed from reality. In order to realize the digital system, the engineer undertakes the design process; a series of refinements to bring the digital system closer and closer to a real working system. This iterative approach is necessary in complex designs because going straight to a final implementation is overwhelming, error-prone, difficult to modify, and difficult to debug and test. The goal in breaking the design process into steps is to allow the proper specification of the digital system using a "language" which is easy to understand and modify, and then to show how to implement this specification using real circuit elements.

Four refinements in the design of digital systems are considered. Each of these refinements define the input, output, and behavior of the digital system.

Word Statement - A written description of the expected input, output, and behavior of the digital system.

Truth Table - A listing of every possible input to the digital system and the corresponding output.

Symbolic - A symbolic (math-like) description of the output(s) as a function of the input variable(s).

Circuit Diagram - A pictorial representation of the physical interconnections of the circuit elements.

When designing a digital system each of these representations should describe the same system, just in different ways. The most abstract representation of a digital system is the word statement. Word statements are notoriously ambiguous and have no standardized structure. However, these shortcomings are the strengths of this representation. The great expressibility of language allows very complex processes to be captured in brief statements. Hence, ideas can be quickly refined without having to expend much engineering effort. Also, because word statements have no prescribed form, there is a great deal of flexibility in structuring a word statements. Because of its potential ambiguity and unclear boundaries, the word statement's step in the design process diagram shown in Figure 2.1 is drawn inside a puffy cloud.

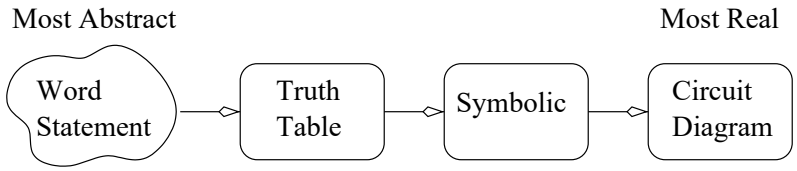


Figure 2.1: The steps in the design of a digital system.

Each of the four design stages shown in Figure 2.1 is a description of a digital system. The arrows in Figure 2.1 are transformations described in this chapter.

After the word statement, all subsequent stages of the design process are unambiguous, they have one clearly understood interpretation. The first refinement of a word statement is a truth table. The truth table strikingly illustrates the difference between digital and analog circuits. When working with analog phenomena, like the AC voltage from a wall outlet, it is impossible to list all possible values of the voltage, because an infinite number of potential values exists. However, when working with digital phenomena, each signal can have only one of two possible values, making it quite easy to enumerate them all. When working with a digital system with three bits of inputs, like that shown in Figure 1.1, there are only eight different combinations of the inputs. For each of these input combinations, a truth table specifies what the output should be.

The symbolic form is a set of equations written in Boolean Algebra. These equations look similar to those encountered in an algebra class. However, instead of operations like addition and multiplication, Boolean Algebra uses the operations AND, OR, and NOT. These operations have hardware counterparts - real physical circuits which "compute" their values. A circuit diagram shows how these hardware components are interconnected to realize the digital system. As real devices, these circuits represent the bit values, 0 and 1 as voltages. Typically, logic 1 is represented by a high voltage level (5v) and logic 0 is represented by a low voltage level (0v).

This text takes a bottom-up approach in designing digital systems. The design process starts with the most fundamental digital hardware components and shows how they are interconnected to form basic building blocks. These building blocks are then arranged into datapath and control circuit. Thus, all the digital circuits studies in this text will be built from a small pallet of fundamental digital hardware components, called the elementary logical functions.

2.1 Elementary Logical Functions

Elementary logical functions get their name because they are the fundamental (elementary) building blocks of digital design, they use the logical operators like AND, OR, and NOT (logical), and they describe a transformation (function) from input to output. For each elementary logical function, its word statement, truth table, symbolic and circuit diagram are presented.

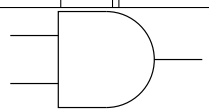
AND

Word Statement	AND is a logical function with two inputs and one output. The output equals 1 only when all the inputs are equal to 1, otherwise the output equals 0.
Symbolic	A*B

Truth Table

A	B	A*B
0	0	0
0	1	0
1	0	0
1	1	1

Circuit Diagram



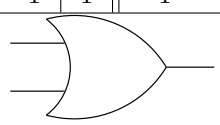
OR

Word Statement	OR is a logical function with two inputs and one output. The output equals 1 when any input is equal to 1, otherwise the output equals 0.
Symbolic	A+B

Truth Table

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

Circuit Diagram



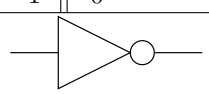
NOT

Word Statement	NOT is a logical function with one input and one output. The output is not equal to the input.
Symbolic	A'

Truth Table

A	A'
0	1
1	0

Circuit Diagram



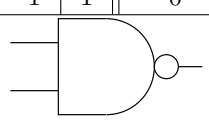
NAND

Word Statement	NAND is a logical function with two inputs and one output. The output equals 1 when any input is equal to 0, otherwise the output equals 0.
Symbolic	(A*B)'

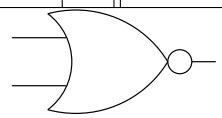
Truth Table

A	B	(A*B)'
0	0	1
0	1	1
1	0	1
1	1	0

Circuit Diagram

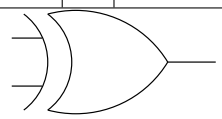


NOR

Word Statement	NOR is a logical function with two inputs and one output. The output equals 0 when any input is equal to 1, otherwise the output equals 0.
Symbolic	(A+B)'
Truth Table	
Circuit Diagram	

A	B	(A+B)'
0	0	1
0	1	0
1	0	0
1	1	0

XOR

Word Statement	XOR is a logical function with two inputs and one output. The output equals 1 when the inputs are different, otherwise the output is equal to 0.
Symbolic	$A \oplus B$
Truth Table	
Circuit Diagram	

A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

The physical elements for the elementary logical functions are typically called *gates*. Thus, the physical circuit for the AND logical function is called an AND gate.

The AND/OR functions can be enlarged to handle more than two inputs because their word statements use words ALL/ANY respectively. Thus a 4-input AND gate will output 1 only when all four inputs equal 1. The circuit diagram for this AND gate would look just like a 2-input AND gate with two additional inputs.

2.2 Conversion Between Representations

Figure 2.2 shows the four representations of a digital system along with the different transformations covered in this section. The arrows are numbered by the process in which the transformation is covered. The design process shown in Figure 2.2 is captured in Steps 7, 6, and 4. The other transformations introduced in this section enable designs to be optimized as well as to explore the relationships between the representations.

2.3 Circuit Diagram to Truth Table

The first transformation is the circuit diagram to truth table. Where the circuit diagram came from is unimportant for now. Figure 2.3 is a circuit diagram with three bits of input labeled A, B, C and one bit of output $F(A, B, C)$ that will be transformed into a truth table.

Before starting with the transformation there are some important observations to make about Figure 2.3.

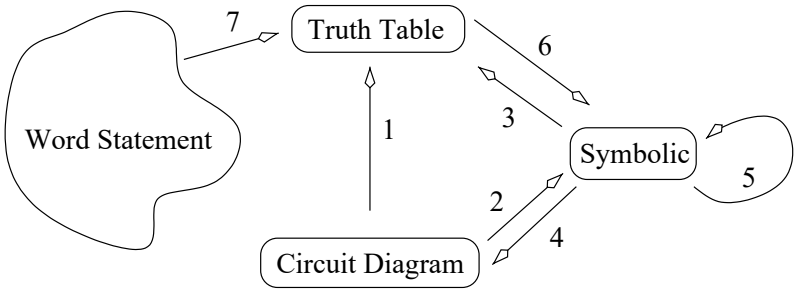


Figure 2.2: The four representations of a logical function. The numbered arrows are the transformations examined in this chapter.

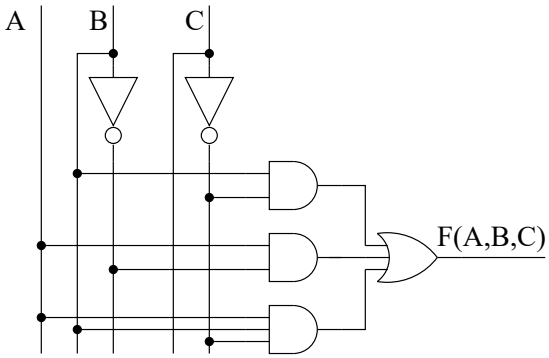


Figure 2.3: A simple 3-input, 1-output circuit.

1. The gates in a circuit may have different numbers of inputs. For example, this circuit contains two AND gates with two bits of input and an AND gate with three bits of input.
2. The lines drawn in the figure represent physical wire.
3. When a voltage is applied to one end of the wire by an external source or by one of the gates, it is assumed to propagate instantaneously to all points on the wire.
4. The black dots on intersecting wires means that the two wires are connected. For example, the vertical wire labeled "A" is connected to the second and third AND gates.
5. Wires that cross one another and do not have a dot are assumed not to be connected. For example, the C signal does not go into the second AND gates.
6. Outputs are never connected directly together, because they could create a short-circuit destroying the participating gates.
7. Inputs are always tied to some signal source, usually the output of a gate.
8. A gate output may drive the inputs of many different gates.

Since the three external inputs in Figure 2.3 do not have sources shown, they must be provided by some external user. The output from a gate can provide input to multiple, different inputs. For example, the output of the NOT gate associated with the C input feeds the first and third AND gates.

Process 2.1: Circuit Diagram to Truth Table

To lend context to the discussion of the conversion process, let's describe this process by converting the circuit diagram in Figure 2.3 into a truth table.

Step 1: Identify the inputs and outputs of the circuit diagram. Any wire which is not being driven by some source is an input and any output which is not driving a source is an output. In the case of Figure 2.3, A, B, C are inputs and $F(A, B, C)$ is an output.

Step 2: Build the shell of the truth table to hold the solution. Remember that, "A truth table is a listing of every possible input to the digital system and the corresponding output." For example $A = 1$, $B = 0$ and $C = 1$ is a possible input to the circuit. All the potential inputs to the circuit could be listed in a *ad-hoc* manner, but a much more organized approach is shown below.

A	B	C	$F(A, B, C)$
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Two good reasons drive this choice of row ordering. First, the columns of the truth table have a strong pattern. The bits of the C-variable column repeat from top to bottom, 0101 . . .. The bits of the B-variable column repeat from top to bottom, two 0s followed by two 1s. The bits of the A-variable column repeat from top to bottom, four 0s followed by four 1s. Using this pattern makes filling a truth table easier. Second, when the A, B, C variables in each row are interpreted as a 3-bit binary number, they form consecutive integers. This approach makes finding a particular input/output much easier.

Step 3: Determine the values in the output column. In order to do this each input is applied, one at a time, to the circuit and determine the output of the circuit. This step is also referred to as evaluating the output of the circuit.

Imagine that the bits are flowing down hill from the inputs to the outputs. A gate computes its output only when the values of all its inputs are known. The output of a gate is determined from the truth tables given on pages 12 through 14. For example, in Figure 2.4A, the input $A = B = C = 0$ is applied to the input. The third AND gate cannot compute its output until it receives the output from the NOT gate associated with the C input. The outputs from each gate are shown on the signals in Figure 2.4B. Due to size constraints, the AND gate inputs are written inside the gate.

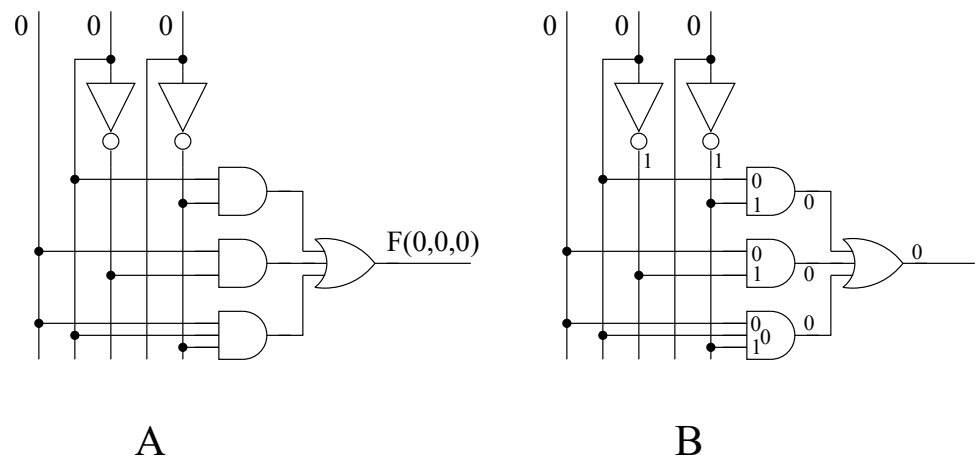


Figure 2.4: A) The circuit diagram from Figure 2.3 with input $A = B = C = 0$. B) The effect of this input shown on the wires.

Figures 2.3 and 2.4 illustrate a subtle notational convention. Why is the output of the circuit shown in Figure 2.3 labeled $F(A, B, C)$ and in Figure 2.4A this same output is labeled $F(0, 0, 0)$? It is because the variables A, B, C in $F(A, B, C)$ are placeholders for the actual values of the variables. Since these variables are given the values $A, B, C = 0, 0, 0$ in Figure 2.4A, this notational change is acknowledged by labeling the output $F(0, 0, 0)$.

Since Figure 2.4B shows that $F(0, 0, 0) = 0$, then a 0 is placed in the top row of the truth table for this function. The remaining seven rows of the truth table are computed using this same method. The resulting truth table is shown below.

A	B	C	$F(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

While this transformation provides an excellent introduction to a variety of important concepts, it is not a practical way to derive the truth table entries. First, it is tedious because the same process must be performed eight times. Second, it is prone to errors because the bits on the figure must be overwritten seven times, once for every row of the table. It turns out that determining the symbolic form for this circuit's output and then determining the truth table is a much less tedious and less error-prone method to determine the truth table from a circuit diagram.

2.4 Circuit Diagram to Symbolic

Process 2.2: Circuit Diagram to Symbolic

The transformation of a circuit diagram into a symbolic expression is very similar to evaluating the output of the circuit, except symbols instead of bits flow through the circuit. To lend context to the discussion of the conversion process, let's describe this process by converting the circuit diagram in Figure 2.5into a symbolic form.

Step 1: Identify the inputs and outputs of the circuit diagram. See Step 1 of Process 1.

Step 2: Build the shell of the truth table to hold the solution. See Step of Process 1.

Step 3: Determine the values in the output column. The symbolic output of a gate is written out only when symbolic inputs are present on all of the gate's input. The output of a gate is determined from the symbolic forms given on pages 12 and 14. For example, in order to determine the output of the AND gates in Figure 2.5A, the symbolic outputs of the NOT gates must first be determined. These are A', B', C' as shown in Figure 2.5B. Since the inputs to the top AND gate are A' and B, the output of the AND gate is labeled $A'B$. The output was not labeled "A'*B" because the "*" is often dropped from the symbolic expressions involving AND in order to save space. This is similar to the convention of dropping the "*" symbol when it is used as the multiplication operator in regular algebra. When two Boolean variables appear next to one another it is assumed that they are ANDed together. After labeling the output of the other AND gate, the output of the OR gate is labeled $A'B + AB'C'$ as shown in Figure 2.5B.

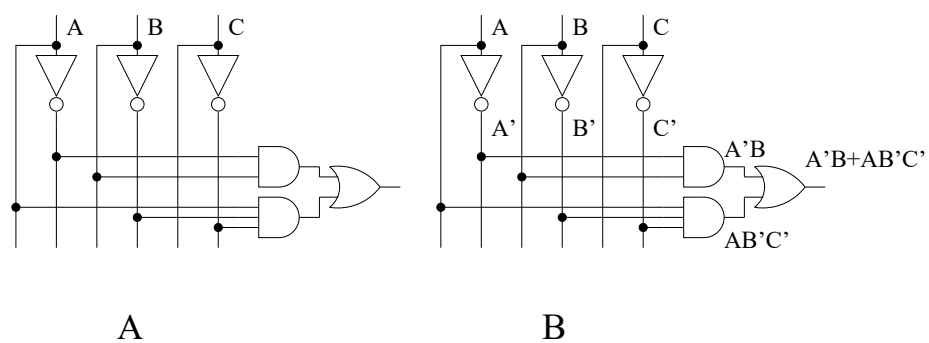


Figure 2.5: A) A circuit diagram. B) The input variables propagated through to the output.

It was earlier suggested that in order to determine the truth table for a circuit diagram, it is easier to first determine the symbolic form for the circuit diagram, and then determine the truth table from this symbolic form. Determining the symbolic form of a circuit's output is eight times easier than determining all eight rows of the truth table. The next section will show how to transform a symbolic form into a truth table.

2.5 Symbolic to Truth Table

Before we get started with this transformation you will need to understand and apply the concept of evaluation.

Evaluation is the process of substituting each variable's value into the equation and applying the operators to the values. For example, evaluate $F(A, B) = (A + B)'$ for $(A, B) = (0, 1)$.

Substituting in the values for the variables yields $F(0,1) = (0+1)'$. Applying the OR operation to 0 and 1 gives the result 1. Applying the NOT operator to the result of the OR operator yields 0. Thus, $F(A,B)$ evaluated at $(A,B) = (0,1)$ equals 0. In other words, $F(0,1) = 0$. The evaluation of more complex expressions requires you to apply the rules of operator precedence in Boolean Algebra.

Operator precedence specifies the order in which the operations of an expression are evaluated. Operations with higher precedence are performed before operations with lower precedence. The rules of operator precedence are listed for both algebras in the following table.

	Regular Algebra	Boolean Algebra
High precedence	Parenthesis	Parenthesis
	Exponents	Not
	Multiplication/Division	And
Low precedence	Addition/Subtraction	Or

For example, if you given the Boolean function F(A,B,C,D) = (A + B(C' + D))' and asked to evaluate $F(0,1,0,0)$ you would first substitute the values of the variables into the expression. To do this look at the left to right order of the Boolean variables in the $F(A,B,C,D)$ expression and match them to the left to right order of the bit values in $F(0,1,0,0)$. Thus everywhere you see a A you will substitute the value 0. B will get replaced with 1, C with 0 and D with 0. This will produce the expression:

$$(0 + 1(0' + 0))'$$

You need to apply operator precedence to determine which operation to perform first. Since parenthesis are the highest precedence operation, we to "look" inside the outer most parenthesis and evaluate:

$$0 + 1(0' + 0)$$

Since parenthesis are the highest precedence operator, we must evaluate what is inside the parenthesis first:

$$0' + 0$$

Since NOT has a higher precedence than OR, we perform the NOT before performing OR.

$$0' + 0 = 1$$

We then go back and put this value into expression where the original parenthesis expression. This gives us

$$0 + 1 * 1$$

Since AND has a higher precedence than OR, we get

$$0 + 1 * 1 = 1$$

Substituting this value into expression where the original parenthesis expression gives us:

$$1' = 0$$

Thus, $F(0,1,0,0) = 0$

Process 2.3: Symbolic to Truth Table

To lend context to the discussion of the conversion process, let's describe this process by converting the symbolic expression $F(A, B, C) = A'B + AB'C'$ into a truth table.

Step 1: Identify the inputs and outputs of the circuit diagram. Like the transformation of a circuit diagram to a truth table, the first step in this transformation is to identify the inputs and outputs of the circuit. Normally, the variable on the left-hand side of the equal sign in an equation is the output and any variables which appear on the right-hand side are inputs. In the symbolic expression $F(A, B, C) = A'B + AB'C'$, F is the output and A, B, C are the inputs.

Step 2: Build the shell of the truth table to hold the solution. Write down the shell of the truth table using the ordering given on page 16. The basic structure of the truth table is the same no matter how many variables the symbolic expression has. For example, if a symbolic expression has four input variables A, B, C, D, then start by listing the variables across the top of the truth table. Then write the bit values for the D variable by alternating 0 and 1 on every row, from top to bottom. The C variable would alternate every other row, from top to bottom. The B variable every fourth row, and the A variable every eighth row. The resulting truth table has 16 rows. This should not come as a surprise since, from page 3, there are $2^4 = 16$ different combinations of four input bits.

Step 3: Evaluate the symbolic expression for each combination of inputs.

We will evaluate $F(A, B, C) = A'B + AB'C'$ for every combination of (A, B, C) Let's start with the first row of the truth table and determine $F(0, 0, 0)$ by substituting $0, 0, 0$ for every occurrence of A, B, C in the symbolic expression. This yields $0' * 0 + 0 * 0' * 0'$. In order to evaluate this expression, perform the highest precedence operator first. In this case, evaluate the three NOTs yielding, $1 * 0 + 0 * 1 * 1$. The next highest precedence operator is AND, evaluating the two ANDs in the expression yields, $0 + 0$. Evaluating the remaining OR yields 0. Hence, $F(0, 0, 0) = 0$. Now continue this process for the remaining 7 row to produce the finished truth table.

A	B	C	$F(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

This process of evaluating the function for large numbers of inputs is tedious and error prone. In order to make good on the promise to produce an efficient transformation of a circuit diagram to a truth table, a better method is needed to complete the truth table from the symbolic form.

Two conditions make evaluating a symbolic expression difficult: its size and the evaluation process itself. Each difficulty will be addressed in turn. Instead of treating the symbolic expression as a single monolithic entity, break the expression into smaller subexpressions, evaluate each subexpression for all the inputs, and then put the values of the subexpressions back together. Three criteria govern the decomposition of an expression into subexpressions.

First, break the the expression into as few pieces as possible. Second, break the expression into subexpressions which are easy to put back together. Third, break the expression into subexpressions which are easy to evaluate.

For example, look at the expression $A'B + AB'C'$. This expression splits nicely into two subexpressions $A'B$ and $AB'C'$. This division meets the three criteria, only two subexpressions are defined, the subexpressions can be put back together by ORing them, and, as shown next, there is a simple way to evaluate the two subexpression.

In order to efficiently evaluate an expression for all of the rows of a truth table the number of questions asked about the expression must be reduced. This can be done for product terms. A *product term* is a group of variables (or variables negated) which are ANDed together. For example, $A'B$ and $AB'C'$ are both product terms. Instead of evaluating a product term for each row of a truth table, ask, "what input(s) cause the product term to evaluate to 1?" Since AND evaluates to 1 when all its inputs are 1, identify the rows of the truth table where the inputs equal 1. For example the product term $AB'C'$ evaluates to 1 only when $A = 1$, $B = 0$, and $C = 0$. Note, the B and C variables are negated in the product term so the variable must equal 0, so that its negation equals 1 before being ANDed. On the row $(A, B, C) = (1, 0, 0)$ the product term $AB'C'$ equals 1. What does the product term equal for all the other rows of the truth table? Since the product term does not equal 1 for these rows, the only alternative is for it to equal 0. Consequently, the remaining seven rows of the truth table equal 0 as shown in Table 2.1.

A	B	C	$A'B$	$AB'C'$	$F(A,B,C)$
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	1	0	1
1	0	0	0	1	1
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	0	0

Table 2.1: The truth table for $F(A, B, C) = A'B + AB'C'$ and its two subexpressions.

What inputs cause the product term $A'B$ to equal 1? Clearly, $A = 0$ and $B = 1$. While filling out the truth table for this product term there may be some confusion how to handle the C variable. Since the C variable is not present in the product term then its value is irrelevant. Consequently, the product term $A'B$ evaluates to 1 for $(A, B, C) = (0, 1, 0)$ and $(A, B, C) = (0, 1, 1)$, and evaluates to 0 for all other inputs as shown in Table 2.1.

Finally, combine these two subexpressions and compute the value of the function $F(A, B, C) = A'B + AB'C'$ for all rows of the truth table. In order to do this, look at each row of the truth table and ORing the value of the two subexpressions together. Since OR evaluates to 1 when any of its inputs equals 1, then it suffices to look for rows where either of the two subexpressions equal 1 and put a 1 for the output for F. All other rows will equal 0. The result is shown in Table 2.1.

The words *realize* and *implement* are used somewhat interchangeably to mean executing the design process in order to build a circuit – to make the digital system real. Since this text focuses on the logical behavior of circuits, not their physical behavior, the design process ends with a circuit diagram, see Figure 2.2. In general, when asked to realize or implement a circuit make it as real as possible. Along similar lines, the realization or implementation of a circuit is its physical manifestation. In order to answer questions about a digital systems realization or

implementation, reason about the behavior of its physical implementation. In the next section the last step in the realization of of digital systems, symbolic to circuit diagram, is examined.

Process 2.4: Symbolic to Circuit Diagram

The transformation of a symbolic expression into a circuit diagram is a recursive journey from the output to the input parsing the expressions along the way. To (loosely) paraphrasing Meriam Webster, parsing is the process of dividing an expression into subexpressions and identifying the relationship between the expression and subexpresions. An *expression* is a collection of boolean variables connected properly to one another by elementary logical functions.

To lend context to the discussion of the conversion process, let's parse $F(A, B, C) = A * (B + C') + B'C$ into a circuit diagram.

Step 1: Indentify the lowest precedence operator in the expression. When a symbolic expression is evaluated, some operation is always performed last, yielding the single-bit output of the function. This operation is the lowest precedence operator of the expression. In the expression for $F(A, B, C)$, the lowest precedence operation is the OR between the subexpressions $A * (B + C')$ and $B'C$.

Step 2: Draw the gate of the lowest precedence operator in the expression. This pretty simple, we draw the OR gate, labeled "1" in Figure 2.6.

Step 3: Connect the lowest precedence operator's output to the parent. Paraphrasing Mariam Webster, a parent is an entity from which other subordinates (children) arise.

In this parsing process a *parent expression* is broken down into zero or more *child expressions* in Step 2 by removing gate from the parent expression.

In this example we started **Step 1** by parsing $F(A, B, C)$ so this is the parent expression. So we will connect the output of the OR gate we removed from the circuit in **Step 2** to $F(A, B, C)$

Step 4: Remove the lowest precedence operation from the expression creating zero or more child subexpressions. Removing the OR operator from $A * (B + C') + B'C$ yields two subexpressions, $A * (B + C')$, and $B'C$.

Step 5: Parse each child subexpression independently. Each of the subexpressions created in **Step 4** need to be parsed independently. This means that we must parse $A*(B+C')$, and $B'C$. This means that we will use $A*(B+C')$ as the starting expression at **Step 1**. The output of the (child) expression $A * (B + C')$ is an input to the parent expression of $A*(B+C')$ which is OR gate formed when we parsed $A*(B+C')+B'C$. We now proceed to complete the parsing process below starting with the two subexpression $A * (B + C')$ and $B'C$.

Parse $A * (B + C')$		Parse $B'C$	
Step 1: The lowest precedence operation in this expression, AND between A and $(B + C')$.		**Step 1:** The lowest precedence operation in this expression, AND between B' and C.	
Step 2: Draw the AND gate labeled 2 in Figure 2.6.		**Step 2:** Draw the AND gate labeled 5 in Figure 2.6.	
Step 3: Connect the output of AND gate labeled 2 to the input of the OR gate labeled 1.		**Step 3:** Connect the output of AND gate labeled 5 to the input of the OR gate labeled 1.	
Step 4: Remove the AND from $A * (B + C')$ producing two child subexpressions A and $(B + C')$.		**Step 4:** Remove the AND from $B'C$ producing two child subexpressions B' and C.	
Parse A	Parse $(B + C')$	Parse B'	Parse C
leaf	**1:** OR $B + C'$	**1:** NOT B'	leaf
	2: OR gate 3	**2:** NOT gate 6	
	3: Connect 3 to 2	**3:** Connect 6 to 5	
	4: B and C'	**4:** B	
Parse B	Parse C'	Parse B	
leaf	**1:** NOT C'	leaf	
	2: NOT gate 4		
	3: Connect 4 to 3		
	4: C		
	Parse C		
	leaf		

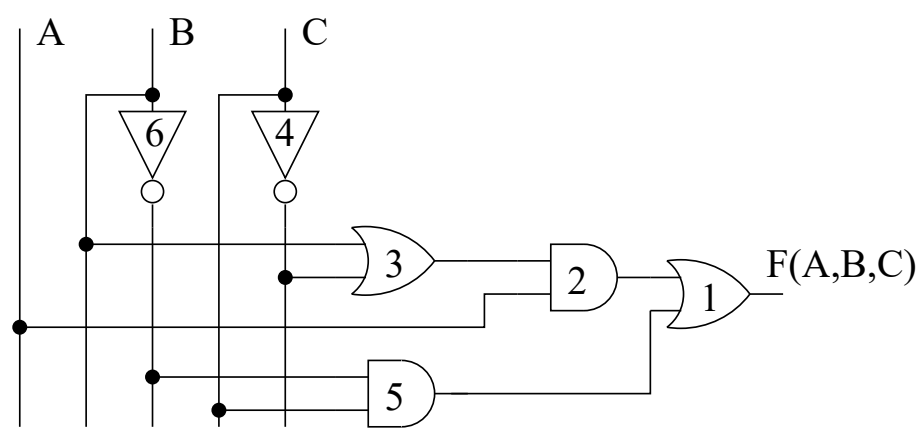


Figure 2.6: The circuit diagram for $F(A, B, C) = A(B + C') + B'C$.

Two simplification were made during the parsing process which deserve some attention leaves and *parenthesis*. When there the parsing process can go no further, there are no more operators in the expression, that expression is called a *leaf* . Examples of leafs include input variables and logic 0 and logic 1. You can think of these as leaves on a metaphorical parse tree.

Parenthesis are special operators which have no circuit representations. They allow us

a notation that enables low precedence operators to be evaluated before higher precedence operators. consequently, when an expression consists of a single set of parenthesis containing a expression, the parenthesis can be removed and parsing resumed with the expression.

During the parsing process, it is critically important to keep track of which parts of the expressions have been parsed and which have not. To do this, add lines under each of the subexpressions as they are created. After correctly parsing the expression, each variable should have a little line underneath it. This parsing-management can be a dizzying process jumping between different levels of the parsing process to resume parsing pieces of the expression left unrealized.

2.6 Symbolic to Symbolic

The laws of "regular" algebra are used to discover relationships between expressions, and to manipulate expressions into more convenient forms. Boolean Algebra has a set of laws which can be employed to manipulate symbolic expressions. The nine laws of Boolean Algebra are listed in Table 2.2.

Law	Primary	Dual
1.	x+0=x	x*1=x
2.	x+1=1	x*0=0
3.	x+x=x	x*x=x
4.	x"=x	
5.	x+x'=1	x*x'=0
6.	x+y=y+x	x*y=y*x
7.	x+(y+z)=(x+y)+z	x*(y*z)=(x*y)*z
8.	x*(y+z)=x*y+x*z	x+(y*z)=(x+y)*(x+z)
9.	(x+y)'=x'*y'	(x*y)'=x'+y'

Table 2.2: The laws of Boolean Algebra using the Boolean variables x, y, x.

Notice that all laws (except Number 4) have a dual. A dual of a symbolic expression is formed by swapping all ANDs and ORs and all 0s and 1s. Boolean Algebra expresses the *duality principle*: If a statement S is true, then the dual of S is also true. Later, it will be convenient to identify which law is used to manipulate a symbolic expression. In these cases, write down the law's number followed by D if it is a dual. For example, if the property $x*x = x$ is used to manipulate a symbolic expression, then indicate that Law 3D was used to manipulate the symbolic expression.

The laws in Table 2.2 can be verified as true by plugging in every possible combination of bits and checking if the two sides of the equation are equal. For example, to check Law 3D, set $x = 0$, yielding $0 * 0 = 0$ which is true, and then set $x = 1$ and verify that $1 * 1 = 1$.

The laws of Boolean Algebra can be used to prove two symbolic expressions are equal. This proof is performed by manipulating one of the expressions until it exactly equals the other side. For example, to prove that $A + A'B' = A + B'$, transform the more complex expression into the simpler expression. In the example, $A + A'B'$ is the more complex expression and $A + B'$ is the simpler expression. At each step of the transformation provide the law used to yield the next step.

$A + A'B' =$	Law 8D
$(A + A') * (A + B') =$	Law 5
$1 * (A + B') =$	Law 1D
$A + B'$	

Working on these derivations is a little like solving a maze or playing a good game of chess. Both require a player to "look ahead" and see what the implications of a decision will be. Not looking far enough ahead may result in arriving at one of the preceding steps. For example, consider what happened in the following loopy-derivation of $A(A + B) = A$. The correct derivation is shown on the right.

Loopy		Correct	
$A(A + B) =$	Law 8	$A(A + B) =$	Law 8
$AA + AB =$	Law 3D	$A * A + A * B =$	Law 3
$A + AB =$	Law 8D	$A + A * B$	Law 1D
$(A + A)(A + B) =$	Law 3	$A * 1 + A * B$	Law 8
$A(A + B)$	huh?	$A(1 + B)$	Law 2
		$A(1)$	Law 1D
		A	

Process 2.5: Expansion Trick

The *expansion trick* can be used to show that two expressions are equal to one another. To do this we will make the expression on the left-hand side (LHS) of the "=" sign equal to the expression on the right-hand side (RHS).

To lend context to the discussion of the expansion trick process, let's prove the following.

$$B' + A'(B'C + C) = (AB + BC')' \tag{2.1}$$

Step 1: Convert the LHS and RHS expressions into SOP form. The LHS has a single set of parenthesis, so we distribute the A' term to each term inside the parenthesis. We leave the B' term alone because it is already in a (degenerate) SOP form. Thus we are left with the SOP expression

$$B' + A'B'C + A'C$$

The RHS is more difficult because we will have to "distribute" the negation outside the parenthesis using **Law 9** using the following sequence of steps.

$(AB + BC')'$	Law 9
$(AB)'(BC')' =$	Law 9D
$(A' + B')(B' + C'') =$	Law 4
$(A' + B')(B' + C) =$	Law 8
$(A' + B')B' + (A' + B')C =$	Law 8
$A'B' + B'B' + A'C + B'C =$	Law 3
$A'B' + B' + A'C + B'C$	

Thus after this step our original proof now looks like:

$$B' + A'B'C + A'C = A'B' + B' + A'C + B'C \tag{2.2}$$

Step 2: Identify the missing variables for each product term. A couple of definitions are needed to understand the term "missing variable"

- **Working Set** The set of variables in an expression. The working set in our example is A, B, C.
- **Canonical Product Term** A product term that contains all the variables (or their negation) in the working set. For example, the product term $A'B'C$ on the LHS is a canonical product term.
- **Missing variable** The set of variable, that when ANDed to non-complete product term, makes the product term canonical. For example, the missing variable for the product term B' are A and C.

Putting all this together we get:

Term	Missing variable(s)
B'	A and C
$A'B'C$	None
$A'C$	B
$A'B'$	C
B'	A and C
$A'C$	B
$B'C$	A

Step 3: AND the missing variable and its complement to each product term. This step relies on Law 5 and La 1D from Boolean Algebra. Let's see how this work for the product term $A'C$ on the LHS.

Law 1D of Boolean Algebra states that $X * 1 = X$. Since the equality sign, "=" works both ways, we could also say that Law 1D of Boolean Algebra states that $X = X * 1$. In our case we will let X be the term $A'C$. Applying Law 1D, we have $A'C = A'C * 1$.

Law 5 of Boolean Algebra states that $X + X' = 1$. We could also interpret this as saying that $1 = X + X'$ In this case we will let X be the missing variable B. So we have $1 = B + B'$.

Combining these two ideas together, we get that $A'C = 1 * A'C = (B + B')A'C = A'BC + A'B'C$ The table below shows how each term is converted when its missing variable(s) are inserted.

Term	Missing variable(s)	Add missing variable	Expand expression
B'	A and C	$B'(A+A')(C+C')$	$AB'C + AB'C' + A'B'C + A'B'C'$
$A'B'C$	None		
$A'C$	B	$A'C(B+B')$	$A'BC + A'B'C$
$A'B'$	C	$A'B'(C+C')$	$A'B'C + A'B'C'$
B'	A and C	$B'(A+A')(C+C')$	$AB'C + AB'C' + A'B'C + A'B'C'$
$A'C$	B	$A'C(B+B')$	$A'BC + A'B'C$
$B'C$	A	$B'C(A+A')$	$AB'C + A'B'C$

Substituting these expanded expressions into Equation 2.2, we get a the rather long expression shown below.

$$AB'C + AB'C' + A'B'C + A'B'C' + A'B'C + A'BC + A'B'C =$$
$$A'B'C + A'B'C' + AB'C + AB'C' + A'B'C + A'B'C' + A'BC + A'B'C + AB'C + A'B'C \tag{2.3}$$

Step 4: Remove redundant product terms from each side. This step relies on Law 3 of Boolean Algebra which states that $X + X = X$. In our case, X will be a product term that is repeated on the LHS or RHS. For example the product term $A'B'C$ is repeated on the LHS as the 3rd and 5th terms. We could rearrange the terms on the LHS to look like $A'B'C + A'B'C$. Law 3 tells us that $A'B'C + A'B'C = A'B'C$. This may be unintuitive because in regular algebra $x + x = 2x$. But then again, in regular algebra "+" means addition not logical OR and regular algebra has the symbol 2 while Boolean Algebra has 0 and 1. Using Law 3 on both the LHS and RHS sides yields:

$$AB'C + AB'C' + A'B'C + A'B'C' + A'BC =$$
$$A'B'C + A'B'C' + AB'C + AB'C' + A'BC \tag{2.4}$$

Step 5: Rearrange the terms so that LHS and RHS have the same order. This step relies on Law 6 of Boolean algebra which states that $X + Y = Y + X$. In our context X and Y are the term in the LHS and RHS. It is a good idea to leave the terms on the LHS alone and move the terms on the RHS to get the same order. This let's you check that you have in fact the same terms on both side and have not made an error in your derivation.

$$AB'C + AB'C' + A'B'C + A'B'C' + A'BC =$$
$$AB'C + AB'C' + A'B'C + A'B'C' + A'BC \tag{2.5}$$

Step 6: Write out the proof. A formal proof requires us to manipulate one side of the equality into the other using the proper laws of Boolean algebra. The way that we will accomplish this is to transform the LHS of Equation 2.1 into the LHS of Equation 2.5 using the first 5 steps of this process. You then continue the derivation starting with the RHS of Equation 2.5 and proceed back up the process steps to the RHS of Equation 2.1.

$B' + A'(B'C + C) =$	Law 8
$B' + A'B'C + A'C =$	Law 1D
$B' * 1 * 1 + A'B'C + A'1C =$	Law 5
$(A + A')B'(C + C') + A'B'C + A'(B + B') =$	Law 8
$AB'C + AB'C' + A'B'C + A'B'C' + A'B'C + A'BC + A'B'C =$	Law 3
$AB'C + AB'C' + A'B'C + A'B'C' + A'BC =$	Law 6
$A'B'C + A'B'C' + AB'C + AB'C' + A'BC =$	Law 3
$A'B'C + A'B'C' + AB'C + AB'C' + A'B'C + A'B'C' + A'BC +$	
$A'B'C + AB'C + A'B'C =$	Law 8
$A'B'(C + C') + B'(A + A')(C + C') + A'C(B + B') + B'C(A + A') =$	Law 5
$A'B' * 1 + B' * 1 * 1 + A'C * 1 + B'C * 1 =$	Law 1
$A'B' + B' + A'C + B'C =$	Law 3D
$A'B' + B'B' + A'C + B'C =$	Law 3
$A'B' + B'B' + A'C + B'C =$	Law 8
$B'(A' + B') + C(A' + B') =$	Law 8
$(A' + B')(B' + C) =$	Law 9
$(A''B)'(B''C')' =$	Law 9D
$(AB)'(BC')' =$	Law 9
$(AB + BC')'$	*Q.E.D.*

The expansion trick runs into problems when there are a large number of terms in parenthesis. With a large number of terms in parenthesis, the distribution process becomes tedious and error prone. The following derivation utilizes Laws 4 and 9 to avoid this problem.

Show: $(A + B)(A' + B)(B + C) = B$

Derivation:

$(A + B)(A' + B)(B + C) =$	Law 4
$[(A + B)(A' + B)(B + C)]'' =$	Law 9D
$[(A + B)' + (A' + B)' + (B + C)']' =$	Law 9
$[A'B' + AB' + B'C']' =$	Law 8
$[B'(A' + A) + B'C']' =$	Law 5
$[[B'1 + B'C']' =$	Law 1D
$[B'(1 + C')]' =$	Law 2
$[B'1]' =$	Law 1
$[B']' =$	Law 4
B	

The first step in the derivation, double negating the expression, establishes the conditions for the application of Law 9D. One of the negations is pulled into the bracketed expression converting the product into the OR of each product term, negated. As shown, Law 9 can be applied to an expression with more than two terms. In the next step, Law 9 is applied to each term in parenthesis. What remains is a set of product terms inside the brackets. Over the next five steps this product term is simplified. In the final step, the second negation introduced back in the first step of the proof is combined with the B' inside the brackets, yielding the desired result.

2.7 Truth Table to Symbolic

The most technical transformation in the design process is transforming a truth table to symbolic expression. The idea behind this conversion is captured by the following analogy. Imagine eight members of the college cheer squad, each have a number between $0 \ldots 7$ pinned to their

uniform, are gathered together. Whenever members of the squad hears the number pinned to their uniform they give a cheer. If a member hears any other number, they do nothing, even if someone else is cheering like crazy.

In order to get a cheer whenever 1, 4, 6, or 7 was announced over the public address system, which members of the squad should be selected? Clearly, four members of the squad would be selected, the member with number 1, the member with number 4, the member with number 6, and the member with number 7.

The squad members in this analogy are circuit elements called *minterms*. A minterm for a function is a Boolean expression evaluating to logic 1 for a single combination of the inputs and evaluates to logic 0 for every other input. A cheering squad member corresponds to a minterm with an output of 1. The number pinned to each squad member corresponds to the binary input which causes the minterm to output 1. In order to build a symbolic expression for a truth table, first select the minterms corresponding to the inputs where the function equals 1. Next, OR together these minterms and label the output of the OR gate with the function. In the cheer squad analogy, the ability of sound from any member of the cheer squad to be heard corresponds to the OR gates.

Minterms Up till now minterms have been abstract entities, albeit with well-defined properties. To change that, consider all the minterms for a function with three input variables A, B, C. Since F has eight different inputs, it will have eight minterms. Next, build the minterm that "recognizes" the input $A = B = C = 1$. This task is accomplished by creating an expression which outputs 1 only when $A = B = C = 1$. In this case, the minterm is ABC because it equals 1 when all the inputs are 1, and for any other input, a 0. This minterm is denoted m_7, the lowercase m signifies that this is a minterm and the 7 denotes the decimal representation of the input which causes this minterm to equal 1.

The minterm for the input $A = 0, B = 1, C = 0$, m_2, is $A'BC'$. Negating the A and C variables allows a 0 input value for these variables to be inverted to 1 before being ANDed.

In order to make generating the remaining six minterms easier, formalize the observation just made and call it the *minterm trick*. A minterm for a particular input needs to include all the variables. If a variable's value is equal to 0, the variable should be negated in the AND expression, if its value is 1, it appears as itself in the AND expression. The table below shows all the minterms for three variables A,B,C.

A	B	C	minterm	symbol
0	0	0	A'B'C'	m_0
0	0	1	A'B'C	m_1
0	1	0	A'BC'	m_2
0	1	1	A'BC	m_3
1	0	0	AB'C'	m_4
1	0	1	AB'C	m_5
1	1	0	ABC'	m_6
1	1	1	ABC	m_7

A minterm is said to "recognizes its input" when that input causes the minterm to evaluate to 1. Thus, minterm m_3 recognizes the input $(A, B, C) = (0, 1, 1)$ and ignores all other inputs. In order to construct the symbolic expression for a truth table, OR together the minterms for which the function equals 1. Thus, when any one of the minterms recognizes its input, one of the inputs to the OR gate will equal 1 causing the output of the OR gate to become 1. Note, that at most one of the OR gates inputs will equal 1, since only one of the minterms will recognize its input. If none of the minterms recognizes the input, all the inputs of the OR gate will equal 0, causing the output of the OR gate to equal 0.

Process 2.6: Truth Table to Symbolic

To lend context to the discussion of the conversion process, let's describe this process by converting the truth table shown below into a symbolic expression.

A	B	C	F(A,B,C)
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	0
1	1	1	1

Step 1: Write out the minterms for each input. While you only need to write out the minterms for the product terms where the output equals 1, it does us no harm in writing out the minterms for all the rows of the truth table as shown in the truth table below.

A	B	C	F(A,B,C)	minterm
0	0	0	0	A'B'C'
0	0	1	1	A'B'C
0	1	0	1	A'BC'
0	1	1	1	A'BC
1	0	0	1	AB'C'
1	0	1	0	AB'C
1	1	0	0	ABC'
1	1	1	1	ABC

Step 2: OR together the minterms where the function equals 1. There are five inputs for which $F(A, B, C)$ equals 1, so the symbolic expression for F will include the five minterms from the rows where F equals 1. Hence,

$$F(A, B, C) = A'B'C + A'BC' + A'BC + AB'C' + ABC$$

Now, examine what happens to F for the input $(A, B, C) = (0, 0, 1)$. The minterm $m_1 = A'B'C$ evaluates to 1 while all the other minterms evaluate to 0. Since the OR function evaluates to 1 when any of the inputs are equal 1, F equals to 1. Hence, $F(0, 0, 1) = 1$ as expected. When $(A, B, C) = (1, 1, 0)$ is applied, all the minterms in the symbolic expression for F evaluate to 0 because none of them recognize this input. Thus, $F(1, 1, 0) = 0$ as expected.

When a function is constructed by ORing together minterms, it is said to be in *canonical sum of products* form or canonical SOP form. A symbolic expression is said to be in SOP form when it consists exclusively of a collection of product terms ORed together. The term sum/product is used to refer to the operators OR/AND respectively because these are what the operators look like in normal arithmetic. The term canonical is used because this symbolic form is the most elementary form possible to describe the function.

The canonical SOP representation for F defined above can be abbreviated as $F(A, B, C) = \sum m(1, 2, 3, 4, 7)$ where the $\sum$ symbol denotes the ORing together of minterms 1,2,3,4 and 7. In general, $\sum m(list)$ describes the inputs for which the function equals 1. When asked for a canonical SOP expression in the homework, write either the symbolic expression out, or use this abbreviated form.

Engineers are always striving to minimize the cost of their solutions. Towards this end, a second way to transform a truth table into a symbolic expression is introduced. It replaces the minterms with a different kind of building block, the maxterm.

Maxterms A *maxterm* for a function is a symbolic expression evaluating to 0 for a single combination of the inputs and evaluates to 1 for every other input. Before constructing symbolic expressions for a truth table, it is necessary to examine the structure of maxterms.

Since the function $F(A, B, C)$ has three inputs, it will have eight maxterms. Start by creating the maxterm for the input $(A, B, C) = (0, 0, 0)$. Many different expressions can be derived which equal 0 only for this input, but the simplest is $A + B + C$. This maxterm is abbreviated M_0, the uppercase M signifying a maxterm and the 0 denoting the decimal representation of the input which causes this maxterm to equal 0. The *maxterm trick* is used to simplify the process of creating maxterms.

A maxterm for a particular input needs to include all the variables. If a variable's value is equal to 1, the variable should be negated in the OR expression; if its value is 0, the expression appears as itself in the OR expression. The table below lists all the maxterms for the three variables A, B, C.

A	B	C	maxterm	symbol
0	0	0	A+B+C	M_0
0	0	1	A+B+C'	M_1
0	1	0	A+B'+C	M_2
0	1	1	A+B'+C'	M_3
1	0	0	A'+B+C	M_4
1	0	1	A'+B+C'	M_5
1	1	0	A'+B'+C	M_6
1	1	1	A'+B'+C'	M_7

In order to construct the symbolic expression for a truth table, AND together the maxterms for which the function equals 0. Thus, when any one of the maxterms recognizes its input, one of the inputs to the AND gate will equal 0, causing the output of the AND gate to go to 1. Note, that at most one of the AND gate's inputs will equal 0, since only one of the maxterms will recognize the input. If none of the maxterms recognizes the input, all the inputs of the AND gate equal 1, causing the output of the AND gate to equal 1. These ideas are applied to the function $F(A, B, C)$ below.

A	B	C	F(A,B,C)	maxterm	symbol
0	0	0	0	A+B+C	M_0
0	0	1	1	A+B+C'	M_1
0	1	0	1	A+B'+C	M_2
0	1	1	1	A+B'+C'	M_3
1	0	0	1	A'+B+C	M_4
1	0	1	0	A'+B+C'	M_5
1	1	0	0	A'+B'+C	M_6
1	1	1	1	A'+B'+C'	M_7

There are three inputs of $F(A, B, C)$ equal to 0, so the symbolic expression for F includes the three maxterms from the rows where F equals 0. Hence,

$$F(A, B, C) = (A + B + C)(A' + B + C')(A' + B' + C)$$

What happens when the input $(A, B, C) = (1, 0, 1)$ is applied? The maxterm $M_5 = (A' + B + C')$ evaluates to 0 while all the other maxterms evaluate to 1. Since the AND function

evaluates to 0 when any of the inputs are equal 0, F equals to 0. Hence, $F(1,0,1) = 0$ as expected. When $(A,B,C) = (1,1,1)$ is applied, all the maxterms in the symbolic expression for F evaluate to 1 because none of them recognize this input. Thus $F(1,1,1) = 1$ as expected.

This representation of F is called its *canonical product of sums* form or canonical POS form, because F is represented in its most elementary form as the AND (product) of a collection of OR (sum) terms. The canonical POS representation in the example can be abbreviated by the expression $F(A,B,C) = \prod M(0,5,6)$ where the $\prod$ symbol denotes the AND, the M symbol stands for maxterm, and the numbers in the list are the inputs for which $F(A,B,C)$ equals 0.

Notice, the functions in the two minterm and maxterm examples are the same. This was done to illustrate two points. First, the inputs present in the minterm list are those that do not appear in the maxterm list. In other words, $F(A,B,C) = \sum m(1,2,3,4,7) = \prod M(0,5,6)$. This should make sense. The minterm list denotes those inputs for which the function equals 1. If the function does not equal 1 for a particular input, this input is not in the minterm list, and then the function must equal 0 for this input, hence this input will appear in the maxterm list.

The second reason for using the same function is to illustrate how different realizations of the same function have different costs. A solution requiring fewer gates will cost less. The following table summarizes the number of gates required in the SOP and POS realization of $F(A,B,C)$.

	SOP	POS
number of NOTs	3	3
number of ANDs	5	1
number of ORs	1	3

Assuming that AND gates and OR gates have the same cost then the POS solution costs less. Thus a good engineer would recommend a canonical POS realization over a canonical SOP realization.

While, the transformation from a truth table to a symbolic expression may be the most technically challenging, it is certainly not the most difficult. This title belongs to the transformation of a word statement into a truth table. The difficulty of this transformation has its origin in the puffy cloud outline of a word statement shown in Figures 2.1 2.2. Words can be ambiguous and the ability of authors to convey precisely what the digital system needs to accomplish depends on their skill with the English language.

2.8 Word Statement to Truth Table

There is no algorithmic approach guaranteed to transform a word statement into a truth table. That said, the following list outlines essential information to obtain from the word statement in order to have any hope of transforming it into a truth table.

Process 2.7: Word Statement to Truth Table

To lend context to the discussion of the conversion process, let's describe this process by converting the following word statement into a truth table.

> Determine the truth table for a function with three bits of input and one bit of output. The output of the function should equal 1 when the 3-bit binary

input represents a binary number greater than 4.

Step 1: Identify the inputs and outputs and their meaning

The function has three bits of input and one bit of output. Since the inputs and outputs were not given names, use the default labels a_2, a_1, a_0 for the inputs and F for the output. The input variables are given subscripts because they are working together to form a 3-bit binary number. The binary number is referred to as A and its individual bits as a_2, a_1, and a_0. Hence, when $(a_2, a_1, a_0) = (1, 0, 1)$ then $A = 5$.

Step 2: Write out the skeleton of the truth table. Many 3-variable truth tables have been shown in this chapter. The notable point in this example is that the output is abbreviated as $F(A)$ to save space.

a_2	a_1	a_0	$F(a_2, a_1, a_0)$
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Step 3: Fill out the truth table. This example requires us to interpret all the inputs bits together as a single value. For example, we will look at the input $(a, b, c) = (1, 1, 0)$ not as three separate bits but as the binary number 110_2 which is equal to 6_{10}. With this in mind, let's try filling up the truth table.

Since the output should equals 0 for inputs less than or equal to $4 = 100_2$, we should list the output for inputs $(0, 0, 0)$ to $(1, 0, 0)$ with 0. Likewise we should fill in the output with 1 for inputs $(1, 0, 1)$ to $(1, 1, 1)$ as shown in the following table.

a_2	a_1	a_0	$F(a_2, a_1, a_0)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Before moving on, let's consider one final problem with two twists; the truth table will describe a function with four variables truth table has three bits of output. For the sake of brevity, the second and the third steps of the transformation process are combined.

Determine the truth table for a digital system having two, 2-bit inputs $A = a_1a_0$ and $B = b_1b_0$. Each 2-bit input represents a 2-bit binary number. The function has three bits of output called G, L, and E, which stand for greater, less, and equal, respectively. E equals 1 when $A = B$, $G = 1$ when $A > B$, and $L = 1$ when $A < B$.

Step 1 The function has four bits of input and three bits of output. The names of these variables are clear from the word statement.

Step 2 & 3 The truth table for this function has 16 rows because there are $2^4 = 16$ combinations of four bits (see page 3). In order to better visualize the solution to the problem, the values of A and B (derived from their values of a_1, a_0 and b_1, b_0) are included in the truth table.

a_1	a_0	b_1	b_0	A	B	G	L	E
0	0	0	0	0	0	0	0	1
0	0	0	1	0	1	0	1	0
0	0	1	0	0	2	0	1	0
0	0	1	1	0	3	0	1	0
0	1	0	0	1	0	1	0	0
0	1	0	1	1	1	0	0	1
0	1	1	0	1	2	0	1	0
0	1	1	1	1	3	0	1	0
1	0	0	0	2	0	1	0	0
1	0	0	1	2	1	1	0	0
1	0	1	0	2	2	0	0	1
1	0	1	1	2	3	0	1	0
1	1	0	0	3	0	1	0	0
1	1	0	1	3	1	1	0	0
1	1	1	0	3	2	1	0	0
1	1	1	1	3	3	0	0	1

In order to transform this truth table into a symbolic expression, realize each of the three outputs are independent of the others. For example, to determine the symbolic expression for G, cover-up the output columns corresponding to the L and E outputs and implement G. In order to realize L, just cover-up the columns associated with G and E and implement L as if the other two outputs did not exist. This yields the following three equations for the outputs.

$$G(a_1, a_0, b_1, b_0) = \sum m(4, 8, 9, 12, 13, 14)$$
$$L(a_1, a_0, b_1, b_0) = \sum m(1, 2, 3, 6, 7, 11)$$
$$E(a_1, a_0, b_1, b_0) = \sum m(0, 5, 10, 15)$$

Of course this leads to the question of how to build the circuit diagram for this 4-input, 3-output function. Students often want to OR together the G, L, and E outputs in order to have a single output, but this is would be incorrect because the word statement asked for three separate outputs. The correct way to build the circuit diagram is to build each of the circuits separately, give each its own output, and have them share the same inputs. A rough sketch of the circuit is shown in Figure 2.7.

Notice that the three output circuits share the same inputs, but otherwise are independent of one another. Sharing inputs allows each circuit to coordinate its output with the other circuits. When each circuit does what it is supposed to do for that input, the output looks like a unified whole even though it is generated from three separate circuits.

2.9 Timing Diagrams

After a circuit has been realized in hardware, the logical representation of 0s and 1s is replaced by physical voltages representing these logic levels. In order to observe the behavior of a physical digital system, an engineer typically uses a logic analyzer or oscilloscope. The waveforms

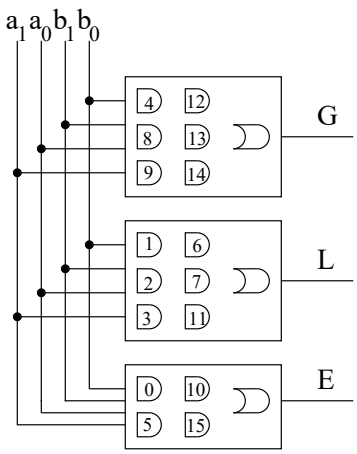


Figure 2.7: The circuit diagram for a circuit which compares the magnitude of the two inputs $A = a_1a_0$ and $B = b_1b_0$. Many of the internal details have been omitted.

drawn by the logic analyzer when applying inputs and examining the outputs is called a *timing diagram*. A timing diagram is a graph of the logic value of a signal versus time. Typically, several signals are combined on the same timing diagram to save space and in order to clearly show the relationship between the signals.

It is imperative to check if the digital system implemented behaves according to its specification. Hence, an understanding of timing diagrams is essential. With a small circuit it is reasonable to apply every possible combination of inputs and examine the output of the circuit.

To clairfy these concepts, let's look at a timing diagram for digital circuit with 3 bits of input A, B, C and 1 bit of output $F(A, B, C)$ shown in Figure 2.8.

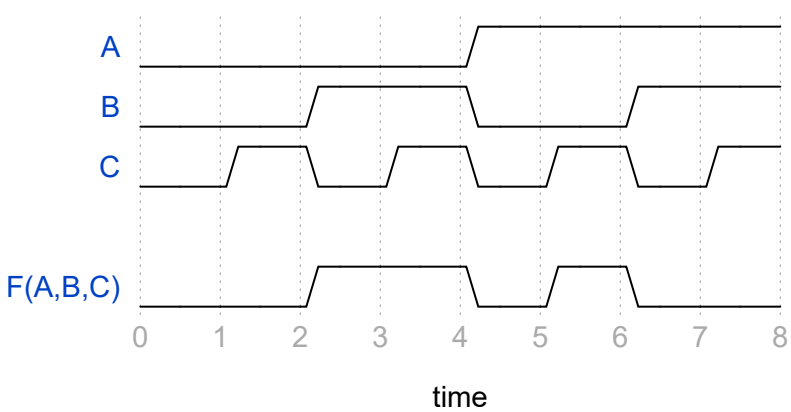


Figure 2.8: A timing diagram for a function with 3-bits of input and 1-bit of output.

A few observations about this timing diagram are in order:

- The vertical axis for each waveform represents the logic level of that waveform. There are two levels low (logic 0) or high (logic 1).

- It takes a finite amount of time for a waveform to change logic levels. This is shown by sloped transitions between logic levels. You do not need to show this slope in your hand-drawn timing diagrams.
- A positive edge occurs when a waveform changes from logic 0 to logic 1. You can remember this by associating "positive" with the slope of the change.
- A negative edge occurs when a waveform changes from logic 1 to logic 0. You can remember this by associating "negative" with the slope of the change.
- The positive and negative edges occur slightly after the time divisions. There is nothing special to read into this. This is just the way the waveDrom software draws waveforms.
- The input waveforms go through all combination of the inputs. While there are many ways the input waveforms could be sequenced to do this, it is convention to sequenced the inputs in truth table order.
- When "reading" information off a timing diagram focus on the regions where the signals a steady, not where they are changing. In our example, the signals are steady half way between the time divisions.
- The units on the time axis are unimportant for the time being. That said, time units in the nanosecond range would be reasonable.

At time=0.5 the input $(A, B, C) = (0, 0, 0)$ and the output in response to this input is 0. At time=2.5 the input is $(0, 1, 0)$ and the output is 1. Continue reading inputs and outputs off the timing diagram to verify that you get the information in the following truth table.

A	B	C	F(A,B,C)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

2.10 Exercises

1. **(2 pts. each)** Given: $F(A, B, C, D) = (AB' + (C + (AD)')(BD))'$

   a) Determine the truth table for $F(A, B, C, D)$

   b) Draw a schematic of the logic circuit which realizes F as shown, i.e. do not use Boolean Algebra on F.

2. **(2 pts. each)** For the circuit in Figure 2.9

   a) Write a Boolean expression for the function.

   b) Draw the truth table for the function.

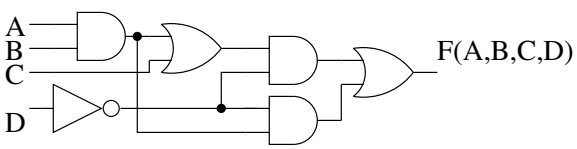


Figure 2.9: The circuit for Problems 2 and 3.

3. **(2 pts. each)** For the functions F,G,H,I defined by the truth table shown below:

   a) Determine the canonical SOP and POS realization for F, G, H, I.

   b) Draw the circuit diagram for the canonical SOP and POS realization.

   Treat each output independently of the other. For example when working with function I, cover up the columns F, G and H.

A	B	C	F	G	H	I
0	0	0	0	1	1	0
0	0	1	1	1	1	1
0	1	0	1	1	0	0
0	1	1	0	1	0	1
1	0	0	1	0	0	0
1	0	1	0	1	0	1
1	1	0	0	1	1	0
1	1	1	1	0	0	1

4. **(2 pts. each)** Prove the validity of the following statements using the laws of Boolean Algebra. For each step of the proof, identify which law was used.

   a) $X'Y' + XY + X'Y = X' + Y$

   b) $(X + Y')X'Y' = X'Y'$

   c) $(X + Y)(X' + Z) = XZ + X'Y$

   d) $X'Y' + (X + Y)Z = X'Y' + Z$

   e) $A'C + BC + AB = A'C + AB$

   f) $A(B + C) = AB + AB'C$

g) $(A+B+C)(A+B+C')(A'+B+C')(A'+B'+C') = (A+B)(A'+C')$

5. **(4 pts.)** Design a circuit called MUX2. MUX2 has three bits of input S, y_0, y_1 and one bit of output F. If $S = 0$, then $F = y_0$; else if $S = 1$, then $F = y_1$.

   a) Write down the truth table for the MUX2 function.

   b) Determine the canonical SOP realization for MUX2; do not simplify.

6. **(6 pts.)** Design a circuit called MUX4. MUX4 has six bits of input $S_1S_0, y_0, y_1, y_2, y_3$ and one bit of output F.
   If $S_1S_0 = 00$ then $F = y_0$
   else if $S_1S_0 = 01$ then $F = y_1$
   else if $S_1S_0 = 10$ then $F = y_2$
   else if $S_1S_0 = 11$ then $F = y_3$
   Without writing down the truth table determine a SOP expression to realize F by listing all possible inputs which will cause F to equal 1. Then try to simplify your expression using Boolean Algebra.

7. **(4 pts.)** Design a logic circuit called MAJ which has three inputs A, B, C and one output Z. The output equals 1 when a majority of the inputs are equal to 1, otherwise the output is 0.

   a) Write the truth table for the MAJ function.

   b) Determine the canonical SOP realization for the MAJ function, do not simplify.

8. **(4 pts.)** Let X and Y each be 2-bit signals whose elements are x_1x_0 and y_1y_0 respectively. Determine the $\sum m$ and $\prod M$ expression for a circuit whose 1-bit output z is defined by the following statement.

```
if (X == Y) then z = 1 else z = 0
```

9. **(4 pts.)** Let X and Y each be 2-bit signals whose elements are x_1x_0 and y_1y_0, respectively. Determine the $\sum m$ and $\prod M$ expressions for a circuit whose 1-bit output z is defined by the following statement.

```
if (X + Y > 3) then z = 0 else z = 1
```

10. **(3 pts.)** Determine the canonical SOP and POS expression for $F(A, B, C) = \prod M(0, 1, 4, 5)$ Hint, compose the truth table for F.

11. **(3 pts.)** Determine the canonical SOP and POS expression for $F(A, B, C, D) = \sum m(0, 4, 12, 15)$ Hint, write out the truth table for F.

12. **(4 pts.)** For the function $F(A, B, C) = BC + AB'C'$, draw a timing diagram for an input sequence that follows the same order as the rows of the truth table. Assume a propagation delay for NOT, AND and OR gate are all 10nS.

13. **(4 pts.)** Complete the timing diagram in Figure 2.10 for the functions $F(A, B, C) = AB' + BC + ABC'$ and $G(A, B, C) = (A + B')C + (BC')'$

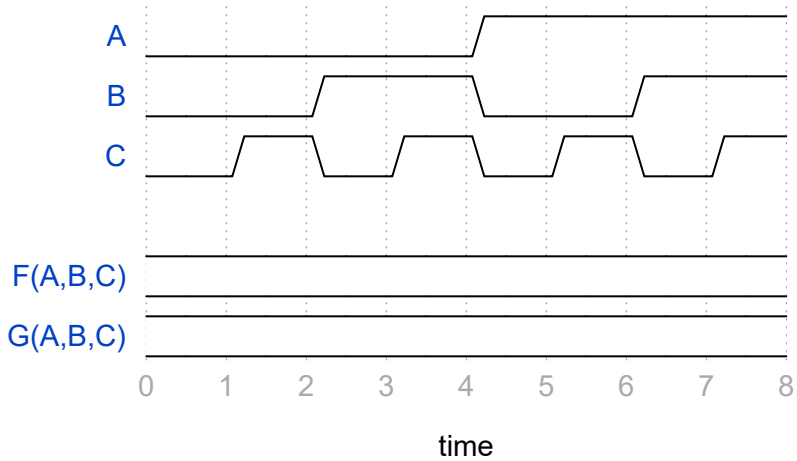


Figure 2.10: The timing diagram for two functions, $F(A, B, C)$ and $G(A, B, C)$.

14. **(16 pts.)** Design a circuit to control the water pump of a washing machine. The pump will not pump water if

    The lid is closed and the cycle is not fill

    The cycle is fill and the detergent level is empty

    The detergent is not empty and the lid is open

    The variables for this problem are:

    L = lid is closed

    C = cycle is fill

    D = detergent is empty

    P = pump will pump water

    Create a truth table which describes when the pump will not pump water. Call this output P'. Determine the canonical SOP expression for P'. Use this canonical SOP expression to generate a circuit diagram for P. This can be done by inserting an inverter onto the output of the circuit.

    Take the P' column from truth table and invert all the entries to generate a new output column called P (because the negation of P' is P). Determine the canonical SOP realization for P using this new column.

15. **Lab** Design a hexadecimal-to-seven-segment display (hex2SevenSegment) digital circuit. The hex2SevenSegment circuit converts a 4-bit input that represents a hexadecimal digit to a 7-bit output that displays the hexadecimal digit on a 7-segment display as shown in Figure 2.11. A 7-segment display is an figure-8 shaped arrangement of 7 LEDs that can be individually illuminated.

    Each of the 7 LEDs is associated with the output from the hex2SevenSegment circuit as shown in Figure 2.12A. Active high LEDs illuminate when their input is logic 1 and

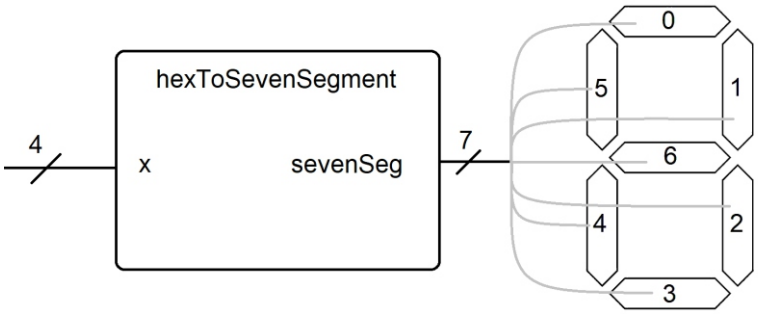


Figure 2.11: The connection beteen the 4-bit input nd the 7-segment display.

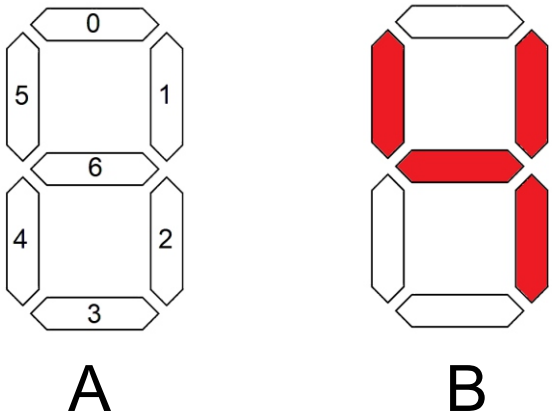


Figure 2.12: A) The individual segments of a 7-segment display and their bit position in the 7-bit output from the he2SevenSegment circuit. b) The illumination patter for the digit "4"

turn-off when their input of logic 0. If the LEDs of a 7-segment display are active then a binary input of 1100110 would display the pattern shown in Figure 2.12B

Since there are many different ways to illuminate the 7 segments to form the characters 0 . . . F, we will standardize our pattern to those shown in Figure 2.13.

The truth table for the hex2SevenSegment circuit is started below. Complete the truth table by filling the outputs for the seg column. Note that the number inside the square brackets is the bit index in the 7-bit output. The relationship between the bit index and the segments is shown in Figure 2.11.

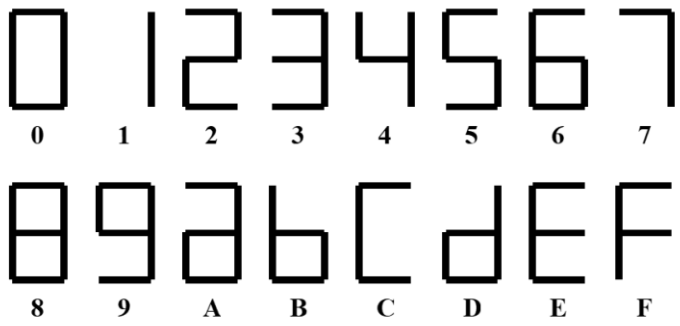


Figure 2.13: The illuminated patterns for all 16 inputs.

Chapter 3

Minimization of Logical Functions

The intent of this chapter is to explore how to realize circuits efficiently. Efficiency can be measured in many different ways; number of gates, speed, power dissipation, and layout size, being just a few. A generally accepted meaning of minimization is to minimize the number of gates required to realize a function in SOP or POS form. It should be clear that any logic function can be realized in an SOP or an POS form. For all but the simplest digital systems, the OR gate cannot be eliminated from the SOP realization. Hence, the focus should be on eliminating as many AND gates as possible. To do this, similar minterms are combined together using a trick from Boolean Algebra.

Consider the function $F(A, B, C, D)$ defined by the truth table below.

A	B	C	$F(A,B,C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

The canonical SOP expression for this function is $F(A, B, C) = A'BC' + A'BC$. This digital circuit requires two NOT gates, two AND gates and one OR gate. This realization, however, is not the most minimal one for F. Consider the following Boolean Algebra manipulations:

$$\begin{array}{ll} 1 & A'BC' + A'BC = \\ 2 & A'B(C' + C) = \\ 3 & A'B(1) = \\ 4 & A'B \end{array}$$

This realization of F requires one NOT gate, one AND gate and zero OR gates. Clearly, this realization requires fewer gates then the first realization. Hence, it is a more efficient solution.

The minterms used to realize $F(A, B, C)$ are $A'BC'$ and $A'BC$, corresponding to the inputs 010 and 011 respectively. The factoring in Line 2 takes advantage of the fact that each of the minterms has two variables in common, A' and B. This factoring could also be viewed as a result of the fact that the binary inputs of the two minterms have two bits in common $A = 0$ and $B = 1$ (see the truth table for F). The input variable changing between the two minterms, C, is factored out in Steps 2 and 3 in the above derivation. This observation forms the basis of the simplification trick:

> **Simplification Trick:** Two minterms whose inputs differ by a single bit may be replaced by a single product term that contains the variables which are the same in both minterms and excludes the variable which changes.

The simplification trick is used to obtain a simplified form for the function G defined by the truth table below.

A	B	C	$G(A, B, C)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

G has four minterms; the main task is to identify which pairs differ by a single bit. Inputs 001 and 101 differ by a single bit, A. Thus, with the simplification trick, the product term derived for this pair of minterms is: $B'C$. Likewise, the inputs 010 and 110 differ in the A bits, hence their product term is BC'. Since the product terms are ORed together in the SOP realization, the reduced product terms generated by the simplification trick are ORed together, hence $G(A, B, C) = B'C + BC'$. Trying to use Boolean Algebra on this expression will not produce a more minimal SOP form.

This simplification trick works fine for small examples. However, when the number of inputs to the function increases by 1, the number of rows in the truth table doubles. Identifying pairs of inputs which differ by a single bit would quickly become tedious and error-prone. A more efficient technique requires rearranging the truth table to make executing the simplification trick easier.

In order to accomplish this goal, the truth table is arranged so that rows of the truth table whose inputs differ by a single bit are adjacent to one another. With this accomplished, the simplification trick can be invoked by looking for adjacent 1s. Such a rearranged truth table is called a *Karnaugh-map* or Kmap for short. A Kmap for a function with three input variables A, B, C is shown in the leftmost Kmap.

Each of the cells in the Kmap corresponds to a row of the truth table and consequently has a unique combination of A, B, C variables. The A, B, C value of a cell is determined by reading off the A bit from the row index on the left and reading the B, C bits from the column index at the top. The Kmap to the right has the decimal representation of ABC placed in each cell. This numbering of the cells make the placement of the 1s of a function easier when specified in the $\sum m$ form.

Most importantly, notice that, given any cell, the cells to the left, right, up, and down all differ by a single bit. For example, the neighbors of the cell for the value 5 ("cell 5") are 1, 4, 7, each differing from 5 in the A, C, B variable, respectively. Cells 0 and 3 are not considered

Figure 3.1: A) The empty shell for a 3-variable kmap. Note it has 8 empty cells for the output of the function for the corresponding input. B) The 8 cells of the kmap filled in with the decimal equivalent of the 3-bit input corresponding to that cell. Note the bit order is A as the MSB and C as the LSB.

$A\backslash BC$	00	01	11	10
0				
1				

A

$A\backslash BC$	00	01	11	10
0	0	1	3	2
1	4	5	7	6

B

neighbors of cell 5 because they differ by two bits. Finally, notice that cell 4 and cell 6 differ by a single bit B, so they should be placed adjacent to one another, but are separated on the Kmap. To do this, imagine a 3-variable Kmap residing on the surface of a torus (doughnut) . The process of manipulating a 3-variable Kmap onto the surface of a torus is shown in Figure 3.2.

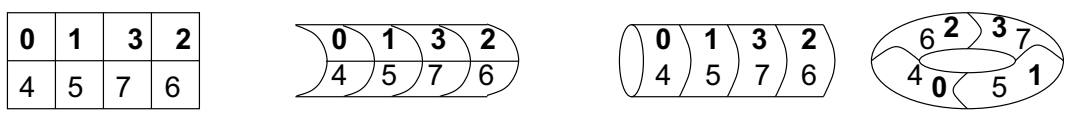


Figure 3.2: The folding and stretching required to transform a 3-variable Kmap onto the surface of a torus. The shaded numbers are on the back side of the surface.

When a Kmap is solved correctly the resulting SOP expression uses the minimum number of gates to realize the circuit in SOP form. Such a minimal SOP expression is referred to as a SOP $_{\text{min}}$ expression.

3.1 Karnaugh Maps

Using a Kmap to determine a SOP $_{\text{min}}$ realization of a function is a 4-step process.

Process 3.8: Solving a Kmap

This process is examined by determining the SOP $_{\text{min}}$ expression for the function $G(A, B, C) = \sum m(1, 2, 5, 6)$. Note, this expression is the same function minimized earlier.

Step 1: Draw an empty kmap

The first step is to draw an empty Kmap using the input variables of the function A, B, C. Since there are three input variables, this will be a three-variable kmap and look exactly like the Table below. Note that you should draw <u>all</u> 3-variable kmaps using this rectangular tabular format with the variables or the function in question in the upper left corner of the table.

$A\backslash BC$	00	01	11	10
0				
1				

Step 2: Place 1s in the Kmap for those inputs for which the function is to equal 1. Typically, the 0s of the function are omitted from the Kmap. It is understood that if you see a blank space in a Kmap, the function equals 0 for that input. The Kmap below shows the Kmap for G.

$A\backslash BC$	00	01	11	10
0		1		1
1		1		1

Step 3: Identify and circle pairs of adjacent 1s in the Kmap By adjacent, we mean cells whose inputs differ by a single bit. Since these neighbors lie up, down, let and right, we sometimes call these Manhattan neighbors, because in the epynomiously named city, streets are laid out in a clean grid and blocks consider neighboring when they share a street. Once you have found a pair of adjacent cells, circle them. Let's circle the pair of 1's in cell 1 and 5 as well as the pair of 1's in cell 2 and 6.

$A\backslash BC$	00	01	11	10
0		1		1
1		1		1

Step 4: Write the Boolean expression for the circled cells. The Boolean expression for a pair of adjacent 1 is formed by applying the simplification trick. Remember that this technique asks us to write down the input variables which do not change and discard the input variable which does change.

For the 1's in the red circle, $B = 0$ and $C = 1$ for both cells and the A variable changes. Hence, the Boolean expression for this grouping is $B'C$.

For the 1's in the blue circle, $B = 1$ and $C = 0$ for both cells and the A variable changes. Hence, the Boolean expression for this grouping is BC'.

Step5: OR together the circled Boolean expressions. Since we OR together the minterms when forming a canonical SOP expression, it makes sense that we should OR together the product terms found using the simplification trick. In our example, this step yields the SOP $_{\text{min}}$ expression $G(A, B, C) = B'C + BC'$.

To explore more fully how to solve Kmaps, consider the following truth table which lists seven functions $F \ldots M$ each having three inputs. The SOP $_{\text{min}}$ expression is derived for each of these functions, along the way illustrating many of the properties needed to solve Kmaps.

A	B	C	F	G	H	I	J	K	L	M
0	0	0	0	0	0	1	1	1	1	0
0	0	1	0	0	1	1	0	1	1	0
0	1	0	0	1	0	0	1	1	0	0
0	1	1	1	1	1	0	0	0	1	0
1	0	0	1	0	0	0	1	0	1	0
1	0	1	1	0	1	1	1	1	1	0
1	1	0	0	0	0	0	0	1	0	0
1	1	1	0	1	1	1	0	1	1	0

$A\backslash BC$	00	01	11	10
0			1	
1	1	1		

Kmap for F

$F = AB' + A'BC$ The grouping of cell 4 and cell 5 yields the product term AB'. Cell 5 and cell 3 cannot be combined because they differ in two bits. What is to do be done with the single cell 3? Since this cell

cannot be combined with another cell, it is just a minterm by itself. Sad perhaps, but perfectly legal. Hence, $F(A, B, C) = AB' + A'BC$.

$G = A'B + BC$ The natural question arising from this Kmap is, "Can cell 3 be reused in two different groups?" The answer is "Yes," and the reason can be demonstrated by performing some Boolean Algebra on the canonical SOP expression for G.

$A\backslash BC$	00	01	11	10
0			1	1
1			1	

Kmap for G

$$\begin{aligned} G(A,B,C) = \quad & A'BC' + A'BC + ABC = \\ & A'BC' + A'BC + A'BC + ABC = \\ & A'B(C' + C) + BC(A' + A) = \\ & A'B(1) + BC(1) = \\ & A'B + BC \end{aligned}$$

$A\backslash BC$	00	01	11	10
0		1	1	
1		1	1	

Kmap for H

In the second line, the minterm $A'BC$ was duplicated using Law 3 of Boolean Algebra. Note, this is the same minterm covered twice in the Kmap. Consequently, if a cell of the Kmap is covered by three different groupings then it would have to be replicated three times in the Boolean Algebra simplification.

$A\backslash BC$	00	01	11	10
0	1	1		
1		1	1	

Kmap for I

The grouping of cell 2 and cell 3 yields the product term $A'B$. The grouping of cell 3 and cell 7 yield the product term BC. Consequently, $G(A, B, C) = A'B + BC$. It is now possible to relate what happens in the symbolic expression when $(A, B, C) = (0, 1, 1)$ to the method used to solve the Kmap.

$A\backslash BC$	00	01	11	10
0	1			1
1	1	1		

Kmap for J

$H = C$ Grouping can be of size four. This can be explained by performing Boolean Algebra on the canonical SOP expression for H.

$$\begin{aligned} H(A,B,C) = \quad & A'B'C + A'BC + AB'C + ABC = \\ & (B + B')A'C + (B' + B)AC = \\ & A'C + AC = \\ & (A' + A)C = \\ & C \end{aligned}$$

$A\backslash BC$	00	01	11	10
0	1	1		1
1		1	1	1

Kmap for K

In general, grouping can be of size $2^i \times 2^j$ for integer i and j. It is a common mistake of students to make a grouping of size 3x2 – 3 is not a power of 2!

$A\backslash BC$	00	01	11	10
0	1	1	1	
1	1	1	1	

Kmap for L

$I = A'B' + AC$ One does not have to form every possible grouping. It is a common error for students to include the term $B'C$ in the expression for I. The expression $B'C'$ does not cause the function to output an incorrect value. Rather, this expression is not necessary for the circuit to function properly. Hence, including the expression $B'C'$ would make the circuit larger by one more AND gate than necessary.

$A\backslash BC$	00	01	11	10
0				
1				

Kmap for M

$J = A'C' + AB'$ Grouping can be made over the edge of the Kmap using the doughnut (torus) property illustrated in Figure 3.2. Notice, the minterms in the grouping on the upper row differ in the B bit.

$K = A'B' + AC + BC'$ **or** $K = A'C' + B'C + AB$ A Kmap may have more than one correct solution.

$L = B' + C$ Avoid the temptation to make a grouping of size 3×2. Instead, make two groupings of size 2×2 which overlap in two cells. Yes, overlapping big groupings is legal, too.

$M = 0$ This trivial function can be confusing when first encountered. Notice that regardless of the input, the output of the function is always 0. Hence, $M = 0$.

The process of "solving" a Kmap correctly leads to a minimal 2-level SOP expression (abbreviated SOP $_{\text{min}}$). SOP expressions are composed of two main levels of gates. A set of AND gates (level 1) leading into an OR gate (level 2). The NOT gates are ignored because they are both small and fast compared to AND and OR gates. The term "minimal" refers to the fact that the realization of the function requires the fewest possible gates among any 2-level SOP realizations.

In order to understand how to most efficiently solve a Kmap, some notation needs to be defined. An *implicant* is a legal grouping of 1s in a Kmap. An implicant which is not contained in any other implicant is called a *prime implicant* . An *essential prime implicant* is a prime implicant which covers a minterm not covered by any other prime implicant. For example, ABC is an implicant in the function $L(A, B, C)$, but it is not a prime implicant because it is contained in the essential prime implicant C.

When solving a Kmap, look for essential prime implicants and remove them from consideration. Removing the 1s from consideration does not mean removing the 1s from the problem. If needed, the 1s in an essential prime implicant can be used to form other groupings. After the essential prime implicants have been removed, look for *secondary essential prime implicants*, the essential largest groupings covering the remaining 1s. This identification of essential prime implicants goes on until either all the 1s are covered or a situation like the K function results. The K function has no essential prime implicants. At this point, resort to intuition (or brute force search) to minimize the number of groupings to cover the remaining 1s in the Kmap. Kmaps can be used to minimize functions in a variety of ways. One such use is discussed below.

Determine the SOP $_{\text{min}}$ expression for $F(A, B, C) = B'C' + BC' + ABC$. The term SOP $_{\text{min}}$ implies that a Kmap is required to solve the problem. In this case, figure out a way determine which region of the Kmap is described by each product term. For example, the product term $B'C'$ is equal to 1 when $(B, C) = (0, 0)$. Thus, 1s are placed in cell 0 and cell 4 of the Kmap. The product term BC' results in 1s being placed in cell 2 and cell 6. The product term ABC requires a 1 to be placed in cell 7. The resulting Kmap is shown in the margin. This Kmap can now be solved to determine the SOP $_{\text{min}}$ expression; $F(A, B, C) = C' + AB$. Incidentally, the grouping of cells 0, 2, 4, 6 in the Kmap above is affectionately referred to as a *Texas doughnut* because it has a big 2x2 grouping straddling the ends of the Kmap.

$A \backslash BC$	00	01	11	10
0	1			1
1	1		1	1

3.2 4-Variable Kmaps

The Kmap method can be adapted to work with functions of more than three variables. These larger Kmaps must be constructed so that adjacent cells differ by a single bit in order for the simplification trick to work. For example, a 4-variable Kmap is shown below with the decimal equivalent of the binary inputs shown in each cell.

$AB\backslash CD$	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

It is easy to verify that adjacent cells differ by a single bit. In addition, notice that adjacencies run across the top/bottom and left/right margins of the Kmap. In order to better understand this new structure, determine the SOP $_{\text{min}}$ expressions for the following functions.

- $F(A, B, C, D) = \sum m(0, 1, 4, 5, 8, 9)$
- $G(A, B, C, D) = \sum m(0, 5, 7, 10, 11, 14, 15)$
- $H(A, B, C, D) = \sum m(0, 2, 3, 5, 6, 7, 8, 10, 11, 14, 15)$

$F = A'C' + B'C'$ Resist the temptation to make a grouping of size 3×2; 3 is not a power of 2. Notice, one of the 2×2 groupings, $B'C'$ spans the edge of the Kmap (Texas doughnut style).

$AB\backslash CD$	00	01	11	10
00	1	1		
01	1	1		
11				
10	1	1		

Kmap for F

$G = A'B'C'D' + A'BD + AC$ This example demonstrates an interesting point: The size of the grouping determines the number of variables in the SOP $_{\text{min}}$ expression. For example, the grouping for covering one cell, $A'B'C'D'$, has four variables, the grouping covering two cells, $A'BD$, has three variables and the grouping covering four cells, AC, has two variables. The larger the grouping, the fewer variables that are required to describe the grouping, and consequently requires a smaller AND gate.

$AB\backslash CD$	00	01	11	10
00	1			
01		1	1	
11			1	1
10			1	1

Kmap for G

$H = B'D' + A'BD + C$ This solution is notable because it contains the *HyperDoughnut* grouping – the cells 0,2,8,10 forming the product $B'D'$. The only other notable feature in this Kmap is the large grouping of size 8, C.

$AB\backslash CD$	00	01	11	10
00	1		1	1
01		1	1	1
11			1	1
10	1		1	1

Kmap for H

3.3 5-Variable Kmaps

A 5-variable Kmap is a rearrangement of the rows of a 5-variable truth table such that adjacent cells of the Kmap differ by a single input bit. This rearrangement is accomplished by floating two, 4-variable Kmaps one above the other. A cell in a 5-variable Kmap can be combined with the cell to its left, right, below, above, up or down! That is, the three perpendicular directions along which adjacencies lie must be checked.

Determine the SOP $_{\text{min}}$ expression for $F(A, B, C, D, E) = \sum m(0, 1, 2, 5, 7, 8, 10, 15, 16, 18, 23, 24, 26, 28, 29, 31)$. Each of the 4-variable Kmaps is labeled with one of the two possible values of A, 0, or 1. The decimal values of the inputs can be formed by converting the binary input $ABCDE$ for each cell into decimal. The 4-variable Kmap with $A = 0$ is labeled just like an ordinary 4-variable Kmap. The 4-variable Kmap with $A = 1$ is labeled in the same order as a regular 4-variable Kmap, except the numbering starts at 16 because the MSB is 1.

$BC\backslash DE$	00	01	11	10
00	1	1		1
01		1	1	
11			1	
10	1			1

$A = 0$

$BC\backslash DE$	00	01	11	10
00	1			1
01			1	
11	1	1	1	
10	1			1

$A = 1$

This Kmap is notable because it contains the granddaddy of all the border jumping groupings, the *ultra-doughnut*. This grouping occupies the eight corners of the 5-variable Kmap. A grouping of size four jumps between the two Kmaps. Other than this aspect, the solution is fairly straightforward. $F(A, B, C, D, E) = C'E' + A'B'D'E + CDE + ABCD'$

To conclude the Kmap concept, a few trends should be noted. First, there is a relationship between the number of variables in a Kmap and the number of neighbors.

Variables	Neighbors
3	3
4	4
5	5

The relationship is linear, that is a cell in a Kmap with N variables should have N neighbors. By considering the simplification trick, this relationship should make sense. Two cells are adjacent if their binary representations differ by a single bit. An N-bit number has N positions where a single bit could be changed, thus it has N neighbors in the Kmap.

Also there is a relationship between the number of 1s in a grouping and the number of variables appearing in the grouping's product term. For example, in the $G(A, B, C, D)$ function above, the grouping of a single minterm was described by the product term $A'B'C'D'$ containing four variables. The largest grouping of four minterms was described by the product term AC containing two variables. The following table describes this relationship for a 5-variable function.

Number of 1's	Variables
32	0
16	1
8	2
4	3
2	4
1	5

This table demonstrates why making the groupings as large as possible is desirable, large groupings have smaller AND gates.

3.4 Multiple Output Circuits

In the previous chapter, digital systems with more than one output were considered. Multiple output functions are realized by realizing each output independently of the others. For example, consider a digital system with three inputs A, B, C and two outputs $F(A, B, C)$ and $G(A, B, C)$ shown in the truth table below:

A	B	C	F	G
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	1	1
1	0	0	1	0
1	0	1	1	0
1	1	0	0	0
1	1	1	0	1

If the functions are solved independently of one another, $F(A,B,C) = AB' + A'BC$ and $G(A,B,C) = BC + A'BC'$. The only connection the two circuits have to each other is their inputs, both share the same (A,B,C). Thus, when $(A,B,C) = (1,0,1)$ $F = 1$ and $G = 0$ just as expected according to the truth table. The circuit diagram of the F and G functions is shown in Figure 3.3.

$A\backslash BC$	00	01	11	10
0			1	
1	1	1		

$F(A,B,C) = AB' + A'BC$

$A\backslash BC$	00	01	11	10
0			1	1
1			1	

$G(A,B,C) = BC + A'BC'$

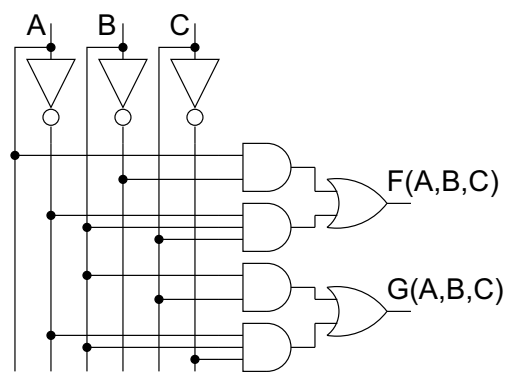


Figure 3.3: Two functions, F and G, which share inputs.

However, when you a design circuit that has multiple outputs which share the same inputs, there are efficiencies that you can extract if you share product terms. A shared product term is an AND gate that can be used in the realization of more than one function. For example, consider the two functions $H(A,B,C) = \sum m(1,4,5,6)$ and $I(A,B,C) = \sum m(1,2,3,5)$.

The Kmaps and the solutions for these two functions are shown in the margins. Notice that the grouping $B'C$ appears in both SOP $_{\text{min}}$ expressions. This AND gate need appear only once in the circuit diagram because its output can be directed to both OR gates which form the outputs for H and I. The circuit diagram for these two functions is shown in Figure 3.4.

$A\backslash BC$	00	01	11	10
0		1		
1	1	1		1

$H(A,B,C) = \sum m(1,4,5,6) = B'C +$
AC'

$A\backslash BC$	00	01	11	10
0		1	1	1
1		1		

$I(A,B,C) = \sum m(1,2,3,5) = B'C +$
$A'B$

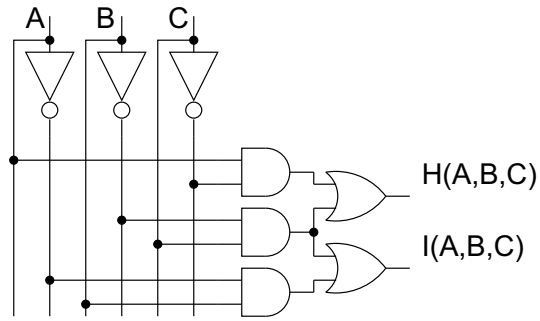


Figure 3.4: Two functions which share the product term $B'C$.

3.5 "Don't Cares"

There are situations when engineers "don't care" what the output of a digital system is for certain inputs. This situation often happens when a particular input will never occur. For example, consider the a digital system which classifies its inputs as either even or odd. It has four bits of input $a_3a_2a_1a_0$, and one bit of output, F. The 4-bit input represents a decimal number, $0 \leq A \leq 9$, the inputs $10 \leq A \leq 15$ should never be applied. The output equals 1 when A is even (0 is considered an even number) and outputs 0 when A is odd.

As a first attempt to solve this problem, ignore the inputs corresponding to $10 \leq A \leq 15$ and leave the corresponding outputs blank. The outputs are to be defined for inputs $0 \leq A \leq 9$. The resulting Kmap and its solution is shown in the margins.

$a_3a_2 \backslash a_1a_0$	00	01	11	10
00	1			1
01	1			1
11				
10	1			

$F(a_3, a_2, a_1, a_0) = a_3'a_0' + a_2'a_1'a_0'$

The solution shown in the margin will certainly work correctly, since the outputs are correctly defined for the legitimate inputs. However, the realization could have been made more efficient if the fact that the inputs $10 \leq A \leq 15$ will never be applied had been taken into consideration. Then, it does not matter what the circuit outputs when $10 \leq A \leq 15$. Notice, that in the first solution, implicitly the illegal inputs were treated as odd numbers; the circuit would output a 0 for $10 \leq A \leq 15$. Since these are illegal inputs, they are assigned any convenient value with an eye on making the final realization as efficient as possible. To denote this freedom in assigning the output of the function either value, an "X" is placed in any cell of the Kmap where the output doesn't matter. These Xs are referred to as "don't cares".

The utility of "don't cares" lies in the fact that they can be used to make groupings larger and consequently the realization of the function more efficient. If an X can be used to make a grouping larger, then it should be included in that grouping. Including "don't cares" in a group will cause the circuit to output 1 for the input corresponding to the "don't care". If, on the other hand, an X cannot be used in any grouping, then leave it uncovered. The circuit will output a 0 for the uncovered "don't care".

To better understand how "don't cares" are used, they are included into

the even/odd function for inputs $10 \leq A \leq 15$ and the resulting Kmap shown in the margin is solved.

$a_3a_2 \backslash a_1a_0$	00	01	11	10
00	1			1
01	1			1
11	X	X	X	X
10	1		X	X

$F(a_3, a_2, 2_1, a_0) = a_0'$

By using the "don't cares" in cells 10, 12, and 14, a grouping of size eight is formed. This grouping reduces the cost of the solution to a single inverter.

"Don't cares" are included in the abbreviated canonical SOP form by writing a "d" followed by a list of inputs for which the output is unimportant. For example, the even/odd function is described as $F(A, B, C, D) = \sum m(0, 2, 4, 6, 8) + \sum d(10, 11, 12, 13, 14, 15)$.

"Don't cares" can be used on the input variables of a truth table in order to compress the size of the truth table. Two rows can be combined when their outputs are the same and their inputs are adjacent (in the Kmap sense) to one another. A single "don't care" on an input of a row of a truth table generates two rows; one with the "don't care" set to 1 and another with the "don't care" set to 0. For example, the following two truth tables describe the same function.

A	B	C	F(A,B)
x	0	x	0
x	1	0	1
0	1	1	0
1	1	1	1

Compressed Truth Table

A	B	C	F(A,B)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Full Truth Table

The row $(x, 1, 0)$ generates two rows in the full truth table, $(0, 1, 0)$ and $(1, 1, 0)$. The row $(A, B, C) = (x, 0, x)$ has two "don't cares". Since the values of these "don't cares" can be selected independent of one another, this row generates four rows in the full truth table, $(0, 0, 0)$, $(0, 0, 1)$, $(1, 0, 0)$, and $(1, 0, 1)$. In general, a row with N "don't cares" generates 2^N rows in the full truth table. The motivation for using "don't cares" will become more clear in later chapters when digital systems with six or more inputs are encountered.

3.6 Minimizing to POS

So far, the chapter has focused on finding efficient SOP $_{\text{min}}$ realizations of circuits. However, Section 2.7 provided a function whose POS realization was less costly than the SOP realization. It is not difficult to imagine functions for which the POS $_{\text{min}}$ realization is less costly than the SOP $_{\text{min}}$ realization. In order to derive the POS $_{\text{min}}$ realization of a function F, the function is transformed – put into a Kmap, a SOP $_{\text{min}}$ realization found, and then the SOP $_{\text{min}}$ realization transformed into a POS $_{\text{min}}$ realization for F. In order for this process to work, the two transformations of the function will have to "undo" one another so they have no net effect on the function.

In order to cover all possibilities, the transform from any of the starting forms into one of the ending forms is investigated.

$\sum m$		
$\prod M$	$\longrightarrow$	SOP $_{\text{min}}$
SOP		POS $_{\text{min}}$
POS		
Starting Form		Ending Form

Eight potential transforms need to be explored. For each transformation, a plan is developed; a series of steps. The following five helpful facts become the steps in all these plans.

Helpful Facts

1 Negating a Kmap for F negates F. If all the bits in a Kmap for a function F are flipped, then it is the same as negating the output of F.

2 Negating a SOP expression for F yields a POS expression for F'. This statement is based on Law 9 and Law 9D of Boolean Algebra. Law 9 states that $(x+y)' = x'y'$. Law 9D states that $(xy)' = x' + y'$. The transformation from SOP to POS is illustrated in the following example.

$F = A'B'C' + A'BC + AB' + C$	negate F
$F' = (A'B'C' + A'BC + AB' + C)'$	Law 9
$F' = (A'B'C')'(A'BC)'(AB')'(C)'$	Law 9D
$F' = (A+B+C)(A+B'+C')(A'+B)(C')$	

The shortcut typically used to determine the POS expression is to replace all ANDs and ORs and negate each variable.

3 Negating a POS expression for F yields a SOP expression for F'. The proof of this statement is derived by examining the previous algebraic example from bottom to top.

4 $\sum m(list) = \prod M(list')$. $list'$ denotes the decimal numbers not in $list$. Since $list$ describes those inputs for which the function equals 1, $list'$ describes those inputs for which the function equals 0.

5 $F'' = F$. This expression is just Law 4 of Boolean Algebra.

The plans used to transform between forms consist of a sequence of steps. At each step, the function (or its complement) is represented in one of its various forms (Kmap, SOP, $\sum m$, etc..). Step 1 always describes the given form of the function. Likewise, the last step always describes the desired form of the function. The action between the steps is implicit from the beginning and ending forms.

To better understand how a plan is put together and how the transformations are performed, consider a plan for the transformation of a $\sum m$ expression into a SOP $_{\text{min}}$ realization.

Process 3.9: Determine SOP $_{\text{min}}$ given a $\sum m(\ldots)$ expression.

This should be the (by now) familiar process of drawing a kmap with the correct number of variable and putting 1's in the cells where the function equals 1 and then solving the kmap.

The Plan:

Step	1	2	3
Function	F	F	F
Form	$\sum m(\ldots)$	Kmap	SOP $_{\text{min}}$

The plan consists of three steps. The function in all three steps is F. In Step 1, the function F is written in the $\sum m(\ldots)$ notation. In the second step, the function is placed into a Kmap using the process outlined in this chapter. In the third step, the SOP $_{\text{min}}$ expression is derived from the Kmap.

Another familiar transform is handled next.

Process 3.10: Determine SOP $_{\text{min}}$ given a SOP expression.

The only difference in this transformation is the beginning form. The discussion on page 48 illustrates how to put a SOP Boolean expression into a Kmap. Once in a Kmap, the remaining steps should be clear.

The Plan:

Step	1	2	3
Function	F	F	F
Form	SOP	Kmap	SOP $_{\text{min}}$

The next transformation has a worked out solution to guide you through the steps of the process.

Process 3.11: Determine POS $_{\text{min}}$ given a $\sum m(\ldots)$ expression.

The Plan:

Step	1	2	3	4	5
Function	F	F	F'	F'	F"=F
Form	$\sum m(\ldots)$	Kmap	Kmap'	SOP $_{\text{min}}$	POS $_{\text{min}}$

An example shows how this plan is put into practice. Determine the POS $_{\text{min}}$ expression for $F(A,B,C) = \sum m(3,4,5)$.

Step 1 $F(A,B,C) = \sum m(3,4,5)$.

Step 2 The Kmap for F

$A\backslash BC$	00	01	11	10
0			1	
1	1	1		

Step 3 This step relies on helpful fact 1, inverting the entries in a kmap, negates the function. Thus, the Kmap for F'

$A\backslash BC$	00	01	11	10
0	1	1		1
1			1	1

Step 4 Requires that we solve the kmap for F' yielding:

$$F'(A,B,C) = A'B' + AB + BC'$$

Step 5 Relies on helpful fact 2, negating a SOP expression for F yields a POS expression for F'. We will work through the steps here as a review.

$$\begin{aligned} F''(A,B,C) = F(A,B,C) &= (A'B' + AB + BC')' \\ &= (A'B')'(AB)'(BC')' \\ &= (A'' + B'')(A' + B')(B' + C'') \\ &= (A + B)(A' + B')(B' + C) \end{aligned}$$

If you have doubts about this process, I invite you to plug in values for (A,B,C), evaluate the POS $_{\text{min}}$ expression for F and compare your answers to the kmap you produced in Step 2. This process will bolster your confidence that these very different looking forms produce the same logical output for all A,B,C combinations.

Process 3.12: Determine POS $_{\text{min}}$ given a SOP expression.

The Plan:

Step	1	2	3	4	5
Function	F	F	F'	F'	F"=F
Form	SOP	Kmap	Kmap'	SOP $_{\text{min}}$	POS $_{\text{min}}$

This transformation is the same as the $\sum m(\ldots)$ to POS $_{\text{min}}$, process 11 from Step 2 onward.

Process 3.13: Determine the SOP $_{\text{min}}$ given a $\prod M(\ldots)$ expression.

The Plan:

Step	1	2	3	4
Function	F	F	F	F
Form	$\prod M(\ldots)$	$\sum m(\ldots)$	Kmap	SOP $_{\text{min}}$

This transformation utilizes the fourth helpful fact. The swapping of the list of 0s for a list of 1s. This step confuses many first time students so lets work an example to show how this plan is put into practice. Determine the POS $_{\text{min}}$ expression for $F(A,B,C) = \prod M(0,1,2,6,7)$.

Step 1 $F(A,B,C) = \prod M(0,1,2,6,7)$.
Step 2 $F(A,B,C) = \sum m(3,4,5)$.
Step 3 Follow steps 2 onward in process 9.

Process 3.14: Determine the POS $_{\text{min}}$ given a $\prod M(\ldots)$ expression.

The Plan:

Step	1	2	3	4	5	6
Function	F	F	F	F'	F'	F"=F
Form	$\prod M(\ldots)$	$\sum m(\ldots)$	Kmap	Kmap'	SOP $_{\text{min}}$	POS $_{\text{min}}$

In Step 2, the list of maxterms, given in Step 1, is transformed into a list of minterms. The Kmap is negated in Step 4 so that the SOP $_{\text{min}}$ description of F' can be negated, yielding a POS $_{\text{min}}$ expression for F.

Process 3.15: Determine SOP $_{\text{min}}$ given a POS expression.

The Plan:

Step	1	2	3	4	5
Function	F	F'	F'	F"=F	F
Form	POS	SOP	Kmap	Kmap'	SOP $_{\text{min}}$

This is an interesting transformation because F must be negated before being placed into a Kmap. Since F' is in the Kmap, the Kmap must be negated so that the solution to the Kmap yields a SOP $_{\text{min}}$ expression for F.

Process 3.16: Determine the POS $_{\text{min}}$ given a POS expression.

The Plan:

Step	1	2	3	4	5
Function	F	F'	F'	F'	F"=F
Form	POS	SOP	Kmap	SOP $_{\text{min}}$	POS $_{\text{min}}$

An example shows how this plan is put into practice. Determine the POS $_{\text{min}}$ expression for $F(A,B,C) = (A' + B + C)(A + C')B'$.

Step 1 $F(A,B,C) = (A' + B + C)(A + C')B'$.

Step 2 Relies on helpful fact 3, negating a POS expression for F yields a SOP expression for F'.

$$\begin{aligned} F'(A,B,C) &= ((A' + B + C)(A + C')B')' \\ &= (A' + B + C)' + (A + C')' + B'' \\ &= A''B'C' + A'C'' + B \\ &= AB'C' + A'C + B \end{aligned}$$

Step 3 The Kmap for F'

$A\backslash BC$	00	01	11	10
0		1	1	1
1	1		1	1

Step 4 $F'(A,B,C) = AC' + A'C + B$

Step 5 Relies on helpful facts 5 and 2.

$$\begin{aligned} F''(A,B,C) = F(A,B,C) &= (AC' + A'C + B)' \\ &= (AC')'(A'C)'B' \\ &= (A' + C'')(A'' + C')B' \\ &= (A' + C)(A + C')B' \end{aligned}$$

3.7 Espresso

The Kmap minimization method can be applied to problems with up to eight variables. Beyond eight variable, the process becomes too tedious and error-prone to be practical. Functions with 20 variables are not uncommon, but would require a truth table containing over a million rows! An algorithmic approach is needed to handle such large instances. Espresso https://en.wikipedia.org/wiki/Espresso_heuristic_logic_minimizer is a 2-level logic minimization tool, that while not guaranteed to find a minimal solution, does a good job on most functions. Since Espresso was written before the days of visual interfaces, it is a small program, run from

a command line interface. (A command line interface is a text interface to the programs and operating services available to the user of a computer.) The operating system prompts the user by displaying a ">" character in a window. The user then types in the name of the program or service they want to access. Typing "espresso" at the command line prompt invokes the Espresso program.

Espresso is simple to use, just create the truth table for a function in a text editor, then run Espresso on the file. The input file for Espresso contains:

1. **Comments.** Comments are always preceded with the # symbol. If a comment is encountered, Espresso ignores the rest of the line.

2. **The number of inputs.** The number of inputs is described by the statement `.i` followed by a number describing the number of inputs to the function. The `.i` must be included in an Espresso file.

3. **The number of outputs.** The number of outputs is described by the statement `.o` followed by a number describing the number of outputs from the function. The `.o` must be included in an Espresso file.

4. **The labels for the inputs.** The labels for the inputs are described by the statement `.ilb` followed by a list of names of the inputs separated by spaces. The `.ilb` does not have to be included in an Espresso file. If not included, then Espresso assigns its own names to the inputs.

5. **The labels for the output(s).** The labels for the outputs are described by the statement `.ob` followed by a list of names of the outputs separated by spaces. The `.ob` does not have to be included in an Espresso file. If not included, then Espresso assigns its own names to the outputs.

6. **The truth table.** The truth table is organized with the input bits on the left and the output(s) bit(s) on the right. "Don't cares" in the inputs or outputs are denoted with a minus sign -. The order of the rows in the truth table is unimportant.

The following is an example Espresso input file for the function $F(a,b,c) = \sum m(1,3,6,7)$.

```
# File: simple.txt
# Name: <your name>
#       <course name>
# Date: <semester>
# Desc: F(a,b,c) = sum m(1,3,6,7)
.i 3
.o 1
.ilb a b c
.ob F

000 0
001 1
010 0
011 1
100 0
101 0
110 1
111 1
```

This file must be created in text format – do not create this document in a "word processor" such as MS Word or WordPerfect. NotePad, Edit, emacs, or vi are preferable choices for creating this file. Save the example Espresso file as `simple.txt`. The command line

Running Espresso on the example file produces the following output.

```
> espresso simple.txt
# File:    simple.txt
# Name: <your name>
#       <course name>
# Date: <semester>
# Desc:    F(a,b,c) = minterms (1,3,6,7)
.i 3
.o 1
.ilb a b c
.ob F
.p 2
11-     1
0-1     1
.e
```

Espresso echoes back comments, the number of inputs, outputs, and labels defined in the input file, and identifies the number of product terms used in the realization. In the example, the `.p 2` statement means that the solution found by Espresso required two product terms. The two lines after the `.p 2` statement describe the realization in a programmable logic array (PLA) format. In the PLA format, each row represents a product term. The left three columns of each row represent the state of the inputs variable in the product term. The variables are in the same order as they were defined in the truth table. The state of a variable can be "1", "0", or "-".

- 1 means, include the variable in the product.
- 0 means, include the negated variable in the product.
- - means, exclude the variable from the product.

Thus, the row "11-" represents the product term AB and the row "0-1" represents the product term $A'C$. The column of 1s to the right of these bits describes which product terms to use in the realization of the function. A 1 means to include the product term in the function, 0 means exclude the product term from the function – and is seen only for two or more outputs.

Putting both of Espresso's product terms together yields the optimal solution $F = A'C + AB$. The Kmap for this function and its SOP $_{\text{min}}$ solution is shown in the margin.

$A\backslash BC$	00	01	11	10
0		1	1	
1			1	1

$F(A, B, C) = A'C + AB$

Command line parameters can be included to specify options, changing the behavior of Espresso. All options consist of a minus sign, a letter, and a command, placed between "espresso" and the name of the input file. As an example the `-o eqntott` option will force Espresso to generate a symbolic output instead of the default PLA output.

```
> espresso -o eqntott simple.txt
#comments
F = (a&b) | (!a&c);
```

The entire set of options is listed by typing `espresso -help` at the command prompt. The `-epos` option inverts the truth table, yielding a SOP solution for F'. The POS solution is generated by inverting Espresso's output using the second helpful fact on page 54. This option allows a digital designer to quickly compare the cost of a SOP and POS realization.

Generated output can be saved by Espresso by redirecting standard output to a file. This option is indicated by putting a "greater than" symbol after a command, followed by the name of the output file. Think of the "greater than" symbol as a funnel, taking the streaming output of the Espresso program and funneling it into the output file. For example, the output of the previous Espresso example can be saved into a file called `simple.out` using the following command.

```
a:\> espresso -o eqntott simple.txt > simple.out
```

Espresso can solve design problems involving multiple outputs. The following Espresso file describes the example on page 51, where $F(A,B,C) = \sum m(1,4,5,6)$ and $G(A,B,C) = \sum m(1,2,3,5)$.

```
# File: harder.txt
# Name: <your name>
#       <course name>
# Date: Fall 2020
# Desc: F(a,b,c) = minterms(1,4,5,6)
#       G(a,b,c) = minterms(1,2,3,5)

.i 3
.o 2
.ilb a b c
.ob F G

000 00
001 11
010 01
011 01
100 10
101 11
110 10
111 00
.e
```

The output from Espresso is:

```
a:\> espresso harder.txt
# Comments
.i 3
.o 2
.ilb a b c
.ob F G
.p 3
1-0 10
01- 01
```

```
-01 11
.e
```

Espresso used three product terms to realize the functions AC', $A'B$, and $B'C$. The two columns to the right of the product terms describe which product terms are used to realize each outputs. Symbolically, the solution found by Espresso is:

- $F(A, B, C) = AC' + B'C$
- $G(A, B, C) = A'B + B'C$

The circuit diagram of the realization for the three functions is shown in Figure 3.4. Notice, the connections – denoted by dots – between the inputs and the AND gates has a pattern which is similar to the PLA description of the product terms by Espresso. Likewise, the pattern formed by the connections of the AND gates and OR gates is similar to the rightmost three columns of the Espresso output. This similarity is no accident; the designers of Espresso used a generalized layout of the of the circuit diagram in Figure 3.4 as a template to describe the output.

3.8 Exercises

1. **(6 pts.)** Design a circuit called DECODE. DECODE has two bits of input S, D and two bit of output $y_1 y_0$. If $S = 0$ then $y_0 = D$ and $y_1 = 0$ else if $S = 1$ then $y_0 = 0$ and $y_1 = D$.

   a) Write down the truth table for the DECODE function.

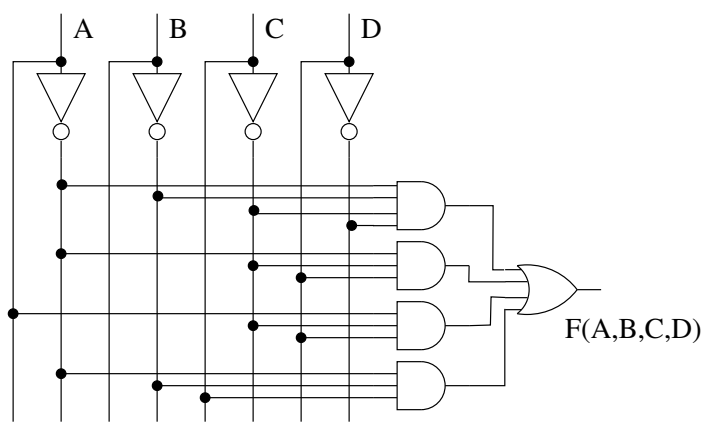
   b) Determine the SOP $_{\text{min}}$ realization for DECODE.

2. **(6 pts.)** Design a circuit called FULLADD. FULLADD has three bits of input a, b, c and two bits of output $s_1 s_0$. The output represents the sum of the three bits.

   a) Write down the truth table for the FULLADD function.

   b) Determine the SOP $_{\text{min}}$ realization for FULLADD.

3. **(4 pts.)** Determine SOP $_{\text{min}}$ expression for the following circuit and draw the circuit using the fewest number of gates possible.



4. **(8 pts.)** Design a digital system with four bits of inputs $I_3 I_2 I_1 I_0$ and two bits of outputs $O_1 O_0$. At least one of the inputs is always equal to 1. The output encodes the index of the most significant 1 in the input. For example, if $I_3 I_2 I_1 I_0 = 0101$, then the index of the most significant 1 is 2, hence $O_1 O_0 = 10$. Submit:

   - The truth table.
   - SOP $_{\text{min}}$ expression for O_1 and O_0.

5. **(8 pts.)** Design a 4-input $a_1 a_0 b_1 b_0$, 4 -output $O_3 O_2 O_1 O_0$ digital system. $A = a_1 a_0$ and $B = b_1 b_0$ represent 2-bit binary numbers. The output should be the product (multiplication) of the inputs, that is $O = A * B$. In addition to determining the output, determine the number of bits of output. Submit:

   - Truth tables.
   - Minimal SOP expression for the outputs.

6. **(8 pts.)** Design a 4-bit Gray-code to binary converter. A 4-bit gray-code is a sequence of 4-bit values where successive values differ by a single bit. For this problem use the sequence: 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000. The index of the 4-bit gray code is its binary value. For example, the 4-bit gray code 0111 is at index 5, therefore when presented with 0111 on its input, the converter should output 0101. Submit:

   - A truth table for the converter.
   - Four k-maps for the converter.
   - SOP $_{\text{min}}$ expression for the outputs, no product sharing please (use the `-Dso` command line option).
   - Espresso file for the converter
   - Espresso output in PLA format
   - Compare the number of gates required in your solution versus the number of gates required by Espresso.

7. **(4 pts. each)** Determine SOP $_{\text{min}}$ expression for:

   a) $F(A,B,C) = \sum m(0,1,3,4,5)$
   b) $F(A,B,C,D) = \sum m(1,5,6,7,11,12,13,15)$
   c) $F(A,B,C,D) = \sum m(0,2,5,6,8,11,12,13,14,15)$
   d) $F(A,B,C,D,E) = \sum m(0,8,9,10,13,15,22,26,29,30,31)$
   e) $F(A,B,C,D,E) = \sum m(0,2,4,5,7,10,13,15,18,21,24,26,28,29)$

8. **(4 pts. each)** Determine SOP $_{\text{min}}$ expression for:

   a) $F(A,B,C,D) = \sum m(4,7,9,12,13,15) + \sum d(0,1,2,3,10,14)$
   b) $F(A,B,C,D) = \sum m(0,1,5,7,10,14,15) + \sum d(2,8)$
   c) $F(A,B,C,D) = \sum m(0,1,3,4,15) + \sum d(10,12)$
   d) $F(A,B,C,D,E) = \sum m(2,3,5,7,11,13,17,19,29,31) + \sum d(1,4,9,16,25)$
   e) $F(A,B,C,D,E) = \sum m(2,3,6,10,12,13,14,18,25,26,28,29) + \sum d(11,27)$

9. **(8 pts. each)** Determine SOP $_{\text{min}}$ and POS $_{\text{min}}$ expressions for:

   a) $F(A,B,C,D) = \sum m(0,1,2,5,8,10,13,15)$
   b) $F(A,B,C,D) = \prod M(0,4,6,10,11,12)$
   c) $F(A,B,C,D) = \sum m(0,5,7,10,11,14) + \sum d(3,12,15)$
   d) $F(A,B,C,D) = \prod M(2,6,7,9,15) * \prod d(4,12,13)$
   e) $F(W,X,Y,Z) = WX'Z' + X'YZ + W'Y'Z + XYZ + WXY'$
   f) $F(W,X,Y,Z) = (W + X' + Y')(W' + Z')(W + Y')$

   Hint, the negation of a "Don't care" is a "Don't care".

10. **(3 pts.)** While grading homework for a digital design class the following question/answer pair is encountered. What is the problem with the answer given?

    Question: Generate the POS $_{\text{min}}$ expression for $F(A,B,C) = \sum m(2,3,4,5)$
    Answer: $F(A,B,C) = (A + B')(A' + B)$

11. **(6 pts.)** Determine the SOP $_{\text{min}}$ realization of the following function.

A	B	C	D	F(A,B,C,D)
x	1	1	x	0
0	x	0	1	0
x	x	0	0	x
x	0	1	x	1
1	x	0	1	1

12. **(6 pts.)** What is the worst function SOP $_{\text{min}}$ of 3 variable that can be created? That is, define a function whose minimal SOP form has the largest possible number of product terms. What is the largest number of product terms that a 4-variable SOP $_{\text{min}}$ expression can have? How about N variables?

13. **(16 pts.)** Sometimes a logic circuit needs to output a logic 0 in order to produce some behavior. For example, an LED can be attached to a digital circuit output so that it lights up when the circuit outputs a 0. This response is called an active low output; the output device is *active* then the digital output is *low*.

    Build a digital circuit that takes as input two 2-bit numbers, A and B. The circuit has three outputs which drive three LEDs labeled G, L, and E. The G LED should be illuminated when A>B. The L LED should be illuminated when A<B. The E LED should be illuminated when A=B. The LEDs are illuminated when the circuit outputs a 0, otherwise they are turned off.

    Determine SOP $_{\text{min}}$ expression for the G, L and E outputs. Determine POS $_{\text{min}}$ expression for the G, L and E outputs.

Chapter 4

Combinational Logic Building Blocks

In Chapter 1, Figure 1.1, a digital system was defined as a box that transforms binary inputs to binary output. This definition of a digital system was sufficient to introduce a wide variety of concepts but is a handicap, now. Following is a more detailed definition of a digital system.

> A digital system transforms the data inputs into data outputs. The transformation performed by the digital system is specified by the control inputs. The status outputs indicate any exceptional events occurring during the processing of the data.

Figure 4.1 shows a digital system with these four types of input and output, namely, data input, control input, data output, status output. This classification of inputs and outputs aids in the construction of complex digital systems using the datapath and control methodology.

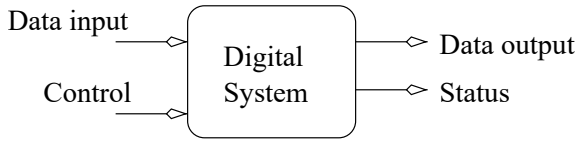


Figure 4.1: A modified diagram of a digital system showing the classification of the inputs and outputs.

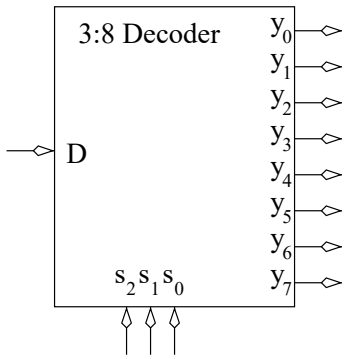
Some basic building blocks of digital systems are introduced. While a wide variety of blocks can be considered, those proven to be most useful in the construction of digital circuits are presented. If the right block for a particular task does not exist, create a new block using the methods presented in the last chapter. First in the examination of basic building blocks is a device that routes one bit of data to one of several outputs based on an address.

4.1 Decoder

Building Block: Decoder

Nomenclature:	N:M decoder
Data Input:	1-bit D
Data Output:	M-bit vector $y = y_{M-1} \ldots y_1 y_0$
Control:	N-bit vector $s = s_{N-1} \ldots s_1 s_0$
Status:	none
Behavior:	$y_s = D$ all other outputs equal 0

To understand this definition, examine an instance of a 3:8 decoder shown at left in Figure 4.2. Normally, the arrows indicating the direction of the information flow in to and out from the decoder are not drawn. To understand the behavior of the decoder, its truth table is shown to the right of Figure 4.2. Due to space constraints, only part of the entire truth table is shown.



s_2	s_1	s_0	D	y_0	y_1	y_2
0	0	0	0	0	0	0
0	0	0	1	1	0	0
0	0	1	0	0	0	0
0	0	1	1	0	1	0
0	1	0	0	0	0	0
0	1	0	1	0	0	1
0	1	1	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	0	1	0	0	0

Figure 4.2: A 3:8 decoder (left) and part of its truth table (right).

From the behavior listed in the description of the decoder, see that the ith output equals the data input, where i is the binary code of the select inputs. In other words, when $S = s_2 s_1 s_0 = 011_2 = 3_{10}$ and $D = 1$, then $y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0 = 00001000$. If $S = s_2 s_1 s_0 = 011_2 = 3_{10}$ and $D = 0$, then $y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0 = 00000000$.

The utility of this second case might be questionable because all the outputs are the same and consequently one cannot "see" where the output is being routed. The resolution to this dilemma requires considering the outputs through time. A decoder is a box which sends a "stream" of bits to some destination determined by the select lines. If the destination knows that it is receiving the stream, then it will be expecting both 1s and 0s through time.

The internal organization of a 3:8 decoder must process its four bits of input, consisting of a 3-bit select and a 1-bit data input, and eight bits of outputs. In the previous chapter, each bit of the output could be solved independently of the others. Hence, let's examine the y_0 output first. Examination of the truth table and the behavior of the decoder shows y_0 only equals 1 when $(s_2, s_1, s_0) = (0, 0, 0)$ and $D = 1$. Borrowing the minterm trick from page 29, $y_0 = s_2' s_1' s_0' D$ results. Every other output shares the characteristic that its output is equal to 1 for a single input. Thus, each output is represented by a minterm as shown in Figure 4.3.

In some digital applications, the need arises to build a larger decoder from multiple smaller decoders. This is done by fanning-out the data input in a tree-like structure. The following 4:16 decoder shows how a larger decoder is built from several smaller 2:4 decoders.

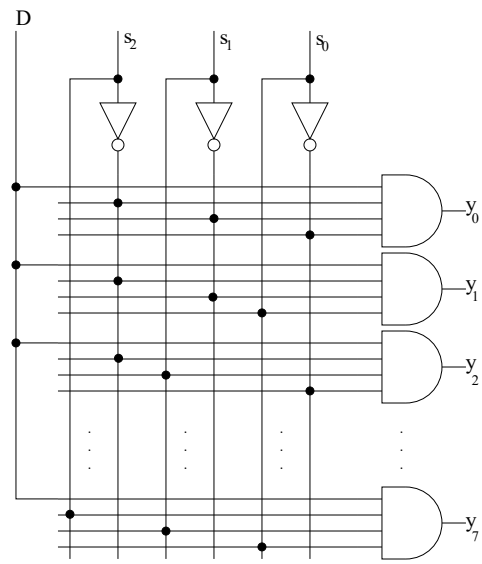


Figure 4.3: The internal organization of a 3:8 decoder.

In order to accommodate the 16 outputs, four 2:4 decoders are stacked on top of one another as shown in Figure 4.4. These four 2:4 decoders have a total of four bits of input. These four bits are sourced by the output of a single 2:4 decoder. The single data input of this decoder is the input of the overall 4:16 decoder.

Having organized the structure of the 2:4 decoders, all that remains is to route the select lines. The outputs of the decoder labeled **3** in Figure 4.4 corresponds to outputs $y_{15}y_{14}y_{13}y_{12}$ of the 4:16 decoder. Each of these outputs requires four bits of select with the form $s_3s_2s_1s_0 = 11xx$. Hence, $s_3s_2 = 11$ must route the data input, D, to decoder **3**. Hence, s_3s_2 must be the select to the first level decoder in Figure 4.4. A similar argument for the 2:4 decoders labeled **0,1,2** reinforces the fact that s_3s_2 must be the select to the first level decoder.

Data routed to output y_{12} has $s_3s_2s_1s_0 = 1100$. Thus, routing at the second level of 2:4 decoders seems to be controlled by s_1s_0. Examining all the other outputs reinforces this assumption for all the outputs.

4.2 Multiplexer

A multiplexer, often referred to as a mux, is data routing device which behaves exactly opposite of a decoder. Its structure and behavior is defined in the following table.

Building Block: Multiplexer

Nomenclature:	N:1 multiplexer
Data Input:	M-bit vector $y = y_{M-1} \ldots y_1 y_0$
Data Output:	1-bit F
Control:	$log_2(N)$-bit vector $s = s_{log_2(N)} \ldots s_1 s_0$
Status:	none
Behavior:	$F = y_s$

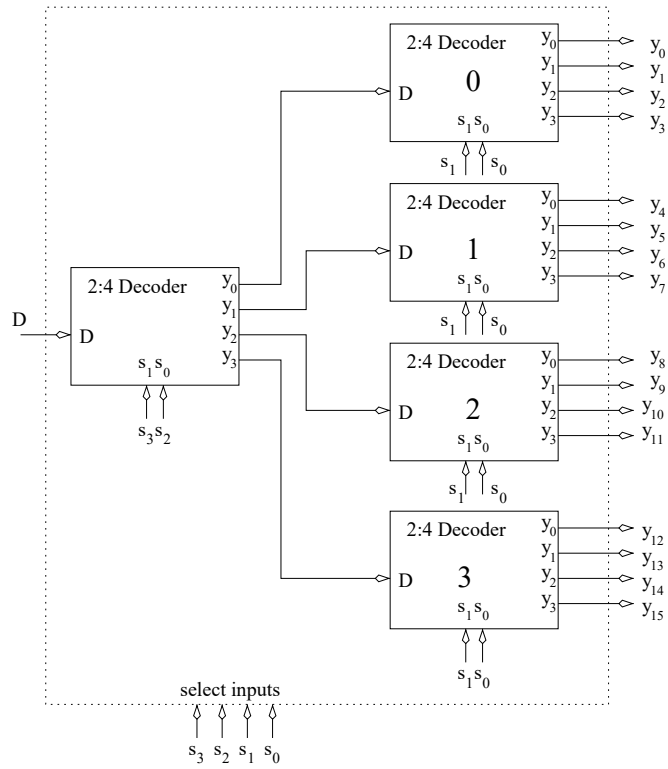
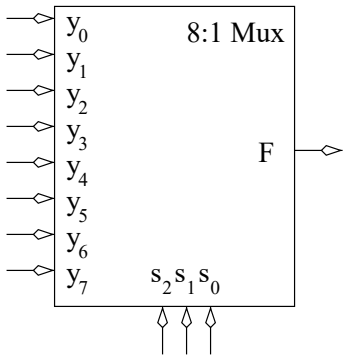


Figure 4.4: A 4:16 decoder built from 2:4 decoders.

To understand this definition, examine an instance of an 8:1 mux shown on the left in Figure 4.5. From the behavior listed in the description of the multiplexer, the ith data input is routed to the data output where the binary code of the select inputs is i. For example, if $S = s_2s_1s_0 = 101_2 = 5_{10}$ and $y_7y_6y_5y_4y_3y_2y_1y_0 = 00100000$, then $F = 1$.



s_2	s_1	s_0	F
0	0	0	y_0
0	0	1	y_1
0	1	0	y_2
0	1	1	y_3
1	0	0	y_4
1	0	1	y_5
1	1	0	y_6
1	1	1	y_7

Figure 4.5: An 8:1 mux (left) and its truth table (right).

The unusual form of the truth table shown in Figure 4.5 results from the fact that an 8:1 mux has a total of 11 inputs. There are a total of 2^{11} rows in this truth table making it infeasible to list every combination of the inputs. Consequently, in order to build an 8:1

mux, the structure of the truth table must be determined without listing every combination of inputs. Observe the y_0 input is routed to the output when $s_2s_1s_0 = 000$. Consequently, $F = s_2's_1's_0'y_0$ for this input. Each of the other outputs has a similar minterm form. Since only one of these "minterms" can equal 1 for a particular select input, the minterms are ORed together to form the output. The resulting internal organization of an 8:1 mux is shown in Figure 4.6.

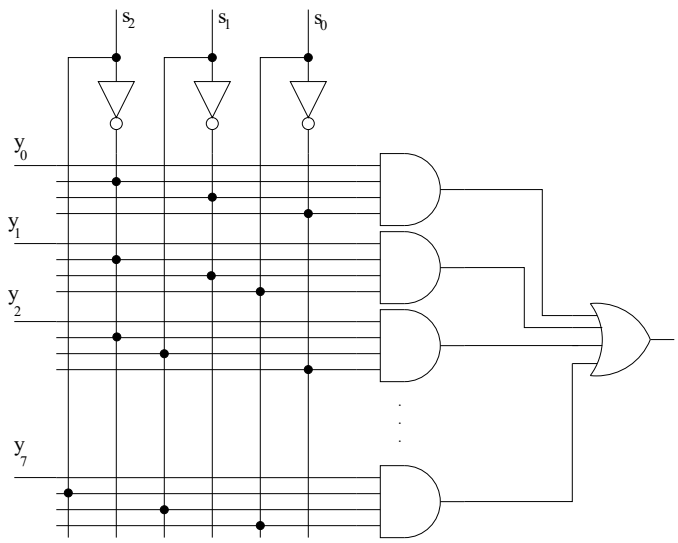


Figure 4.6: The internal organization of an 8:1 mux.

As with decoders, situations arise when a larger mux is built from smaller muxes. The key idea is to funnel down the many data inputs from smaller muxes to a single output. For example, construct a 16:1 mux from several 4:1 muxes.

In order to accommodate the 16 inputs, four 4:1 muxes are stacked on top of one another as shown in Figure 4.7. These four 4:1 muxes have a total of four bits of output. These four bits can nicely be routed by the inputs of a single 4:1 mux. The single data output of this mux is the input of the overall 16:1 mux.

The select lines are assigned to the 4:1 muxes based on the following argument. The inputs of the mux labeled **3** in Figure 4.7 corresponds to inputs $y_{15}y_{14}y_{13}y_{12}$ of the 16:1 mux. Each of these inputs requires four bits of select with the form $s_3s_2s_1s_0 = 11xx$. Hence, $s_3s_2 = 11$ must be the select for the output mux **4**. A similar argument for the 4:1 muxes labeled **0,1,2** reinforces the fact that s_3s_2 must be the select to the output mux.

In order to route y_{12} to the output, the select must equal $s_3s_2s_1s_0 = 1100$. Thus, routing at the input level of 4:1 muxes is controlled by s_1s_0. Examining all the other inputs reinforces this assumption.

Occasionally the need arises to construct a mux to handle "wide" data inputs, that is data inputs which consist of more than a single bit. A mux that can handle many bits is referred to as a *multibit mux*. A M-bit N:1 mux is defined as a N:1 mux whose data inputs and data outputs are M-bits wide. For example, the 4-bit 2:1 mux shown in Figure 4.8 has two data inputs and one data output each 4-bits wide. The internal organization of this mux is shown on the right-hand side of Figure 4.8. Four, 2:1 muxes are needed to accommodate all the data. Since the data inputs are to be handled as a single whole, they must be routed to the same input on each of the 2:1 muxes.

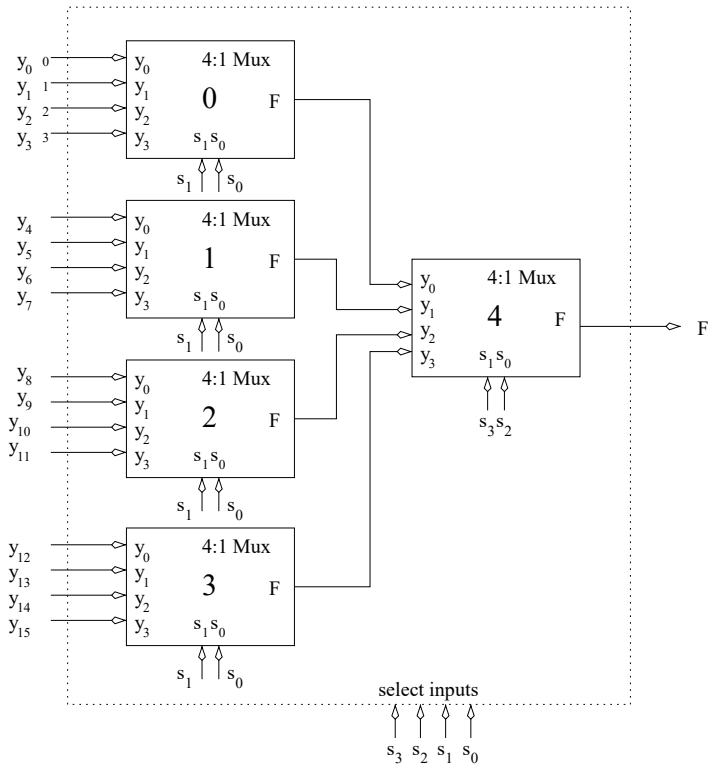


Figure 4.7: The construction of a 16:1 mux from 4:1 muxes.

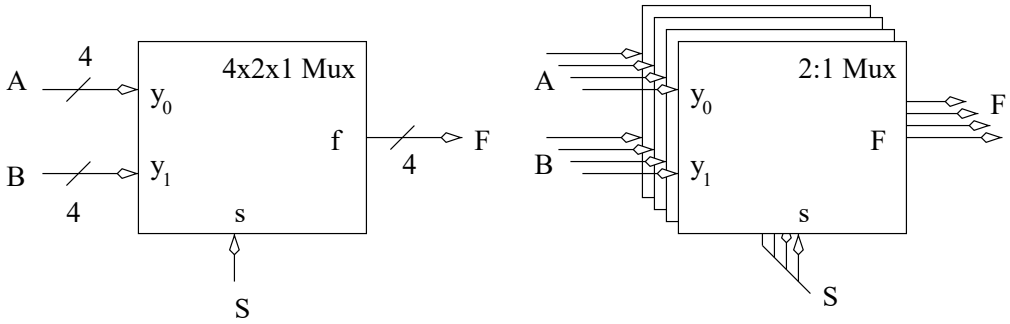


Figure 4.8: The organization of a 4-bit 2:1 mux. The slash labeled "4" denotes the fact that these signals are 4-bits wide.

4.3 The Adder

An N-bit adder is a circuit which adds two N-bit binary numbers A, B together generating an N-bit sum S and an overflow indication, Ovf.

Building Block: Adder

Nomenclature:	N-bit adder
Data Input:	two N-bit vectors A and B
Data Output:	N-bit vector sum
Control:	none
Status:	1-bit ovf
Behavior:	$sum = A + B$

The behavior of the adder is exactly as expected after reading page 5. Furthermore, the overflow output is asserted when the adder output is greater than or equal to 2^N. The construction of an adder illustrates an approach that can be utilized for circuits whose function can be broken down into modular pieces.

Approaching the design of a 4-bit adder ($N = 4$) using the design philosophy introduced in Chapter 2, the truth table has eight bits of inputs, 256 rows, and five bits of outputs. The task would be tedious, error-prone, producing a realization with a large number of gates. Instead of building the circuit as one monolithic piece, a module is designed that adds one set of bits together. Four of these modules are then connected in series to add four bits together. For example, in Figure 4.9 $A = 1011$ and $B = 0001$ are added together using the approach of page 5.

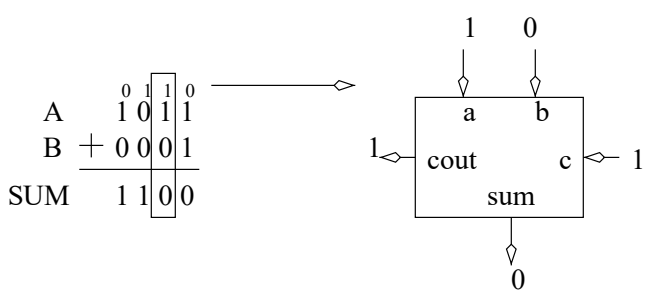


Figure 4.9: An addition problem and the values of one of its bit slices placed on a full adder.

This addition problem contains four *bit-slices* . Each bit-slice adds one bit's worth of the overall addition problem. The addition circuit is then constructed by stringing together four of these bit-slice adders (called full-adders). In the example shown in Figure 4.9, the second bit-slice of the addition problem is outlined. The inputs and outputs of this bit-slice are placed on a full adder. The full adder has three bits of input and two bits of output. To build a 4-bit adder, four full adders are strung together in series using the carry lines. The truth table for the full adder is created by enumerating every combination of inputs and determining the sum and carry-out for each. The carry-out and sum together are the 2-bit sum created by adding together the three input bits.

a	b	c	$cout$	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$a\backslash bc$	00	01	11	10
0			1	
1		1	1	1

cout = bc + a b + ac

$a\backslash bcin$	00	01	11	10
0		1		1
1	1		1	

sum=a b' c' + a'b'c + abc + a'bc'

The SOP $_{\text{min}}$ expression for the outputs is arrived at by solving Kmaps for each of the outputs. Once, the internal organization of a full adder is complete, four of them are connected together as shown in Figure 4.10 to build a 4-bit adder.

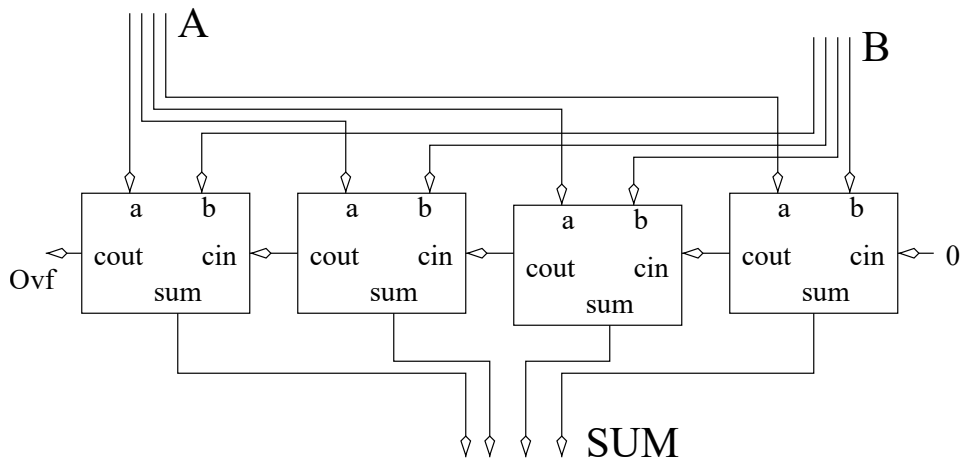


Figure 4.10: The arrangement of full adders to create a multi-bit adder.

The carry-out of each bit-slice becomes the carry-in of the next, more significant, full adder. This sequence is necessarily broken at the beginning and end of the chain of full adders. Since there is no carry-in to the least significant bit of an addition problem, the carry-in to the least significant full adder is set to 0. Since overflow occurs when the result of the addition process requires more bits than the word size, a carry-out from the most significant full adder indicates overflow.

4.4 The Adder Subtractor

As its name implies, an adder subtractor can perform two separate functions. From a black-box perspective, the only difference between this module and an adder is the presence of a control input to select which function the adder subtractor performs.

Building Block: Adder Subtractor

Nomenclature:	N-bit adder subtractor
Data Input:	two N-bit vectors A and B
Data Output:	N-bit vector s
Control:	1-bit f
Status:	1-bit ovf
Behavior:	if c=0 then $s = A + B$ else $s = A - B$

For now, assume the inputs and output of an adder subtractor are 2's-complement numbers. Addition of 2's-complement numbers proceeds like the addition of regular binary numbers. For now, ignore overflow conditions. The subtraction process for 2's-complement numbers, described on page 7, rewrites the subtraction problem $A - B$ as an addition problem $A + (-B)$. The caveat is that B must be negated. Since both the addition and the subtraction problem

require an adder, all that is required is to pass either B or $-B$ to the adder depending on which operation is to be performed. This idea is presented in Figure 4.11.

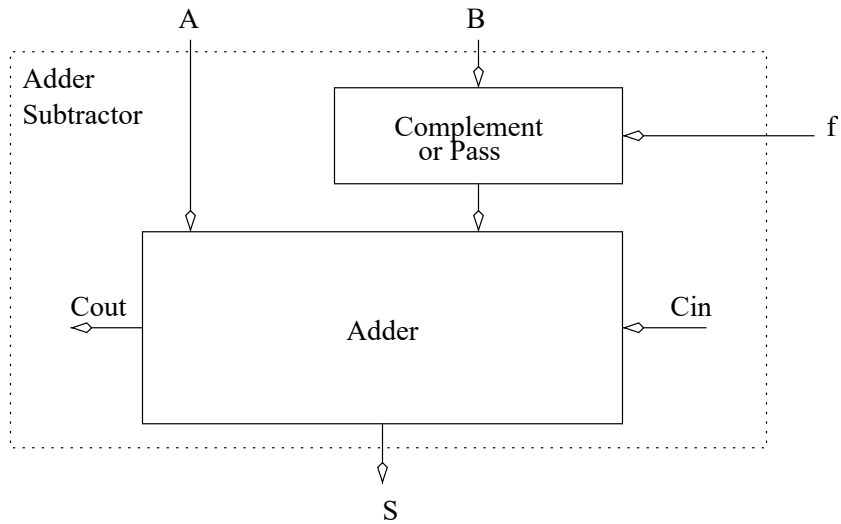


Figure 4.11: The idea behind the creation of an adder subtractor circuit.

The "Complement or Pass" box in Figure 4.11 produces either B when $f = 0$ (addition) or $-B$ when $f = 1$ (subtraction). The process of complementing B is to flip the bits and to add 1. To avoid adding a second adder to perform the "add 1" operation the adder shown in Figure 4.11 performs both $A + B$ and the "add 1". This little piece of magic is accomplished by hijacking the carry-in to the least significant bit of the full adder chain shown Figure 4.10. Instead of hardwiring *cin* shown in Figure 4.11 to 0, *cin* is connected to f. When $f = 1$, an extra 1 will be added to sum. The 2's-complement of B is computed in two separate steps: The "Complement or Pass" box negates the bits of B (when $f = 1$) and the adder adds 1 (when $f = 1$). Hence, the circuit performs a subtraction when $f = 1$. When $f = 0$, the "Complement or Pass" box will pass through B unchanged, the adder does not add anything extra to the inputs and, consequently, the circuit performs $A + B$.

The "Complement or Pass" box is broken down into bit-slices, each bit of B is sent to its own slice along with the f control bit. If $f = 0$, then the slice outputs the input B bit. If $f = 1$, then the slice outputs the complement of the B bit. The truth table is shown below.

b	f	out
0	0	0
0	1	1
1	0	1
1	1	0

Solving the truth table yields $out = b'f + bf' = b \oplus f$. Each of these bit-slices is organized along with the components in Figure 4.11 yielding the circuit diagram in Figure 4.12.

According to page 7, overflow in a 2's-complement operation occurs when the carry-in and carry-out to the most significant bit-slice of the addition disagree. In order to build a circuit to check this condition, the "Adder" box shown in Figure 4.12 must be opened revealing the chain of full adders as shown in Figure 4.11. The derivation of the truth table and the circuit realization is left as an exercise at the end of the chapter.

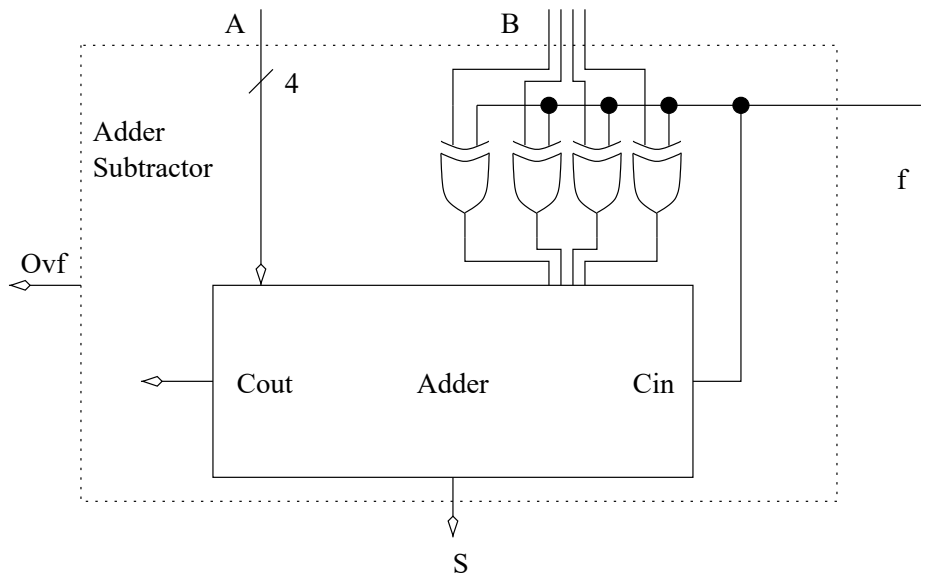


Figure 4.12: The construction of 4-bit adder subtractor.

4.5 The Comparator

A comparator is a device which determines the relative magnitude of its two inputs.

Building Block: Comparator

Nomenclature:	N-bit comparator
Data Input:	two N-bit vectors X and Y
Data Output:	none
Control:	none
Status:	1-bit G, L, E
Behavior:	

cond	E	L	G
$X = Y$	1	0	0
$X < Y$	0	1	0
$X > Y$	0	0	1

Comparators are rather unique because they lack data output. That is not to say, comparators do not have any output. Rather, their status outputs are quite useful. The comparator examines its two N-bit inputs, denoted X and Y, and outputs three bits describing their relative magnitudes. Each of these three bits, E, L, G, is asserted when X equals Y, X is less than Y, or X is greater than Y, respectively.

Much like the adder circuit the truth table method is not very useful for designing large comparators. For example, a 16-bit comparator has two 16-bit inputs, or 32 bits of inputs, for an astounding 2^{32} rows in the truth table.

The construction of large comparators is based on the method that employed to determine the relative magnitude of two numbers X and Y, working from the most to least significant

bits. At each step, compare a bit of X and Y. If the bits are equal, then continue to the next least significant bit. Otherwise, either X or Y is larger, and the comparison is over.

Large comparators are constructed by stringing together a series of modified 1-bit comparators as shown in Figure 4.13. Each bit-slice has as input a pair of bits from X and Y, and *Ein*, *Gin* and *Lin* signals from a more significant bit-slice. *Ein*, *Lin* and *Gin* tell a bit-slice the status of the magnitude of X and Y in the bit positions to its left. Each bit-slice has three bits of output, *Eout*, *Lout*, and *Gout*, communicating the relative magnitude of the inputs so far.

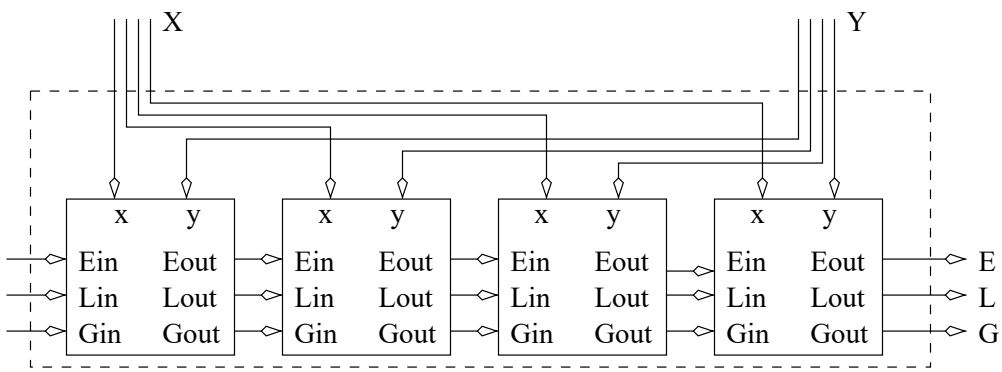


Figure 4.13: The arrangement of the bit-slices for the comparator.

The truth table for a bit-slice of the comparator has five inputs, x, y, Ein, Lin and Gin. Each bit-slice has three outputs, $Eout, Lout$ and $Gout$. A portion of the truth table is shown below.

x	y	*Ein*	*Lin*	*Gin*	*Eout*	*Lout*	*Gout*
0	0	0	0	0	x	x	x
0	0	0	0	1	0	0	1
0	0	0	1	0	0	1	0
0	0	0	1	1	x	x	x
0	0	1	0	0	1	0	0

With five bits of input, the truth table has a total of 32 rows, only five of which are shown above. Two rows have their outputs set to "don't cares" because they represent impossible situations. The inputs in the first row, state that the two numbers have no relative magnitude relative to one another. The inputs on the fourth row claim that $X < Y$ and $X > Y$ simultaneously, an impossible situation. The second row states that it has already been determined that $X > Y$, so it does not matter what x and y are because their value cannot effect a decision that has occurred in a more significant bit-slice. Similarly, the third row states that $X < Y$, so the outputs just propagate this fact, regardless of x and y, because the bits of this slice are less significant than those which decided $X < Y$. The fifth row states that so far $X = Y$ and since the bit-slice inputs are equal, the two inputs are still equal. The remainder of the truth table will be completed in an exercise at the end of the chapter.

The final question that must be answered is "What should *Ein*, *Lin*, and *Gin*, on the far left of the Figure 4.13, be assigned?" The answer is that the Ein, Lin, Gin inputs to the most significant bit-slice should be set to 1,0,0 because X and Y are initially assumed to be equal. From the preceding paragraph, observe that whenever it is determined that $X > Y$ or $X < Y$, the comparator bit-slices will not change this fact. Hence, starting the circuit off from any

other initial condition would cause an irreversible bias. Another way to view this situation is to realize that any binary number can be considered to have an infinite number of leading 0s, all of which are equal.

4.6 Wire Logic

When designing a digital system which contains a device such as an adder which manipulates a pair of N-bit inputs representing binary numbers, it is easy to forget that those N bits are really N separate wires being grouped together. As a digital designer, one has access to each of these wires and can manipulate them as necessary. This point is important to remember when designing with the basic building blocks in this chapter because the number of bits representing a value must match the size of the device input. For example, a 4-bit value cannot be run into an 8-bit-wide input and expected to work without some consideration of how to handle the four, remaining bit positions.

For example, assume a 4-bit binary number needs to be added to an 8-bit number. In order to accommodate the 8-bit number, an 8-bit adder needs to be used. Unfortunately, this decision creates a problem with the 4-bit operand because the upper four bits of its input are unoccupied. The solution is to fill in the upper four bits with 0s. This solution is called padding with 0s because just like when padding is placed around an item being shipped in a container to prevent it from moving around, padding a binary number with 0s keeps it aligned in its word-sized container. Further, the padding operation does not change the value of the 4-bit number.

Padding a 2's-complement number, a process called sign-extension, is slightly more complex. Consider the problem of adding or subtracting a 4-bit 2's-complement number to an 8-bit 2's-complement number using an 8-bit adder subtractor. The important point to keep in mind is that the value of the original, 4-bit, 2's-complement must be the same as the value of the sign-extended 8-bit 2's-complement value. If the 4-bit value were 1111, representing -1 in decimal, then padding the upper four bits with 0s would yield 00001111, which represents +15 in 8-bit 2's-complement. This approach is not the correct way to proceed because the original value of -1 was converted into +15 through the sign-extension process. Instead, the value should have been padded with 1s yielding 11111111, which represents -1 in an 8-bit 2's-complement number. To show that any negative 4-bit value padded with 1s retains its value, show that the values of original and sign-extended numbers are the same when the bits are flipped and 1 is added. Further, any positive 4-bit value should be padded with 0s to retain its value. In general, the 4-bit value needs to be padded with four copies of the most significant bit.

When representing a decimal value, using the base-10 numbering system, multiplication of a number by 10 can be accomplished by moving all the digits one position to the left and inserting a 0 in the vacated digit position. With binary-represented values, multiplication by 2 can be accomplished by moving all the bits one position to the left and inserting a 0 in the vacated position. This manipulation can be handled by padding the least significant bit position with a 0.

Another useful manipulation is to combine signals together. For example, consider a pair of 4-bit binary numbers are to be added together, and their 5-bit sum is to be reported. One alternative might be to pad each of the 4-bit inputs with a 0 and pass them along to a 5-bit adder. Alternatively, the numbers could be used unmodified as inputs to a 4-bit adder, and combine the overflow output of the 4-bit adder to the 4-bit sum output, producing a 5-bit result. Though unconventional, this manipulation is perfectly legal and perfectly correct because, the overflow signal is just the carry-out from the most significant full adder.

4.7 Combinations

The devices introduced in this section have limited utility when used by themselves. Their real potential is realized when combined together. When connecting two components, the data outputs of one device are typically connected to the data inputs of other. Likewise, the status outputs are typically connected to control inputs of another device. But how are the components arranged? A useful starting point is to phrase the solution of the design problem in terms of a simple algorithm. Algorithms are composed of statements. The algorithms to be constructed have several different types of statements.

Arithmetic Statements

Arithmetic statements perform the data manipulations required by design problems. An arithmetic statement consists of two parts, a left-hand side (LHS) and a right-hand side (RHS), related to one another by an equal sign. The RHS describes the operation and its inputs, while the LHS represents the variable denoting the output of the arithmetic operation. For example, the following line of code describes the addition of two values.

```
x = y + 3
```

The hardware realization of this line of code is an adder with `y` and 3 as inputs and `x` as its output. The algorithm does not specify the width of the `x` and `y` signals. These must be determined from accompanying information.

Conditional Statements

Conditional statements arise in programming languages in the form of `if/then/else` statements. All conditional statements consist of three parts, the condition to be checked (the `if` clause), the statement to be evaluated when the condition is true (the `then` clause), and the statement to be evaluated when the condition is false (the `else` clause).

Typically, the condition being evaluated seeks the relative magnitude of two binary numbers. For example, consider checking whether `(a<4)`. This comparison can be realized by routing `a` and `4` into the x and y inputs of a comparator and using the L output.

The consequence of the condition is to cause the evaluation either of the `then` clause or of the `else` clause. For now, these clauses will be arithmetic statements. In order to illustrate the hardware realization of a conditional statement, consider the following example.

```
if (a<4) then x=y+3 else x=y+7
```

The solution to the conditional assignment statement utilizes a comparator to determine the relative magnitudes of `a` and `4`. It is important to note in the solution shown in Figure 4.14 which of the comparator's inputs is x and y. Of the three status outputs, the L signal is used as the select on the multiplexer's inputs. The other two comparator outputs should not be shown since they are not used.

Each of the arithmetic operations in the algorithm is realized by its own adder. Notice in Figure 4.14 that both adders compute their values, and the job of the multiplexer is to select one of the adder outputs based on the results of the L output from the comparator. Since the LHS of both assignment statements involved the same variable, the output of the mux is labeled with x. When `a<4`, then $L = 1$ and, consequently, the y_1 output of the mux is routed to the output. According to the algorithm, when `a<4`, then $x = y + 3$. Consequently, $y + 3$

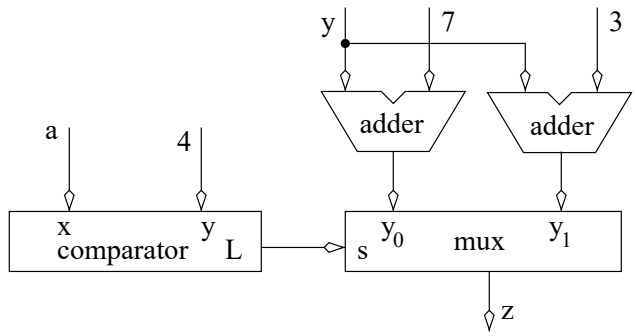


Figure 4.14: A combination of components to realize a conditional assignment statement.

should be routed to the y_1 input of the mux. Now, only $y+7$ is left to be routed to the y_0 input of the mux. It is important to annotate the mux inputs with y_1 and y_0 in order demonstrate that the solution works correctly.

The previous solution is more complex than it needs to be. An adder can be removed from the circuit by noting that in the RHS of both assignments, the variable y has either 3 or 7 added to it. Consequently, a mux can be used to switch through either a 3 or 7 and then to add this mux's output to y as shown in Figure 4.15.

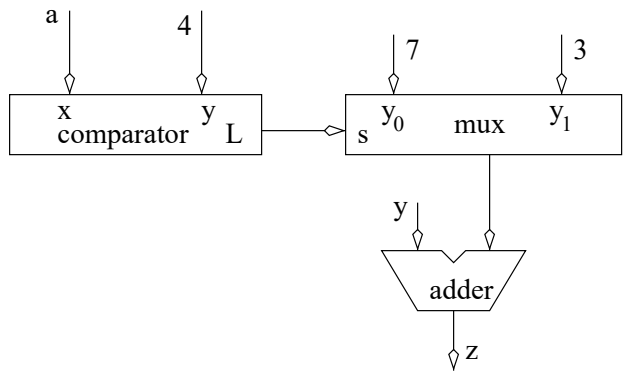


Figure 4.15: A better realization of the conditional assignment statement.

Make it a habit to identify ways to reduce the complexity of a circuit whenever possible. After all, one of the core principles of engineering is always endeavor to do the most with the least.

4.8 Exercises

1. **(2 pts. each)**Short answer:

   a) How many 3:8 decoders would it take to build a 9:512 decoder?

   b) How many AND gates are there in a 2^N:1 mux?

   c) How many AND gates are there in a 2^N : 1 mux which is constructed out of 2:1 muxes?

   d) How many AND gates are there in a 2^N:1 mux which is constructed out of 2^L:1 muxes, assume that 2^N is an integer multiple of 2^L?

2. **(6 pts.)**Determine the SOP $_{\text{min}}$ expression for each of the three outputs of a bit-slice of the comparator.

3. **(2 pts.)**Show how to connect together four 4-bit comparators to construct a 16-bit comparator.

4. **(2 pts.)**Determine the circuitry for the overflow detection circuit for a 2's-complement adder subtractor. See page 7.

5. **(10 pts.)**Build a BCD to 7-Segment Display converter using Espresso.

Building Block: BCD to 7-segment

Nomenclature:	BCD to 7-segment converter
Data Input:	4-bit vector $D = d_3d_2d_1d_0$
Data Output:	7-bit vector $Y = y_6 \ldots y_1y_0$
Control:	none
Status:	none
Behavior:	The output drives a 7-segment display pattern representing the BCD digit.

A binary coded digit (BCD) is a 4-bit binary number that is constrained to assume the values of 0-9. That is, 1010 ... 1111 are illegal BCD digits.

A 7-segment display is a box with seven inputs and seven output LED bars. Each input is wired to an LED bar that is illuminated when a 1 is applied to its input. Each of the seven LED segments is numbered according to the pattern shown on the left-hand side of Figure 4.16.

The pattern of LEDs to illuminate for each BCD digit is shown on the right-hand side of Figure 4.16. A BCD to 7-segment converter has four inputs, $d_3d_2d_1d_0$ and seven outputs $S_7 \ldots S_1$. Complete the design using Espresso. Make sure to include "Don't cares" in the truth table specification.

a) Use Espresso to determine the SOP $_{\text{min}}$ expression for the outputs $S_7 \ldots S_1$. Underline product terms that are shared. Submit the Espresso source file.

b) Use Espresso to determine the POS $_{\text{min}}$ expression for the outputs $S_7 \ldots S_1$. Underline sum terms that are shared. Submit the Espresso source file

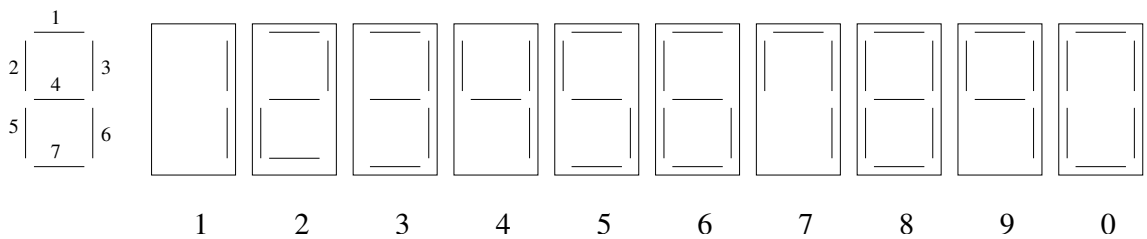


Figure 4.16: The numbering of the segments in a 7-segment display. The patterns of the BCD digits.

6. **(10 pts.)** Build a box which has one 4-bit input called A and one 4-bit output called T. The output T is the 2's-complement value of the input A. Use the bit slice paradigm to solve this problem. That is, create a building block for one bit of the problem then string four of them together to solve the problem. For the problem at hand this can be done as follows:

   a) Start at the LSB of A.
   b) If this is the first, least significant, 1, flip all bits to the left.
   c) If this is not the first 1, leave the bit alone.
   d) Move one bit to the left.
   e) Goto Step b.

   A bit-slice should communicate whether there has been a 1 to the right, to the more significant bit. Submit:

   - How the above "algorithm" behaves when presented with the inputs A=1100
   - The truth table for one bit slice
   - SOP $_{\text{min}}$ expression and circuit diagram for a bit slice.
   - The organization of four bit slices to solve the problem

7. **(4 pts.)** Build a 7:128 decoder using a minimum number of 4:16, 2:4 and 1:2 decoders. Describe the wiring of the select lines.

8. **(4 pts. each)** Design a circuit with two 8-bit inputs X, Y, an 8-bit output Z and a 1-bit input *sel*. Construct a circuit that yields the correct value of Z using only the basic building blocks presented in this chapter; do NOT show the internal organization of these building blocks. If a mux is used, denote which input is the y_0 and which is y_1. If a comparator is used denote which input is X and which is Y. Do not use any AND or OR gates; it will tempting in the later problems.

   a) `if (sel==0) then Z = X else Z = Y`
   b) `if (sel==0) then Z = X+Y else Z = Y`
   c) `if (sel==0) then Z = X+Y else Z = X-Y`
   d) `if (X==0) then Z = X else Z = Y`
   e) `if (X==Y) then Z = X-Y else Z = Y`
   f) `if (X==Y) then Z = X+Y else Z = X-Y`

g) `if (X < Y) then Z = X else Z = Y`

h) `if (X <= Y) then Z = X else Z = Y`

i) `if (X > Y) then Z = X else Z = Y`

j) `if (X > Y) then Z = X+X else Z = Y+Y`

9. **(10 pts.)** Build a 4-bit priority encoder.

Building Block: Priority Encoder

Nomenclature:	N-bit priority encoder
Data Input:	N-bit vectored $D = d_{N-1} \ldots d_1 d_0$
Data Output:	$\log_2(N)$-bit vector $Y = y_{log_2(N)} \cdots y_1 y_0$
Control:	none
Status:	none
Behavior:	$F = i$ where i is the highest indexed input which equals 1. When all inputs equal 0, the output is a "don't care".

The idea is for the outputs to represent (in binary code) the highest input index which equals 1. For example, a 4-bit priority encoder with input $D = 1010$ has inputs $d_3 = 1$ and $d_0 = 1$. Of these two inputs, the index of d_3 is greater than the index of d_0 so the output, F is equal to 3, or in binary 11. If the input were $D = 0111$ then $F = 10$.

a) Write down the truth table for a 4-bit priority encoder. Hint, the truth table could be structured so that it contains only five rows by using "don't cares" on the inputs.

b) An SOP $_{\text{min}}$ realization of the circuit.

10. **(10 pts.)** Build a 4-bit saturation adder. A saturation adder performs normal 4-bit addition when the resulting sum is less than 15. If the sum is greater than 15, the saturation adders outputs 15. The following table summarizes.

Building Block: Saturation Adder

Nomenclature:	4-bit saturation adder
Data Input:	2, 4-bit vectors `A, B`
Data Output:	4-bit vector `sum`
Control:	none
Status:	none
Behavior:	`if (A+B > 15) sum = 15` `else sum = A+B`

Submit a schematic showing the basic building blocks, their data status, and control interconnections. Show any truth tables used to build glue logic.

11. **(10 pts.)** Build a mod-6 adder. The mod-6 adder takes as input two 3-bit (mod 6) numbers and adds them together modulus 6.

    Modular arithmetic only operates with a limited portion of the integers. The range of numbers is $\{0, 1, 2, \ldots, m-1\}$ where m is called the *modulus*; note there are m different integers because counting started at 0. For example, when working in mod-6 arithmetic use the integers $\{0, 1, 2, 3, 4, 5\}$. To solve any addition problem in modular arithmetic, it is only necessary to perform regular addition with the special rule that the addition process rolls over from the largest number, $m-1$ to 0 when the result is larger than $m-1$. For example, in mod-6 arithmetic $(5+1) \mod 6 = 0$. The statement " mod 6" is always included in the addition problem to indicate to the reader that mod-6 arithmetic is being performed. Here are a few more examples to help

    $2 + 3 \mod 6 = 5$
    $3 + 3 \mod 6 = 0$
    $4 + 3 \mod 6 = 1$
    $5 + 5 \mod 6 = 4$

Nomenclature:	3-bit mod 6 adder
Data Input:	two, 3-bit (mod-6) vectors `A, B`
Data Output:	3-bit (mod-6) vector `sum`
Control:	none
Status:	none
Behavior:	`sum = A+B mod 6`

    Submit a schematic showing the basic building blocks, their data status, and control interconnections. Show any truth tables used to build glue logic. Be careful that the word size of the result is handled correctly.

12. **(1pt. each)** Convert the following to 2's-complement assuming a word size of eight bits.

    a) -35
    b) -128
    c) 67
    d) 128

13. **(1 pt. each)** Perform the following operations for the given 2's-complement numbers. Assume a word size of eight bits in all cases. Indicate where overflow occurs. If there is no overflow, convert the result to decimal.

    a) 01011101 + 00110111
    b) 11101011 + 11110001
    c) 01011101 + 10101011
    d) 10111011 - 11110001
    e) 01011101 - 00110111
    f) 01011101 - 10101111

14. **(10 pts.)** Build a flip box. A flip box is defined by the following input, output, and behavior definition.

Nomenclature:	8-bit flip box.
Data Input:	8-bit $D = d_7 \ldots d_0$
Data Output:	8-bit $F = f_7 \ldots f_0$
Control:	3-bit $S = s_2 s_1 s_0$
Status:	none
Behavior:	The output is the same as the input except for one bit which is inverted. The index of the inverted bit is given by S.

The flip box takes the 8-bit data input, flips a single bit identified by S, then sends the new 8-bit value to the output. For example, if $D = 11110000$ and $S = 010$ then $F = 11110100$. If $D = 11110000$ and $S = 101$ then $F = 11010000$. The solution should rely heavily on the basic building blocks.

15. **(10 pts.)** Build a box which recognizes some keyboard scancode. When a key is pressed on a keyboard, the keyboard transmits (among other things) an 8-bit scancode of the pressed key. Each key has its own scancode listed in Table 4.1. The relationship between the keys and their scancode is not based on ASCII.

Key	scancode	Key	scancode	Key	scancode	Key	scancode
0	45_{16}	1	16_{16}	2	$1E_{16}$	3	26_{16}
4	25_{16}	5	$2E_{16}$	6	36_{16}	7	$3D_{16}$
8	$3E_{16}$	9	46_{16}	A	$1C_{16}$	B	32_{16}
C	21_{16}	D	23_{16}	E	24_{16}	F	$2B_{16}$
P	$4D_{16}$	L	$4B_{16}$	M	$3A_{16}$	I	43_{16}

Table 4.1: IBM 6450225 Keyboard scancodes.

Nomenclature:	scancode classifier
Data Input:	8-bit $D = d_7 \ldots d_0$
Data Output:	IsP, IsL, IsM, IsI, IsS
Control:	none
Status:	none
Behavior:	IsP =1 when D is the scan code for the letter "P". IsL =1 when D is the scan code for the letter "L". IsM =1 when D is the scan code for the letter "M". IsI =1 when D is the scan code for the letter "I".

16. **(10 pts.)** Build a box which converts an 8-bit scancode for a hexadecimal digit into a 4-bit hexadecimal values.

Nomenclature:	scancode classifier
Data Input:	8-bit $D = d_7 \ldots d_0$
Data Output:	4-bit $H = h_3 h_2 h_1 h_0$
Control:	none
Status:	none
Behavior:	Converts the scancode D, representing a the key of a hexadecimal character, into its 4-bit value H.

For example, if $D = 25_{16}$, the scancode for the "4" key, then the converter should output $H = 0100_2$. Assume that the inputs are always legal hexadecimal scancodes.

Chapter 5

Sequential Circuits

The preceding four chapters have focused on *combinational circuits*, digital systems whose output depends solely on the input. That is,

$$output = F(input)$$

A good example of a combinational circuit is an adder. When the inputs 3 and 2 are applied to a 3-bit adder, the output is always 5 regardless of what inputs were applied to the circuit in the past. However, another whole class of circuits remembers events in the past and adjusts their outputs to reflect these events.

The output of a *sequential circuit* to a given input depends on what inputs were applied in the past. A good example of a sequential circuit is a counter: a digital circuit that outputs a count value, increasing every time a "count up" signal is asserted. Clearly, the same input (telling the counter to count up) may elicit many different outputs depending on the current count value. A desktop PC is another good example of a sequential circuit. The same input (clicking a mouse) may cause an English paper to be saved or an alien invader to be destroyed. Thus, the relationship between the inputs and outputs of a sequential circuit is more complex than in a combinational circuit and needs to be described in a fundamentally different way. In order to relate the inputs of a sequential circuit to the output, the concept of a state variable needs to be introduced. In some sequential circuits, the state of a system can be thought of as a memory of the past inputs. In other sequential circuits representing a process, it is best to regard the state as the current step within that process. The relationship between the output of a sequential circuit is described as,

$$output = F(input, state)$$

A digital circuit which controls a home heating furnace is examined in order to emphasize the differences between combinational and sequential circuits as well as to clarify the concept of state.

When the temperature is below T_{set}, the furnace should be on and when the temperature is above T_{set}, the furnace should be off. The furnace controller circuit has 1-bit of output, when it is 0, the furnace turns off and when its output is 1, the furnace turns on. The input to the furnace controller comes from a temperature sensor which outputs a 0 when the temperature

is below T_{set} and outputs a 1 when the temperature is above T_{set}. This behavior is captured in the graph shown in Figure 5.1.

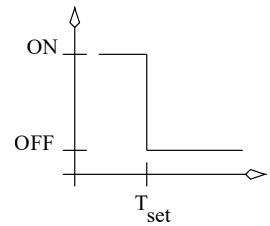


Figure 5.1: A simple furnace controller for a house.

The y-axis of this graph describes the state of the furnace; the furnace is either on or off. The x-axis describes the temperature of the house. If the temperature is less than T_{set}, then the furnace is on. If the temperature is greater than T_{set}, then the furnace is off. It should not be hard to surmise that the digital circuit required to control the temperature is a NOT gate. While this may seem like a satisfactory solution to the temperature control problem, it would perform quite poorly in practice. It is instructive to examine why.

Assume the house was initially cold when the furnace controller was turned on. Clearly, the controller would turn on the furnace and warm the house up. At some point, the temperature in the house would warm up to T_{set} causing the controller to turn off the furnace. Because of the residual heat in the furnace the temperature may overshoot T_{set}. Soon however, the temperature would drop below T_{set} causing the controller to turn the furnace on. This time however only a small amount of heat would be required to raise the temperature of the house above T_{set}. In a short time, the controller would turn the furnace off. Since little heat had been added to the house, the house would quickly cool down. Thus, starting a cycle where the furnace would turn on and off rapidly. While the temperature of the house would be held right near T_{set}, the continuous cycling of the furnace would be annoying to the occupants as well as introducing wear on the furnace. In addition, the controller would be susceptible to thermal noise in the environment. Any slight disturbance might cause air to pass over the temperature sensor causing the furnace to switch on and off.

The shortcomings of the controller are addressed utilizing the principle of hysteresis. The design is changed to have two temperature sensors called hi-sensor and low-sensor, each having its own set points T_{hi} and T_{low}, respectively. From their names it should be clear that $T_{hi} > T_{low}$. The hi-sensor outputs a 1 when the temperature is greater than T_{hi}; otherwise it outputs 0. The low-sensor outputs a 1 when the temperature is greater than T_{low}; otherwise it outputs 0. The new controller turns the furnace on when the house temperature drops below T_{low} and turns off the furnace when the house temperature rises above T_{hi}. When the temperature is between T_{hi} and T_{low}, the furnace continues whatever it was doing. The behavior of the new controller is described by the graph shown in Figure 5.2.

The y-axis of this graph describes the state of the furnace; either it is on or it is off. The x-axis describes the temperature of the house. If the house temperature is less than T_{low}, then the furnace is on. If the house temperature is greater than T_{hi}, then the furnace is off. However, when the house temperature is between T_{hi} and T_{low}, then the behavior of the furnace depends on what it was doing before the temperature reached this value. If the furnace was on, then it continues to be on while it passes through the region T_{low} to T_{hi}. This behavior is the upper horizontal line with the right-facing arrow in Figure 5.2. If the furnace was off, then it continues to be off while in the region T_{low} to T_{hi}. This behavior is the horizontal line with the left-facing arrow in Figure 5.2.

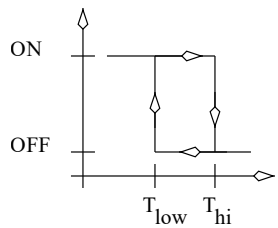


Figure 5.2: A more complex furnace controller for a house.

The graph in Figure 5.2 describes a relation because for a given x value there can be more than one $y = f(x)$ value. In order to describe the relationship between the input and output, a variable storing the *state* of the furnace is needed. The output of the controller is then a function of the input and the state. Instead of using the graph shown in Figure 5.2 to describe the behavior of the furnace controller, digital designers utilize *state diagrams*.

A state diagram is a diagram which describes the behavior of a sequential circuit (a digital system with states). In a state diagram, each of the states is given a name and placed inside a circuit. States are then connected together with arcs. States **A** and **B** are connected together with an arc pointing from **A** to **B** if there is an input which causes the system to transition from state **A** to state **B**.

The furnace controller has two states, "on" and "off" as shown in Figure 5.3. The arc from "on" to "off" is labeled T_{hi} because when T_{hi} is true the furnace is turned off. The "*" in the off state indicates that it is the reset state; the state the system starts in when first turned on.

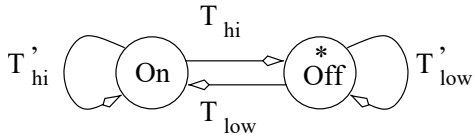


Figure 5.3: A state diagram describing the furnace controller of Figure 5.2.

In general, when building a state diagram, visit each state and identify the transition the system makes for every possible input combination. For example, "What state should the controller go into when it is currently in the on state and T_{hi} is false and T_{low} is false?" Since this case means that the temperature is below T_{low}, the furnace should stay in the on state. Notice that this transition is not shown in Figure 5.3. This is an instance of the general principle used in the construction of all state diagrams; any input combination not explicitly shown on the state diagram is assumed to not cause a change of state. This principle helps reduce the clutter on state diagrams by removing a multitude of arcs beginning and ending in the same state. The state diagram in Figure 5.3 also exhibits two other properties, completeness and consistency, necessary for its correctness.

Completeness - The logical OR of the conditions on all the outgoing arcs must equal 1. This implies that every possible combination of the variables used on the outgoing arcs has been described. In other words, the transitions out of a state have been completely specified. For example, state **A** in Figure 5.4 is complete because $x + x'y + x'y' = x + x'(y + y') = x + x' = 1$. State **B** in Figure 5.4 is not complete because $x + xy + x'y' = x(1 + y) + x'y' = x + x'y' \neq 1$.

Unequivocal - The logic AND of the conditions on any pair of arcs must equal 0. This implies that no pair of arcs include the same input condition. If two arcs included the same input condition, then this would make the decision of which state to go to next ambiguous or equivocal. For example, state **A** in Figure 5.4 is unequivocal because $(x)(x'y) = 0$, and $(x)(x'y') = 0$, and $(x'y)(x'y') = 0$. State **B** in Figure 5.4 is not unequivocal because $(x)(xy) = xy$. If the circuit was in state **B** and the input $(x, y) = (1, 1)$ arrived, then the arc labeled x and the arc labeled xy are both true and would both be taken. However, the circuit may only be in one state at a time, leaving it to decide which true condition to accept.

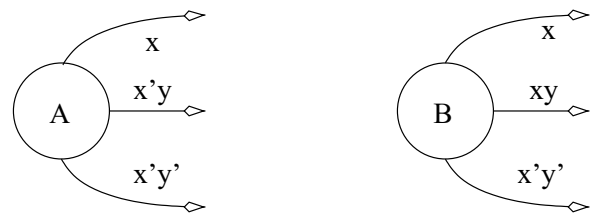


Figure 5.4: A portion of a state diagram showing two states. State **A** is complete and unequivocal, while state **B** is neither complete nor unequivocal.

A state diagram in which every state is complete and unequivocal is completely unambiguous. In every state, for any combination of inputs, there is exactly one next state. In order to build machines whose behavior can be described by state diagrams, a circuit element is needed which can store information. These circuit elements, called the basic memory elements, are the sequential circuit analogy of the AND, OR, and NOT gates to combinational logic.

5.1 Basic Memory Elements

The fundamental building blocks of sequential circuits are basic memory elements. A basic memory element has the ability to store one bit of information. This stored bit is the state of the basic memory element. The state is always available on an output signal called Q. There are a variety of basic memory elements; they are differentiated by two fundamental characteristics:

1. **Clock** - Basic memory elements are characterized by when they sample their inputs. Having tight restrictions on the times when a memory element can sample its inputs greatly simplifies the design of sequential circuits. A *clock* input is used to tell the basic memory element when it should sample its data input(s). A digital clock is a signal which changes its logic level between 0 and 1 with a fixed frequency. A basic memory element can be either a latch, clocked latch, or a flip flop.

   a) Latches: A latch continuously samples its inputs. Latches do not have a clock input.

   b) Clocked latches: A clocked latch samples its inputs when the clock input equals 1. When the clock input equals 0, the clocked latch does not change the currently stored bit.

c) Flip flops: A flip flop samples its input when the its clock input rises. The clock is said to rise when it goes from a logic 0 to a logic 1. When the clock input is not rising, the flip flop does not change the currently stored bit.

2. **Data** - Basic memory elements are characterized by their data inputs and how the inputs are transformed into the stored bit value; the clock is not considered a data input. A basic memory element can be either a D, T, SR, or JK device.

D	Q+
0	0
1	1

T	Q+
0	Q
1	Q'

S	R	Q+
0	0	Q
0	1	0
1	0	1
1	1	x

J	K	Q+
0	0	Q
0	1	0
1	0	1
1	1	Q'

a) A D memory element has a single data input called D; the D stands for *data*. When the D memory device samples its input, it stores this value.

b) A T memory element has a single data input called T; the T stands for *toggle*. If $T = 1$ when the input is sampled, then the T device toggles (flips) its stored bit. If $T = 0$ when the input is sampled, then the T device holds its current stored bit.

c) A SR memory element has two data inputs called SR; the S stands for *set* and the R for *reset*. If $SR = 00$ when the input is sampled, then the SR device retains its stored bit; the device *holds* its stored bit. If $SR = 01$ when the input is sampled, then the SR device stores a 0; the device *resets* its stored bit. If $SR = 10$ when the input is sampled, then the SR device stores a 1; the device *sets* its stored bit. The input of a SR device should never be set to 11.

d) A JK memory element has two data inputs called J and K. They have no good acronym, but act very much like S and R, respectively. If $JK = 00$ when the input is sampled, then the JK device retains its stored bit; *holds*. If $JK = 01$ when the input is sampled, then the JK device stores a 0; *resets*. If $JK = 10$ when the input is sampled, then the JK device stores a 1; *sets*. If $JK = 11$ when the input is sampled, then the JK device toggles its stored bit value; *toggles*.

The information on how a basic memory element processes the input into the stored bit is summarized in the state tables shown in the margins. A state table is a truth table describing the changes in state of a sequential circuit. Remember that the state of a basic memory element is the value of its stored bit. Notice the state column is headed by Q^+. The "+" indicates the state to be assumed in the (very) near future.

The type of basic memory element being used in a circuit diagram is indicated by the labels present on the data inputs and the notation present on the clock input. A latch does not have a clock input, a clocked latch has the label "clk" on its clock input, while a flip flop's clock input has a triangular notch accompanying the label "clk". Figure 5.5 shows a D latch, D clocked latch, and a D flip flop. Whenever an input is accompanied by a triangular notch it means that the input is *edge sensitive* .

Edge sensitive means that the input responds (or is sensitive) to changes in the logic level of the input, not the level of the of input. There are two types of edge sensitive inputs, positive and negative. An input sensitive to positive edges responds to an input transition from logic 0 to 1, a positive change in the logic level. An input sensitive to a negative edge responds to an input transition from logic 1 to logic 0, a negative change in the logic level. Input which

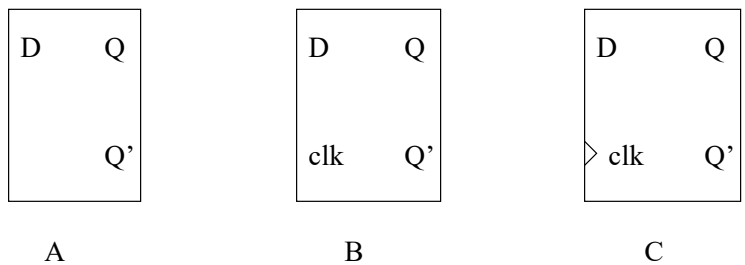


Figure 5.5: Different symbols are used to describe *when* a basic memory element samples its input. A) D latch. B) D clocked latch. C) D flip flop.

are sensitive to negative edges have a inversion "bubble" placed to the left of the triangular notch, like that shown in Figure 5.8.

Forming every combination of the two characteristics of basic memory elements yields 12 variations. These variations are organized by listing the data characteristics as column headers and listing the clock characteristics as rows headers. The resulting table is shown below.

	Latch	Clocked Latch	Flip Flop
D			
T			
SR			
JK			

While this table encompasses every possible basic memory element, several of them are not built either because they exhibit non-deterministic behavior or because their implementation is trivial. Before delving into these issues, a well-behaved basic memory element, the D clocked latch, is examined.

5.2 The D Clocked Latch

Understanding the behavior of the D clocked latch requires an understanding of its two characteristics, its clock input and its data input. Being a latch, it samples its data inputs when the clock is 1 and holds its stored bit when the clock is 0. Being a D device, its sampled value becomes the stored bit. Putting these two characteristics together, a D clocked latch is defined as a memory element that samples its D input continuously while the clock equals 1, stores this value as the state, and outputs it on the Q output. In other words, when the clock equals 1, the Q output follows the D inputs. When the clock input is 0, the D clocked latch ignores the D input and holds the value of its stored bit. Hence, when the clock is 0, the Q output does not change.

To better understand the behavior of the D clocked latch, examine how the device responds to inputs through the timing diagram in Figure 5.6.

Initially, the value of Q is given as 0. The initial value of any basic memory element on power-up is an important issue to be addressed later. For now, assume the D clock latch starts off in the 0 state. From time=0 to time=10, the clock input is 0. Hence, Q holds its value of 0. When the clock input goes to logic 1 at time=10, the device immediately starts sampling the D input, storing this value as the state, and then making the state available as the Q output. From time=10 to time=40, the clock equals 1 and the Q output follows the D input. From

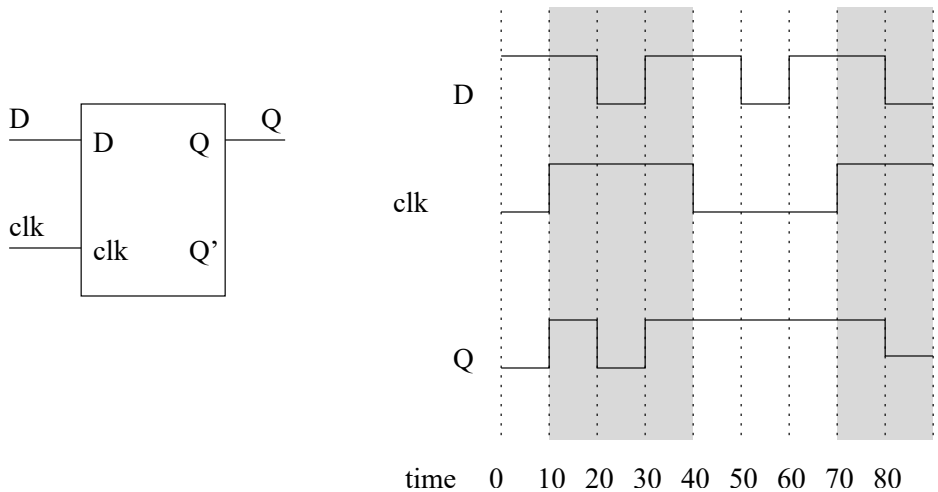


Figure 5.6: A timing diagram showing the behavior of a D clocked latch. The shaded regions describe when the latch samples its data inputs.

time=40 to time=70, the clock input is 0, and the device holds its stored state, so Q does not change regardless of what D is doing.

5.3 The JK Flip Flop

The JK flip flop has two data inputs, an edge sensitive clock input and an output Q, representing the stored bit. The JK flip flop samples its two data inputs on the rising edge of the clock and transforms the stored bit according to the JK's state table. At all other times, the stored bit remains unchanged, hence Q remains unchanged.

The behavior of the JK flip flop can be better understood by examining how it responds to inputs through time in the timing diagram of Figure 5.7. The timing diagram of Figure 5.7 is used to explain its behavior.

Since a flip flop only samples its inputs during the rising edge of the clock input, its J and K values are examined at times 10, 30, 50, 70. At all other times the stored bit, and hence Q, does not change. The initial value of Q is 0. At time=10, the clock rises, and so the inputs are sampled, $J = 1$ and $K = 0$. According to the state table, this input combination causes the JK flip flop to sets its stored bit to 1, causing Q to go to 1 at time=10. At the next rising edge of the clock at time=30, $J = 1$ and $K = 1$, and so the JK flip flop toggles its stored bit value. Since the stored bit was 1 before the clock edge, the stored bit becomes 0 and Q goes to 0. The reader is invited to verify the remainder of the timing diagram.

5.4 The Negative Edge Triggered D Flip Flop

Often, the polarity of the clock input to a basic memory element is reversed. For a clocked latch, this means the data input is sampled while the clock equals 0. For a flip flop, this means the data input is sampled on the negative edge of the clock, when the clock transitions from a 1 to a 0. The term "negative edge" is used because when drawn on a timing diagram, the slope of the clock signal is negative. The transition from logic 1 to logic 0 is also referred to as a falling edge for similar reasons. One indicates the polarity of the clock is inverted by

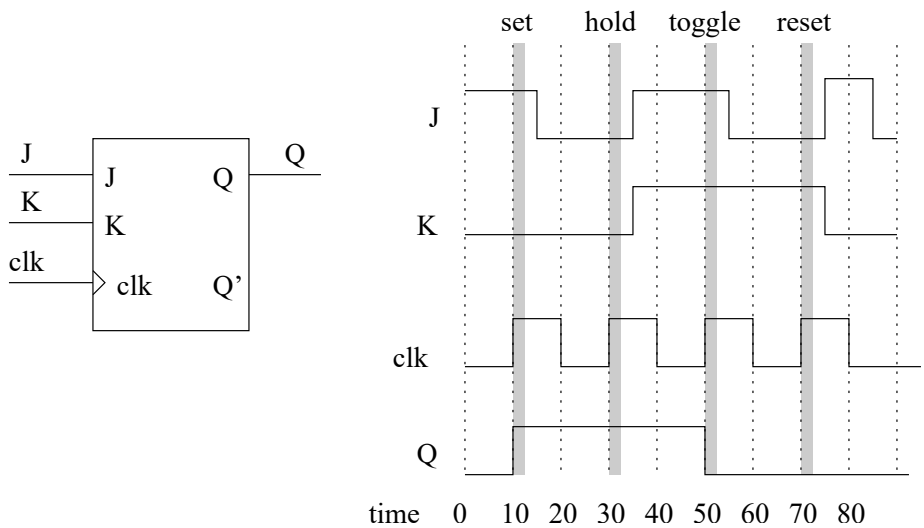


Figure 5.7: A timing diagram showing the behavior of a JK flip flop.

placing a "bubble" in front of the clock input on the schematic representation as shown in Figure 5.8. The bubble is reminiscent of the bubble on the output side of an inverter and represents inversion of the clock input. Beside the fact that a negative edge-triggered D flip flop samples its data input on the falling edge of the clock, it acts like a D flip flop in all other respects.

To better understand the behavior of the negative edge-triggered D flip flop its behavior is observed when subjected to the inputs shown in Figure 5.8.

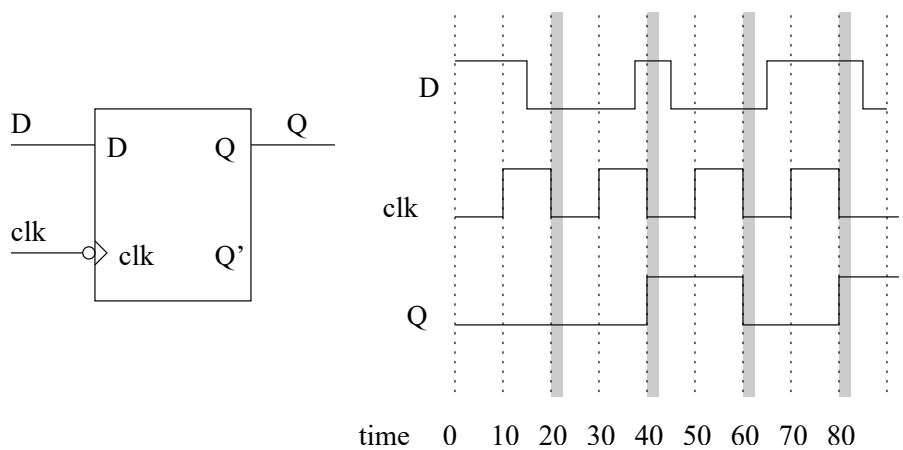


Figure 5.8: A timing diagram showing the behavior of a negative edge triggered D flip flop.

The negative edge-triggered D flip flop samples its inputs on the falling edges of the clock, at times 20, 40, 60, and 80. Only at these times does the stored bit take on the value of the D input. In other words, when the clock transitions from logic 1 to logic 0, the D input is sampled, and then asserted on the Q output. At all other times, the Q output remains unchanged.

5.5 The SR Latch

The SR latch has two data inputs, no clock input and an output Q, representing the stored bit. The SR latch continuously samples its S and R inputs and sets the stored bit when $S = 1$, resets the stored bit when $R = 1$, and holds its stored bit when $S = 0$ and $R = 0$. Since simultaneously setting and resetting the stored bit makes no sense, the input $S = 1$ and $R = 1$ should never be applied to the device. Thus, the stored bit is a "don't care" for this input combination.

The behavior of the SR latch can be better understood by examining how it responds to inputs through time in the timing diagram in Figure 5.9.

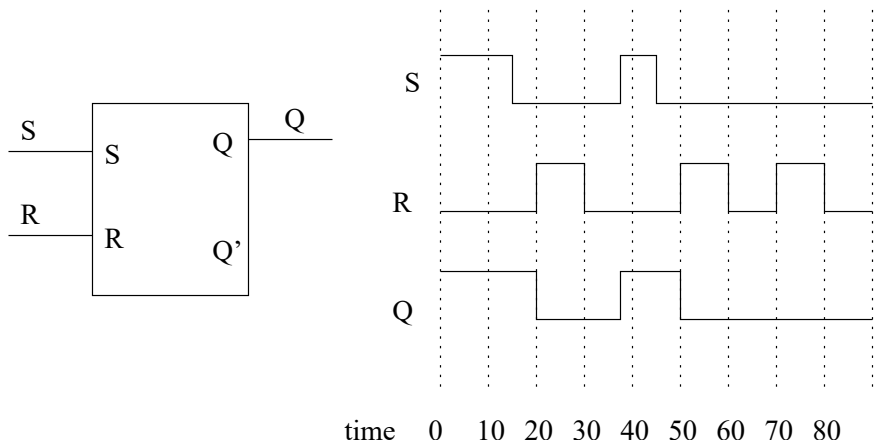


Figure 5.9: A timing diagram showing the behavior of a SR latch.

Initially, at time=0, the SR latch is in a set condition, so its stored bit and consequently its output Q equals 1. At around time=15, both S and R go to 0, so the stored bit holds a 1. From time=20 to time=30, the device is being reset, so Q goes to 0. Between time=30 and time=38, both S and R are 0, so the stored bit holds at 0. The reader can verify the remainder of the timing diagram. Note that at time=18 and time=32, the SR latch has the same input, both S and R are 0. However, the output of the device is different because of the past history of the inputs. In other words, the SR latch remembers what its inputs were in the past.

The SR latch is the only basic memory element whose internal organization will be analyzed. The goal is to understand how the circuit for a SR latch realizes the state table for a SR device.

Consider the circuit realization of an SR latch given in Figure 5.10. Three important observations can be made about this figure. First, the SR latch is a simple circuit, sometimes referred to as cross-coupled NORs for obvious reasons. Second, the circuit contains feedback; outputs are routed back to the inputs. Feedback is an essential feature of any system which has memory. Third, the outputs are labeled Q and Q' even though there is no inverter between the Q and Q' outputs. In most, but not all cases, these outputs will have opposing values.

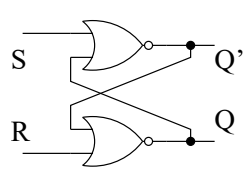


Figure 5.10: The circuit inside a SR latch.

The behavior of the circuit in Figure 5.10 is governed by the behavior of a NOR gate. Thus, it makes sense to start with the truth table for a NOR gate.

A	B	$(A+B)'$
0	0	1
0	1	0
1	0	0
1	1	0

From this truth table, a simple observation should be made: Whenever the input to a NOR gate equals 1, the output equals 0. This will be referred to as the *NOR gate heuristic*.

To understand the behavior of the circuit shown in Figure 5.10, start by examining what happens when $S = 1$ and $R = 0$. According to the NOR gate heuristic, the output associated with the top NOR gate will output 0, hence $Q' = 0$. The Q' output is an input to the bottom NOR gate. Since both of the bottom NOR gates' inputs are 0, it outputs 1. Hence, $Q = 1$. Thus, when $S = 1$ and $R = 0$, the Q output is set to 1 and its negation, Q' is 0.

The analysis of the circuits behavior when $S = 0$ and $R = 1$ is identical except the roles of the top and bottom NOR gates are interchanged. Consequently, the device is reset causing Q to output 0 and its negation, Q' to output 1.

The behavior of the circuit shown in Figure 5.10, is more complicated when $S = 0$ and $R = 0$ because the output depends on what has happened in the past. (See time=18 and time=32 in Figure 5.9.) Furthermore, the NOR gate heuristic does not help, at least initially, because neither input is 1.

In order to make some headway into the analysis, start by assuming the SR latch is being reset ($S = 0$ and $R = 1$), causing $Q = 0$ and $Q' = 1$. Figure 5.11A shows what happens in the instant after the R input is changed to 0. The inputs to the upper NOR gate, labeled TOP, are both 0. Hence, the output of the upper NOR gate is 1, consistent with the existing value of Q'. This result causes one of the inputs to the lower NOR gate, labeled BOT, to equal 1. By the NOR gate heuristic, the output of the lower NOR gate goes to 0, consistent with the existing value of Q. Thus, when the inputs to a SR latch which is being reset are changed to $S = 0$ and $R = 0$, the output remains unchanged. Now, change this assumption and start by setting the SR latch and see what happens when the inputs are changed to $S = 0$ and $R = 0$.

If $S = 1$ and $R = 0$, then $Q = 1$ and $Q' = 0$. Figure 5.11B shows what happens in the instant after $S = 0$. The inputs to the upper NOR gate, labeled TOP, are 1 and 0. By the NOR gate heuristic, $Q' = 0$, consistent with its value. The inputs to the lower NOR gate, labeled BOT, are both 0. Hence, $Q = 0$, consistent with its existing value of Q.

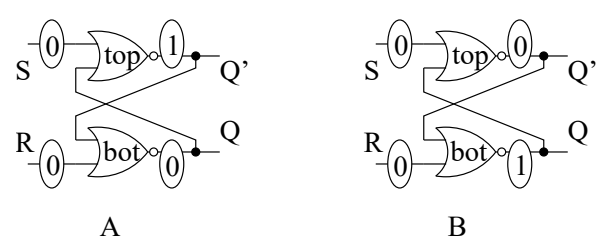


Figure 5.11: The analysis of the SR latch when the S and R inputs are both 0.

In either case when $S = 0$ and $R = 0$, the outputs Q and Q' retain their values. In other words, the SR latch holds its value when both inputs are 0.

5.6 Unimplemented Basic Memory Elements

As mentioned earlier, there are some basic memory elements which are not realized because they exhibit non-deterministic behavior or their implementation is trivial. The sole example of the latter is the D latch. Since it is a latch, it is continuously sampling its input. Since it is a D device, its sampled input is stored and output to Q. Hence, the D latch continuously samples its input and outputs it to Q. The D latch is nothing more than a piece of wire.

A basic memory element which exhibits non-deterministic behavior is the T latch. A T latch continuously samples its input and updates its stored bits. As a T device, it toggles its stored bit when it samples $T = 1$. Unfortunately, when $T = 1$, a T latch will continuously toggle its stored bit. The problem is there is no way to know how fast the T latch is toggling its stored bit and consequently the state of the stored bit after the T input returns to 0 is undetermined. Hence, a T latch exhibits non-deterministic behavior, behavior that cannot be determined/predicted beforehand. The only situation when this behavior is beneficial is for a random number generator.

Any latch or clocked latch which can be forced to continuously toggle its stored bit exhibits non-deterministic behavior. Consequently, T latches, T clock latches, JK latches, and JK clocked latches are not commonly constructed. However, T flop flops and JK flips flops are constructed because they do not continuously sample their inputs, because the clock edge is a singular event. Hence, these two devices sample their inputs once (per clock edge) and toggle at most one time (per clock edge).

5.7 Flip Flop Details

There are two issues that have, up till now, been ignored that must now be addressed. The first involves the possibility of changing the data input of a flip flop at the same time as the clock input. The second involves the initial value of a basic memory element at power-up. The first question is now examined.

What would happen if the D input to a D flip flop were changed at the same time the clock signal changed? Since flip flops are real devices, the answer is that the stored bit will be undetermined because the sample taken could be prior to, or after, the clock edge. In order to avoid the problem, manufacturers of flip flops specify a region of time, before and after the clock edge, during which changes to the data input are prohibited. Changing the data input in the prohibited region may result in the stored bit being undetermined.

Setup time, denoted T_{su}, is the amount of time before the rising edge of the clock when the data inputs must be stable, graphically illustrated in Figure 5.12. Hold time, denoted T_h, is the amount of time after the rising edge of the clock when the data input must be stable. Finally, propagation delay, denoted T_p, is the amount of time after the rising edge of the clock required for the new Q value to become valid.

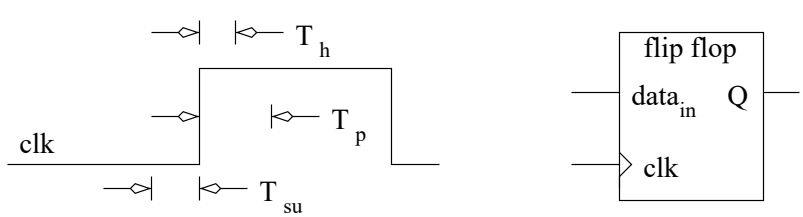


Figure 5.12: The significant time intervals for flip flop inputs and outputs.

Typical values for a D flip flop are shown in the following table. Notice the propagation delay is longer than the hold time. This characteristic makes it possible to cascade flip flops, that is, to run the Q output of one flip flop into the data input of a second flip flop. Since $T_p > T_h$, the hold time of the second flip flop will be completed before the first flip flop changes its value.

Quantity	Time
T_{su}	20nS
T_h	5nS
T_p	15nS

Of the three times associated with a flip flop, only the setup time is relevant to the design of the circuits in this text. The only situation where the setup time is needed is to determine the maximum clocking frequency of a circuit. When doing so, the clocking frequency must be set slow enough to allow all the inputs to the flip flops to stabilize prior to clocking them. This problem will be taken up in a latter chapter. This section will finish by examining the second problem mentioned at the beginning of this section, namely, what is the value of the stored bit at power-up?

When a basic memory element is powered-up, it has no previous history and consequently its stored bit is undefined until one is assigned to the device. To address this problem, manufacturers often equip basic memory elements with asynchronous active low set and asynchronous active low reset inputs, shown at the top and bottom on the D flip flop in Figure 5.13 as S and R respectively.

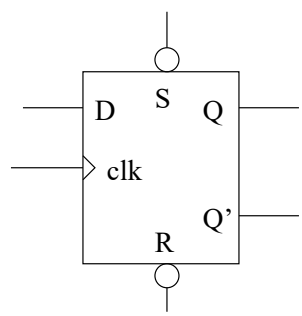


Figure 5.13: A D flip flop with asynchronous active low set and asynchronous active low reset inputs, labeled S and R respectively.

To understand what these inputs do, examine each word in its name in turn. *Asynchronous* means that the S and R inputs function without regard to the clock. *Active low* means that these inputs elicit their behavior when the signal value equals 0. *Set and reset* means that these inputs either set $Q = 1$ or reset $Q = 0$. Thus, when $R = 1$, the output is $Q = 0$ – regardless of what the clock is doing. Consider the S and R inputs to have the highest priority of all the inputs to the flip flop. That is, when R is pulled low, the Q output immediately goes to logic 0, every other input is ignored, and stays there as long as R=0.

Normally, all the asynchronous active low reset lines in a digital system are tied together in one massive reset net. Pulling this line low then resets all the memory elements in the circuit giving them a known state. If some of the memory elements should be initialized to 1, then their asynchronous active low sets are tied together to form a set net. Often, the reset net (and the set net) is connected to a special output from the system power supply. This power supply output holds the system in reset until the power supply has reached its operating output voltage.

5.8 Exercises

1. **(8 pts.)**Determine the state table for the circuit in Figure 5.14.

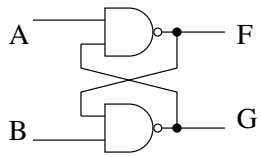


Figure 5.14

2. **(8 pts.)** Determine the state table for the circuit in Figure 5.15. Which basic memory element does it act like? Hint, one of the inputs is acting like a clock. Additional hint, in order to simplify the analysis, replace a portion of the circuit with a component from this chapter.

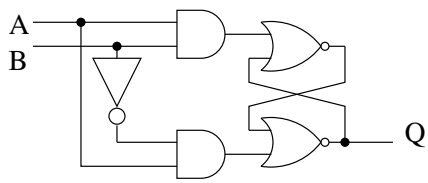


Figure 5.15

3. **(8 pts.)** Complete the timing diagram for the circuit shown in Figure 5.16. Which basic memory element does this circuit act like?

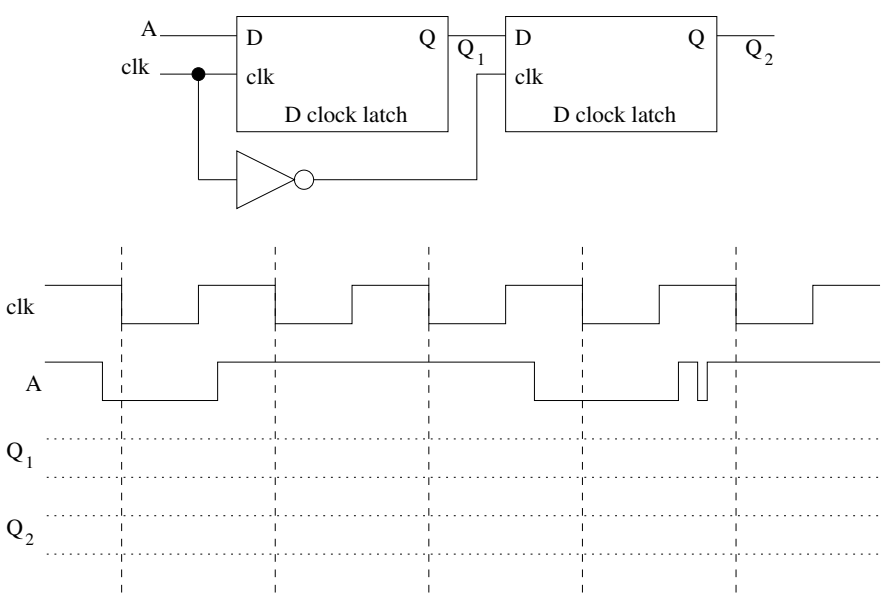


Figure 5.16: A 2-stage sequential circuit.

4. **(15 pts.)** Complete the timing diagram for the basic memory elements in Figure 5.17. The clock cycle is 20 ns. When necessary, assume that Q is initialized to 0 and the output settles to 0 after a period of rapid toggling.

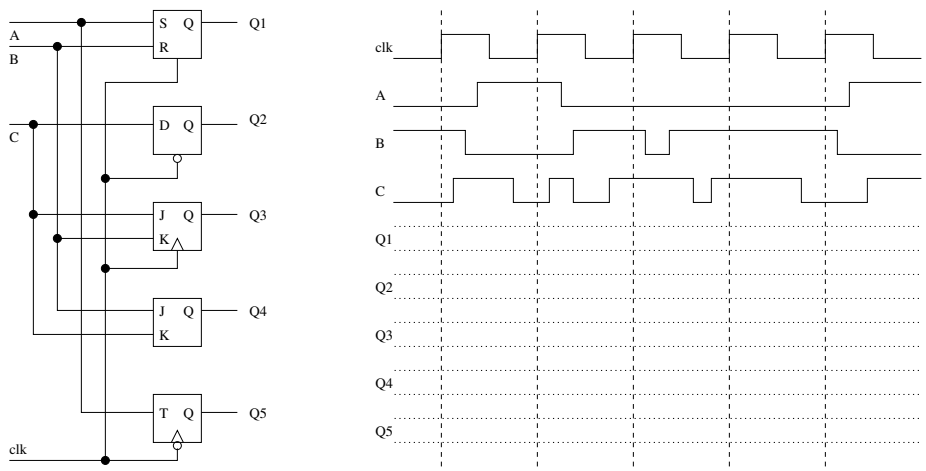


Figure 5.17: A variety of basic memory elements and the signals applied to them.

5. **(15 pts.)** Complete the timing diagram for the basic memory elements in Figure 5.18. The clock cycle is 20 ns. When necessary, assume that Q is initialized to 0 and the output settles to 0 after a period of rapid toggling.

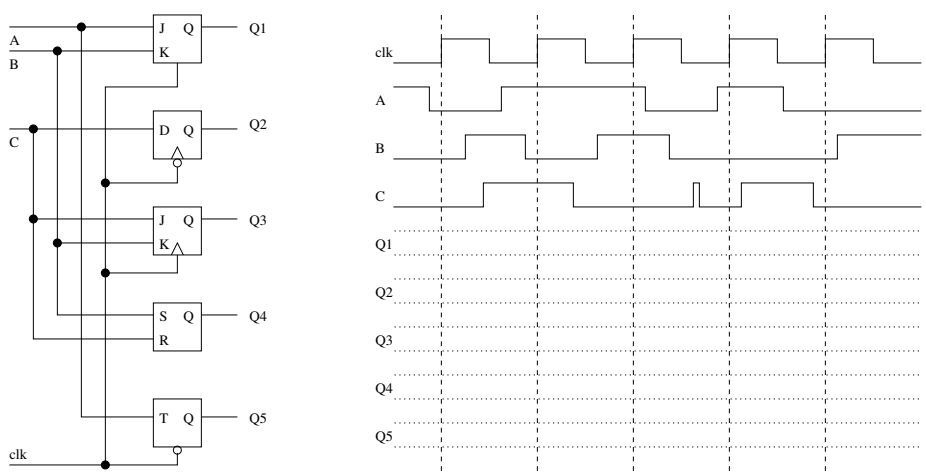


Figure 5.18

6. **(4 pts.)** Consider the furnace controller discussed at the beginning of this chapter. Determine which state the controller should transition into when in a particular state and given a particular combination of inputs. Fill in the eight entries in the following table with the next state the system should move into. The next state should be either ON if the system should transition (or remain in) the ON state, OFF if the system should transition (or remain in) the OFF state, or X if the input combination is meaningless.

Current State\ $T_{hi}T_{low}$	00	01	11	10
ON				
OFF				

7. **(8 pts.)** Derive the next state equations for each type (D, T, SR, and JK) of basic memory element. The next state equation is a symbolic equation describing the next state (Q^+) as a function of the inputs (D,T,SR, or JK) and state (Q). In order to determine the next state equations for a a JK memory element, build a 3-variable Kmap with Q, J, and K as the inputs. The entries in the Kmap should be Q^+. Solving this Kmap will yield the next state equation. Show all work for full credit.

8. **(16 pts.)** Derive the transition list for each type (D, T, SR, and JK) of basic memory element. A transition list describes the input(s) necessary to elicit a particular change in state. For example, imagine that a D flip flop output is currently in the 0 state and it needs to transition to the 1 state after the clock edge. In other words, $Q = 0$ and $Q^+ = 1$. What input would have to be provided on the D input to make this happen? Clearly, $D = 1$. This entry is filled in Table 5.1.

   Hint, the Kmaps used to determine the next state equations will help in visualizing all the conditions which elicit a particular change of input. Complete the transition list for all four memory types. For full credit, show how the entries in the transition list are determined.

$Q \rightarrow Q^+$	D
$0 \rightarrow 0$	
$0 \rightarrow 1$	1
$1 \rightarrow 0$	
$1 \rightarrow 1$	

$Q \rightarrow Q^+$	T
$0 \rightarrow 0$	
$0 \rightarrow 1$	
$1 \rightarrow 0$	
$1 \rightarrow 1$	

$Q \rightarrow Q^+$	S	R
$0 \rightarrow 0$		
$0 \rightarrow 1$		
$1 \rightarrow 0$		
$1 \rightarrow 1$		

$Q \rightarrow Q^+$	J	K
$0 \rightarrow 0$		
$0 \rightarrow 1$		
$1 \rightarrow 0$		
$1 \rightarrow 1$		

Table 5.1: The transition lists for the four types of basic memory elements.

Chapter 6

Sequential Building Blocks

In order to design complex systems, a suitable abstraction is required for building them. This requirement stems from the limitations of the human mind to only manage about a dozen items at once. A complex system containing hundreds of components simply cannot be organized in one pass. Instead, the components are organized into larger units which compose the system. Thus, the number of components need to be considered at a development stage is reduced by an order of complexity. These intermediate units should have a high utility and be modular. In other words, they should be useful and applicable in a wide variety of design situations.

As an example, consider the design problem of writing a technical document describing the operation of a pacemaker for a human heart. This process does not begin by thinking about the spelling of the individual words, instead it makes more sense to first draft an outline. This outline is an abstraction of the written document, it is a simplified representation of the final written document. In somewhat the same way as an author ignores spelling issues when constructing the outline, a designer is not concerned with the operation of AND and OR gates when designing a digital-signal processing chip.

Clearly, choosing components, or as they are called "basic building blocks," which are reusable and have non-overlapping functionality, result in a small number of highly useful components. The set of available building blocks has largely been determined by the electronics industry which provides basic blocks as off-the-shelf prepackaged components. These time-tested components have established themselves over the years as the accepted language of hardware design. Several sequential logic building blocks are examined, next. The word "sequential" in their name implies that these building blocks are different from those presented in Chapter 4 because they have memory.

Like the combinational building blocks shown in Figure 4.1 each of the sequential basic building blocks have control and data inputs, and status and data outputs. In addition, being sequential devices, most also have an edge-sensitive clock input. First to be examined is the most basic sequential basic building block, the register.

6.1 The Register

Building Block: Register

Nomenclature:	N-bit register
Data Input:	N-bits vector $D = d_{N-1} \ldots d_1 d_0$.
Data Output:	N-bit vector $Q = q_{N-1} \ldots q_1 q_0$
Control:	1-bit C
Status:	none
Others:	1-bit edge-sensitive clock. 1-bit asynchronous active low reset.

Behavior:

reset	*clk*	C	D	Q^+	comment
0	x	x	x	0	reset
1	0,1,falling	x	x	Q	hold
1	rising	0	x	Q	hold
1	rising	1	D	D	load

An N-bit register is very much like a wide D flip flop. It samples its N data inputs, denoted D on the rising edge of the clock input. Depending on the control input, C, the register either holds its current value when $C = 0$ or loads the new value when $C = 1$. The stored value of the register is asserted on its output, denoted Q. The columns in the register's state table are organized from left to right, from highest priority to lowest priority. Holding the asynchronous active low reset line to 0 causes the stored value and the outputs to remain at 0 regardless of the value on any other input; the reset input has priority over all other inputs.

A timing diagram for a 4-bit register is shown in Figure 6.1. The initial value of the register is arbitrarily set to $A_{16} = 0001_2$. Since the value of Q is represented using four bits, its value on the timing diagram is shown as a wide trace. This reflects the fact that Q is composed of many bits. At time=10, a positive edge of the clock arrives with $C = 1$, hence the register loads $D = 5_{16} = 0101_2$, as its new value. The fact that the Q outputs changes slightly after time=10 is an acknowledgment that the circuit elements inside the register have propagation delay. The goofy behavior of the C input around time=20 has no effect on the Q outputs of the register because the clock is not rising. The rising clock edge at time=30 does not change the stored value of the register because $C = 0$, hence the register holds its stored value. The change in the Q output at $time50$ results from the rising clock edge and $C = 1$.

An N-bit register is constructed using N, D flip flops. A common error committed by beginning students, and even some text books, is to AND the *clk* and C signals together, sending the AND gate output to the clock input of a D flip flop. This technique is incorrect because it causes the D flip flops to sample their input when the $clk = 1$ and C rises, contrary to the behavior described in the register's state table. As a general rule, avoid modifying the clock signal unless it is absolutely necessary.

The correct construction of an N-bit register is shown in Figure 6.2. Two modes are present for this circuit, corresponding to $C = 0$ and $C = 1$. When $C = 1$, the four multiplexers shown in Figure 6.2, all route the data input D_i to the input of the D flip flop. When a clock edge arrives, each D_i is loaded into its respective flip flop and soon thereafter appears on the Q output.

When $C = 0$, the four multiplexers shown in Figure 6.2, all route their data output Q_i back to the input of the D flip flop. When a clock edge arrives, each Q_i is loaded into its respective

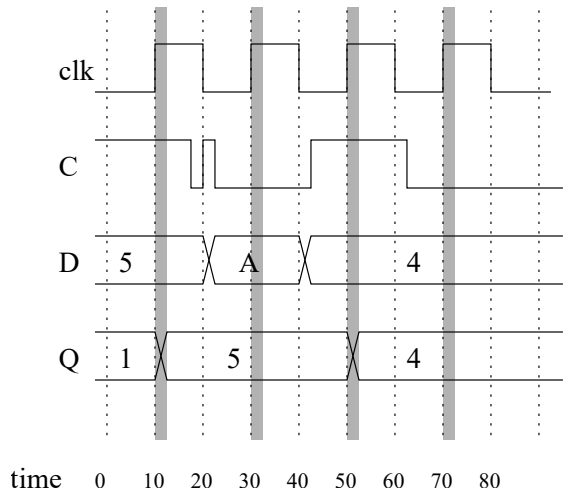


Figure 6.1: A timing diagram for a 4-bit register.

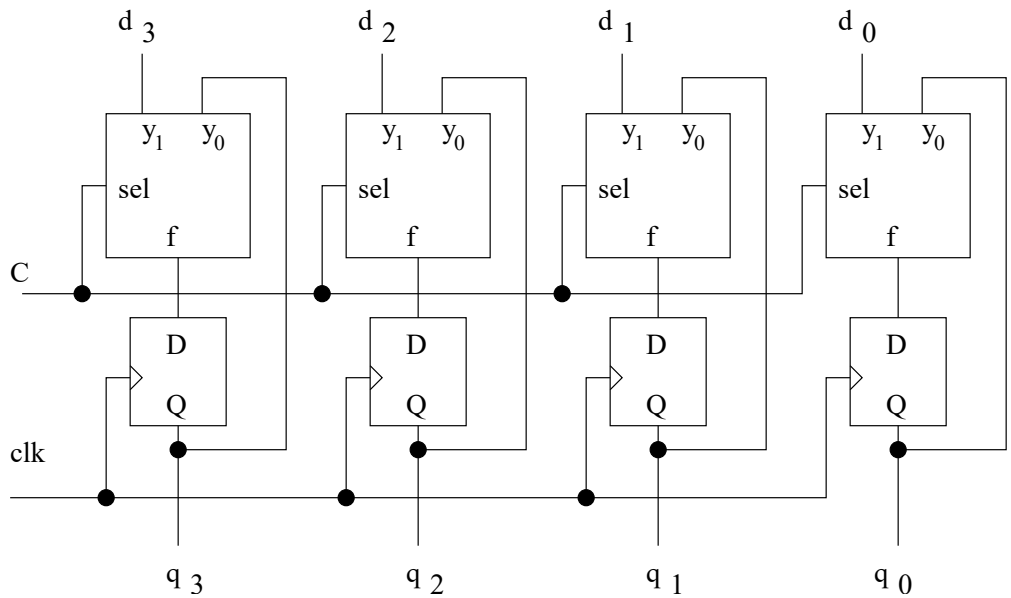


Figure 6.2: The internal organization of a 4-bit register.

flip flop and soon thereafter appears on the Q output. Thus, the Q outputs appear to have held their output value even though the internal D flip flops have loaded a value.

6.2 The Shift Register

A shift register is a register with the additional capability of shifting its stored bits to the left or to the right. The input, output, and behavior of a shift register are shown in the following table.

Building Block: Shift Register

Nomenclature:	N-bit shift register with parallel load
Data Input:	N-bits vector $D = d_{N-1} \ldots d_1 d_0$.
Data Output:	N-bit vector $Q = q_{N-1} \ldots q_1 q_0$
Control:	2-bits $C = c_1 c_0$
Status:	none
Others:	1-bit edge-sensitive clock. 1-bit asynchronous active low reset.

Behavior:

reset	*clk*	*C*	*D*	Q^+	comment
0	x	xx	x	0	reset
1	0,1,falling	xx	x	Q	hold
1	rising	00	x	Q	hold
1	rising	01	x	$Q >> 1$	shift right
1	rising	10	x	$Q << 1$	shift left
1	rising	11	x	D	load

If $Q = 0110$ then shifting Q to the left, denoted $Q << 1$, yields 1100. The symbol "<<" denotes a shift left and the "1" describes how many bits to shift. Shifting the original value of Q to the right by one bit, denoted $Q >> 1$, yields 0011. The ">>" symbol denotes a shift right and the "1" describes how many bits.

Shifting is used to examine bits one at a time and in the multiplication and division of binary numbers. The bits could be examined by looking at the LSB of a shift register as it shifted its bits successively to the right. Multiplication and division involve a bit more explanation.

For instance, consider multiplying a 4-bit binary number $X = x_3x_2x_1x_0$ by 2. This task is accomplished by shifting X to the left one bit, yielding $X << 1 = x_3x_2x_1x_0 0$. That is a "0" is place in the LSB. In order to verify this, write down the decimal equivalent of the shifted value of $X << 1$, $x_3 * 2^4 + x_2 * 2^3 + x_1 * 2^2 + x_0 * 2^1$. Now, factor a 2 from each component of the sum, yielding $2 * (x_3 * 2^3 + x_2 * 2^2 + x_1 * 2^1 + x_0 * 2^0)$. But this is $2 * X$.

For each shift left by one bit, each of the exponents in the decimal representation of X increases by 1, adding a factor of 2 to every term of X which can be factored out. Hence, every shift left increases X by a factor of 2. These factors accumulate for each shift, so that shifting X left three bits increases X by a factor of $2^3 = 8$. Shifting can be used to multiply by constants which are not powers of two by rewriting the constant as the sum of powers of two. For example, to multiple a binary number X by 10, rewrite 10 as $8 + 2$ yielding $10X = (8+2)X = 8X + 2X$. So $10 * X$ is computed by adding together X shifted left by three bits to X shift left by one bit.

Shifting left may create a result which cannot fit in the prescribed word-size. For example, if $X = 12_{10} = 1100_2$ is shifted left one bit in a 4-bit shift register, the result $1000_2 = 8_{10}$ does not represent 24_{10} because this value cannot fit into four bits. It is easy to see a shift left results in overflow whenever the MSB equals 1.

Dividing binary numbers by powers of two is accomplished by shifting the bits to the right. Since it is possible that division by two results in a fraction and combined with the fact that binary numbers represent integers, some form of rounding must occur. For example, if $X = 5_{10} = 0101$ is shifted right by one bit, the result is $010_2 = 2_{10}$. The 1 in the LSB of X is lost. Hence, shifting to the right divides a number by 2 and rounds down when there is a fractional result.

In order to accommodate, the diverse set of situations when shifting can be used, three types of shifts are available: arithmetic, logical, and circular. Since each of these can occur to the left or right, then six possible effects are to be considered due to shifting a 4-bit string $x_3x_2x_1x_0$. These are enumerated in the following table.

	Left	Right
Arithmetic	$x_2x_1x_00$	$x_3x_3x_2x_1$
Circular	$x_2x_1x_0x_3$	$x_0x_3x_2x_1$
Logical	$x_2x_1x_00$	$0x_3x_2x_1$

All these shifts are characterized by how they "fill in" the void created by the shift. Logical shifts always fill in the void with a 0 and are used mainly for multiplication and division of binary numbers.

Circular shifts fill in the void with the bit that "falls off" from the other end of the shift. For example, circularly shifting 0101 to the right yields 1010 because the 1 that falls off the LSB is inserted into the void created at the MSB. Circular shifts are useful when each bit of a register needs to be inspected without destroying the registers contents.

Arithmetic shifts are used mainly to manipulate 2's-complement numbers. An arithmetic shift right fills the void with a duplicate of the MSB, maintaining the "sign" of the 4-bit 2's-complement number. This process is the same as the one governing the sign-extending of 2's-complement numbers as discussed on page 8. An arithmetic shift left fills in the void with a 0 because this multiplies both positive and negative quantities by 2.

To better understand its organization the design of a 4-bit circular shift register that holds its value, circularly shifts its contents to the right, circularly shifts its contents to the left, or loads an external 4-bit input is examined. Since this circular shift register has four functions, it requires two bits of control, denoted c_1c_0. The assignment of bit values to the various functions is arbitrary and defined in the table below. The external 4-bit data input will be denoted D. A clock signal, clk, indicates when the circular shift register should perform its function. Finally, the 4-bit output from the circular shift register is denoted Q.

clk	C	D	Q^+	comments
0,1,falling	x	x	Q	
rising	00	x	Q	hold
rising	01	x	$Q(0)\|Q >> 1$	CSR
rising	10	x	$Q << 1\|Q(3)$	CSL
rising	11	D	D	load

The notation in Q^+ column of the state table needs some further explanation. The $Q(0)$ symbol refers to the LSB of Q, the | symbol denotes concatenation, the merging of the bits to the left and right of the | symbol. Thus, the expression $Q(0)|Q >> 1$ means that the LSB of Q should be "glued" to the most significant three bits of Q.

The circuit for the circular shift register is shown in Figure 6.3 and consists of two major components, D flip flops and 4:1 muxes.

The organization of the shift register is similar to the register, the difference being the larger mux. When the 2-bit control signal, denoted by the slash with a 2 through it, is 00, Q_i is routed to the input of each D flip flop. The rising edge of the clock causes each D flip flop to latch its previous output, causing no change in the outputs. When $C = 01$, the data input to each D flip flop is Q, circularly shifted to the right. This movement is denoted by writing the name of each Q bit on the mux input instead of drawing the lines because otherwise the diagram quickly becomes messy and confusing. Hence, when on the rising edge of the clock, the D flip flops latch the shifted value of the outputs, making all the outputs appear to circularly shift to the right, one bit. The $C = 10$ input cause the inputs to circularly shift to the left. The

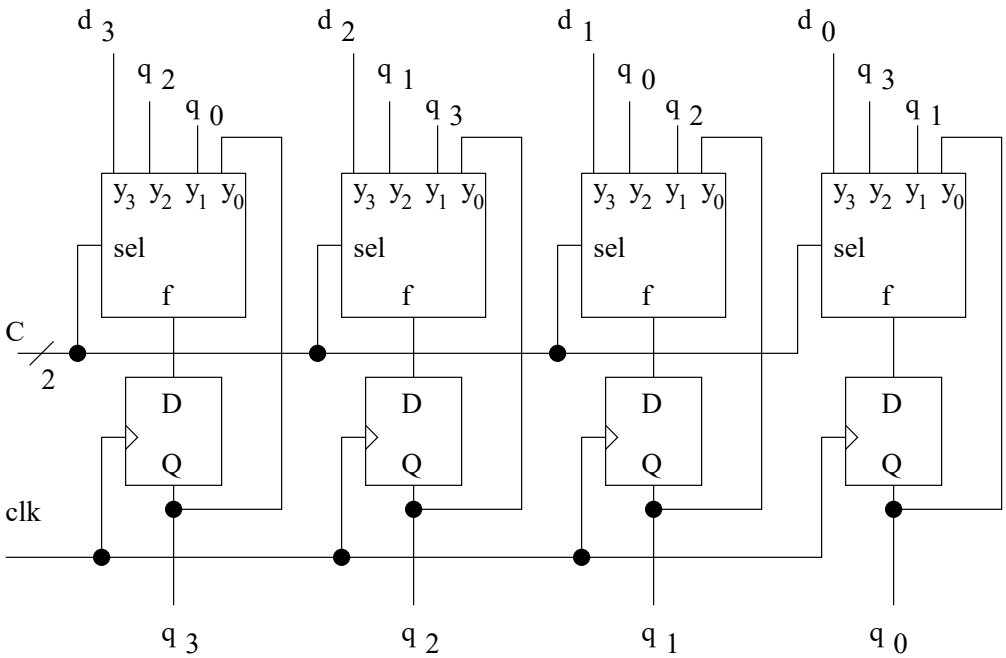


Figure 6.3: The internal organization of a 4-bit circular shift register.

$C = 11$ input, sometimes called a parallel load because all four bits are loaded simultaneously, loads each bit of the external 4-bit input, D, to its respective flip flop.

6.3 The Counter

A counter is a simple but surprisingly versatile piece of hardware. Its behavior is obvious from its name; it counts up when instructed to do so.

Building Block: Counter

Nomenclature:	N-bit counter with parallel load
Data Input:	N-bits vector $D = d_{N-1} \ldots d_1 d_0$.
Data Output:	N-bit vector $Q = q_{N-1} \ldots q_1 q_0$
Control:	2-bits $C = c_1 c_0$
Status:	none
Others:	1-bit edge-sensitive clock. 1-bit asynchronous active low reset.

Behavior:

reset	*clk*	*C*	*D*	Q^+	comment
0	x	xx	x	0	reset
1	0,1,falling	xx	x	Q	hold
1	rising	00	x	Q	hold
1	rising	01	x	D	load
1	rising	10	D	$Q+1$	count up
1	rising	11	x	x	

When the 2-bit control input equals 10 and a clock edge arrives, the counter counts up. Since number of bits are limited, the counting will at some point overflow. When this happens, the count value rolls-over back to 0 and begins counting up again. For example, a 4-bit counter rolls-over to 0 when it tries to count up at 15. This behavior is similar to the behavior of the digits of a car's odometer rolling over from 9 to 0.

A timing diagram for a 4-bit counter is shown in Figure 6.4. The initial value of the counter was arbitrarily set to $E_{16} = 1110_2$. At time=10, a positive edge of the clock arrives. Since the C input is equal to 10_2 at time=10, then the counter counts up to $F_{16} = 1111_2$. The goofy behavior of the C input between time=10 and time=20 has no effect on the Q outputs of the counter because the clock is not rising. At time=30, the C input is equal to 00_2 so the counter holds the current count value. At time=50, the C input is equal 10_2 so the counter counts up rolling over to 0_{16}. At time=70 the counter counts up to $1_{16} = 0001_2$. At time=90, the counter loads $7_{16} = 0111_2$. Notice, as in Figure 6.1, the counter timing diagram shows a small propagation delay for the Q output.

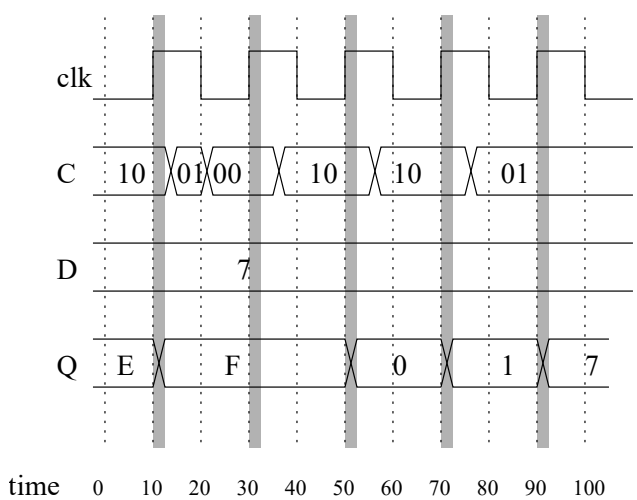


Figure 6.4: A timing diagram for a 4-bit counter.

The internal organization of a counter is very similar to the structure of the register and shift register. A set of D flip flops holds the current count value. A mux decides which input is presented to the D flip flops to be latched up when the positive clock edge arrives. An adder is used to add 1 to the outputs of the flip flops (current count value). If the overflow output of the adder is ignored, then the adder has the advantage of rolling over to 0 when the count value is at the maximum. It is easy to verify that 111...1 + 1 = 000...0. The entire circuit is shown in Figure 6.5.

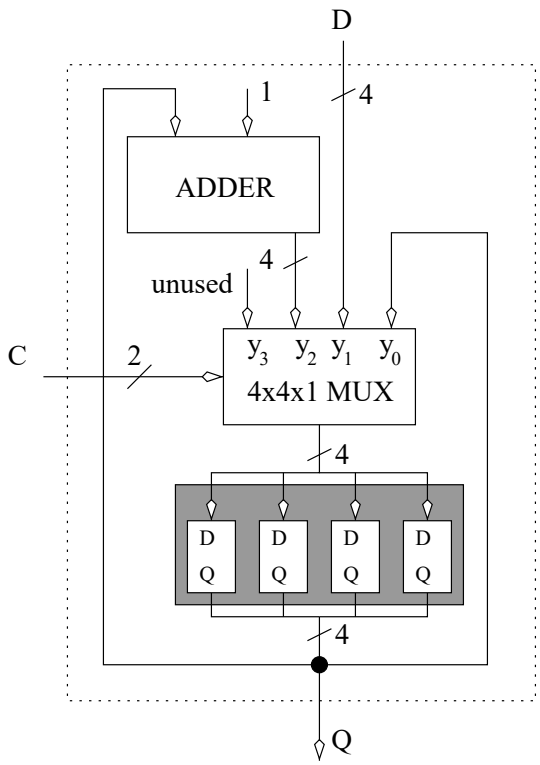


Figure 6.5: The internal organization of a 4-bit counter.

Notice, the data inputs to the mux shown in Figure 6.5 all have a slash through them with the number 4. This notation indicates each of the data inputs is a four bit wide vector, and consequently, this device is a multibit mux, as discussed on page 69. Additionally, the y_3 input is unused. This inefficiency is the cost of using a generalized building block.

6.4 The Static RAM

Random Access Memory (RAM) is an unfortunate name for a digital circuit that has nothing random about it. The name "RAM" originated in the early days of computing when engineers used cassette tapes as an inexpensive way to store "lots" of data. One of the downfalls of a cassette tape is the sequential nature of data access. That is, the time to access a piece of data depends on its position on the tape and the current position of the tape. RAMs do not suffer from the limits of a sequential access devices; the amount of time to access any *random location* is the same.

In order to understand how RAMs work, you will need to understand a few concepts first, so let's start.

A RAM is a device that stores and retrieves bundles of bits, called a word, from an address. You can envision a RAM as tower of binary numbers like that shown in Figure 6.6. The width of a RAM is called the **word size**. In Figure 6.6 the word size is 4-bits. Each set of 4-bits is called a word. In Figure 6.6, the words are organized in rows. Each word in the RAM has an address. By convention addressing starts at location 0. In Figure 6.6, the word at address 5 is 0011. Retrieving information from a RAM is called a **read** operation. Storing information into a RAM is called a **write** operation. The terms read/write should invoke the idea that the RAM is a book or ledger that you are reading with your eyes or writing into it with a pen.

0	0110
1	1010
2	1101
3	0010
4	1000
5	0001
6	1101
7	1111

Figure 6.6: A 8x4 RAM has eight words each containing four bits. The addresses are shown to the left.

While there is no relationship between the word size and the number of words stored in the RAM, there is a relationship between the number of words stored in the RAM and the number of address bits: The number of bits in the address must be sufficient to assign each word a unique binary address. In Figure 6.6, the addresses range from 0 to 7, hence the address is a 3-bit value. In general, if a RAM has 2^N words, it requires a N bit address so that each word is assigned a unique/distinct address.

The number of words in a RAM is often described using the metric system notation.

1k	2^{10}	kilo
1M	2^{20}	mega
1G	2^{30}	giga
1T	2^{40}	tera

The metric names are only close approximations to the actual, binary values they describe. For example, 1 kilometer is a 1,000 meters, but 1k bytes of memory is 2^{10} bytes, or 1024 bytes. The number of address bits can be determined quickly from the metric abbreviations. For example, a 256k RAM has $256k = 2^8 * 2^{10} = 2^{18}$ words, or 18 bits of address.

A wide variety of random access memories are available to meet the wide variety of applications in which users need to store data. For situations where data needs to be retained even though power is removed, non-volatile memories are employed. These memories typically trade-off access and storage speed for the convenience of non-volatility. Volatile memories, memories which lose their contents when power is removed, can be either static/dynamic, or synchronous/asynchronous.

Dynamic memories store data on tiny capacitors and consequently require periodic refreshing in order to retain their values. These versions are typically the highest density memories available, but the refresh circuitry adds to their complexity. Static memories store data in an arrangement of transistors which does not need refreshing. Static memories generally faster, more expensive, and consume less power than their dynamic counterparts.

Access to a synchronous memory requires a clock and is reminiscent of a flip flop. Asynchronous memories require the control and data signals be applied for certain minimum duration in order for the operations to take effect.

In order to convey the behavior and utilization of a RAM we consider a popular type of RAM used in many FPGAs, synchronous static RAM. The input, output, and behavior of a synchronous static RAM is defined by the following table.

Building Block: RAM

Nomenclature:	NxM RAM (random access memory)
Data Input:	M-bit vector $D = d_{M-1} \ldots d_1 d_0$ $log_2(N)$-bit address $A = a_{log_2(N)-1} \ldots a_1 a_0$
Data Output:	M-bit vector $D = d_{M-1} \ldots d_1 d_0$
Control:	1-bit *enb* (read enable), 1-bit *wen* (write enable),
Status:	none
Others:	1-bit edge sensitive clock

Behavior:

clk	enb	wen	A	D_{in}	D^+_{out}	Internal	Note
0, 1, falling	x	x	x	x	D_{out}		
rising	0	0	x	x	D_{out}		hold
rising	1	0	A	x	ram[A]		read
rising	0	1	A	D_{in}	D_{out}	ram[A] $\Leftarrow D_{in}$	write
rising	1	1	A	D_{in}	D_{in}	ram[A] $\Leftarrow D_{in}$	R/W

there are many subtleties burred in this truth table that we will explore in the following example. Before we delve into this example, take a close look at the column headed with D^+_{out} The "+" reference in this column is meant to elicit ideas we explored in Chapter 5, the next state of a basic memory element. In this case, the "+" in D^+_{out} means that this is the value of the D_{out} signal after the clock edge. A digital designer can implement this behavior using an output register to hold the value of Dout. This output register is enabled anytime there is a read operation, enb = 1.

A write operation changes the internal values stored inside the RAM which is not always explicitly obvious. Hence a "Internal" column was included to make these changes clear. During write operations, the the RAM location given by the addr input has its contents modified to the value on Din.

Let's examine how the truth table for a RAM is applied to the timing diagram shown in Figure 6.7.

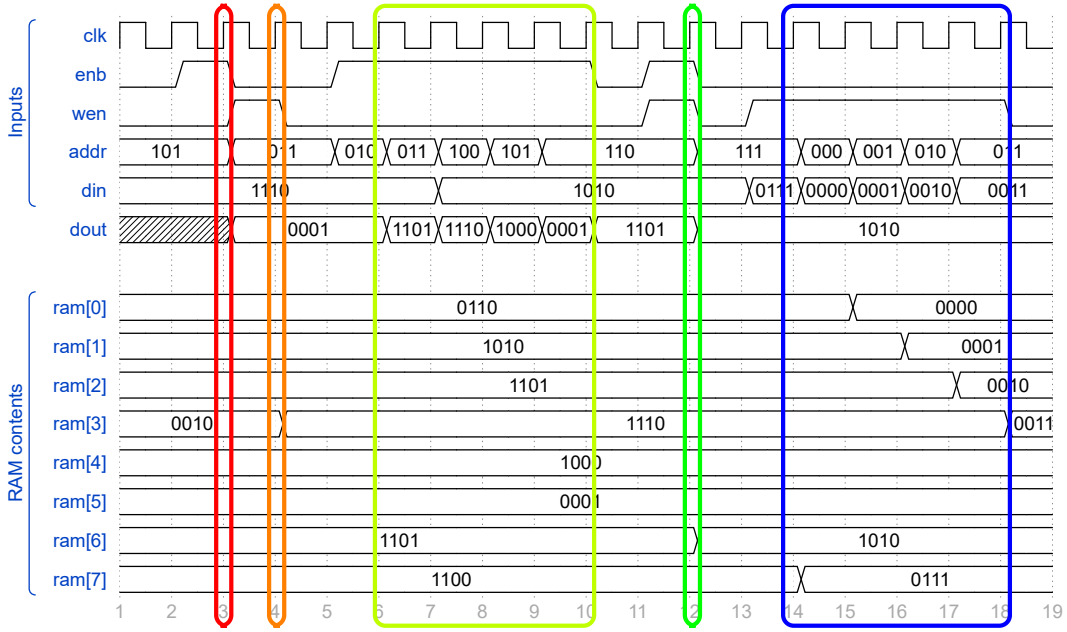


Figure 6.7: The behavior of a 8x4 RAM whose initial values are given in Figure 6.6. Time is in units of nanoseconds (ns).

time = 3ns Just before the rising edge of the clock enb=1, wen=0 so this is a read operation. Since addr = $101_2 = 5_{10}$ on the rising edge of the clock, the value contained in ram[5] (which is 0001) is latched into dout after the rising edge of the clock.

time = 4ns Just before the rising edge of the clock enb=0, wen=1 so this is a write operation. Since addr = $011_2 = 3_{10}$ on the rising edge of the clock, the value of din (which is 1110) is stored in ram[3] after the rising edge of the clock.

time = 6ns to ...10ns During this interval, enb=1, wen=0 on consecutive clock edges so this is a series of read operations. The address is incremented from 2 to 5, so dout get the contents of ram[2] through ram[5].

time = 12ns Just before the rising edge of the clock enb=1, wen=1 so this is a simultaneous read/write operation. Since addr = $110_2 = 6_{10}$ on the rising edge of the clock, the value of din (which is 1010) is stored in ram[6] and latched into dout after the rising edge of the clock.

time = 14ns...18ns During this interval, enb=0, wen=1 on consecutive clock edges so this is a series of write operations. The address is incremented from 0 to 3, so ram[0] through ram[2] have their values changed on consecutive clocked edges.

Cases occur when it is necessary to build a larger RAM from several smaller RAMs. RAMs have two dimensions which can be increased: (1) the word size or (2) the number of words.

Increase the word size of a RAM

Increasing the word size of a RAM is fairly straightforward because each RAM chip gets all the address lines and handles some portion of the data lines. Structurally, this transformation is accomplished by placing several RAM chips side-by-side. For example, in order to construct a 256kx32 RAM from 256kx8 RAM chips, then four, 256kx8 RAM are placed side-by-side as shown in Figure 6.6.

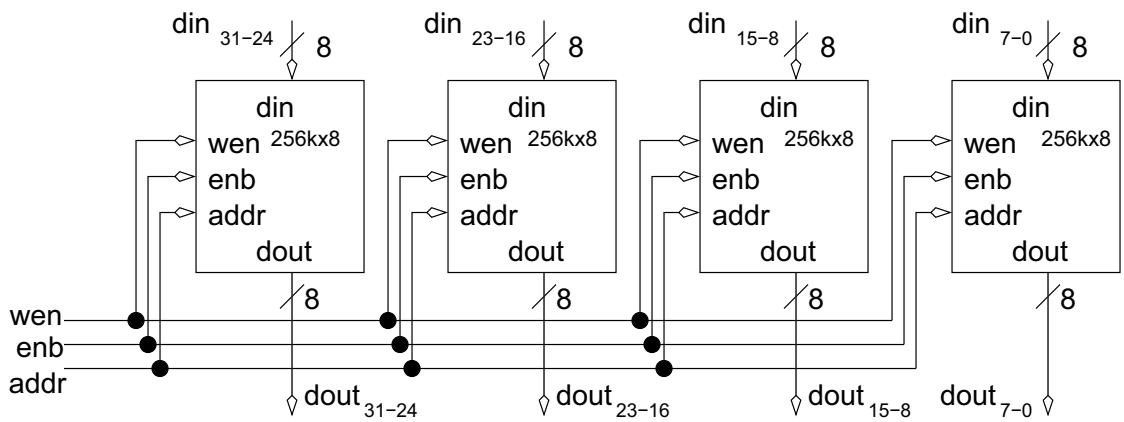


Figure 6.8: The construction of a 256kx32 RAM from 256kx8 RAM chips.

Each of the four chips have the same address lines and control lines *enb*, and *wen*. Thus, all the RAM chips behave in exactly the same fashion, each handling its own set of eight bits. Their actions are coordinated by the common address and control signals.

Increase number of words stored in RAM

Combining RAM chips to increase the number of words involves some manipulation of addresses. For example, consider the problem of using 64kx8 RAM chips to construct a circuit which behaves like a 256kx8 RAM. Stacking four, 64kx8 RAM chips above one another would create the required depth of 256k words. However, each of the 64k RAMs would have 16 address lines, but the 256k RAM being constructed requires 18 address lines. How is this discrepancy of the extra two address bits resolved? The solution depends on whether you are performing a read or write operation.

Figure 6.9 shows how a write operation is performed. The 18 bits of address are split into two components; the lower 16 bits are sent to all four of the 64kx8 RAMs and the upper two bits are used to decide which of the four 64k RAM chips is being written. A decoder uses the upper two address bits as select, and routes the *wen* signal to one of the four 64k RAMs.

The read operation is not shown in Figure 6.9 and is left as an exercise. The solution partitions the address in the same way as the write operation. But instead of a decoder, a multiplexer is used. In order to latch dout, a register also needs to be added to the output.

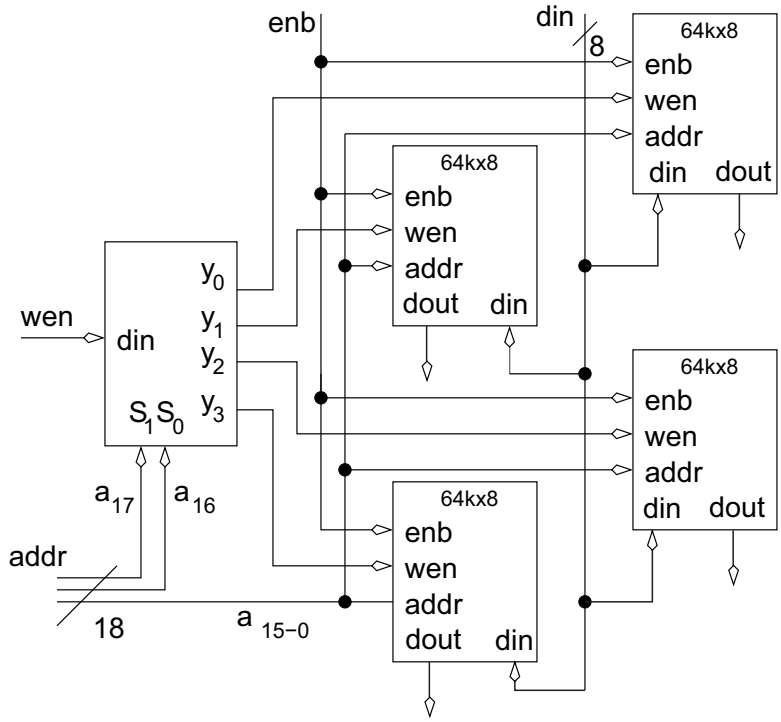


Figure 6.9: An, incomplete 256kx4 RAM from 64kx4 RAM chips.

6.5 Register Transfer

A digital system designed using the datapath and control approach transforms data into some predetermined sequence. For now, focus on the task of transforming the data performed by the datapath. The datapath is composed of the basic building blocks discussed in Chapters 4 and 6; their input, output and behavior is summarized on page 152. Although a gross simplification, a datapath can be considered as some combinational logic "sandwiched" between registers. The clock signal to the datapath governs how data moves in the datapath. After the rising edge of the clock, the register outputs become valid. The output data from the registers flows into the combinational logic which transforms this input into an output. The outputs of the combinational logic flow into the register inputs. The next rising edge of the clock causes the registers to latch these new values. This process then proceeds into the next clock cycle. In order to understand this discussion, consider the simplified datapath shown in Figure 6.10. In this datapath, an adder is sandwiched between three registers. Here, the control inputs on all three registers are assumed to be hardwired to 1, causing them to load on every positive edge. The inputs A and B to the two registers are provided by some external agent.

When a signal has an unknown value, it is given a shaded regions. Prior to time=0, none of the registers contains a known value, therefore all their outputs are shaded. Since $A = 3$

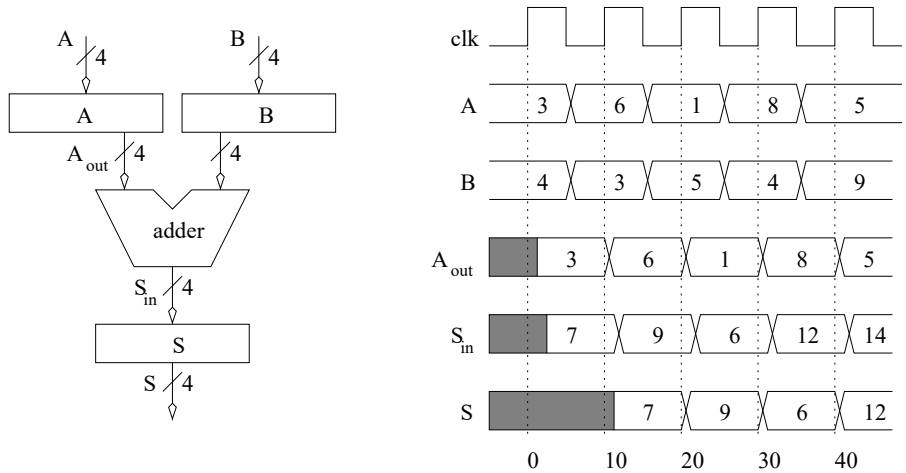


Figure 6.10: A simple datapath and the timing diagram describing its behavior.

and $B = 4$ prior to the positive edge at time=0, the outputs of register A/B equals 3/4 after the positive edge at time=0, respectively. The slight delay in the A_{out} signal becoming valid, emphasizes how the real outputs of a register exhibit propagation delay. Note, the output of the B register is not shown because its behavior is similar to A_{out} and would clutter up the timing diagram. Since the outputs of the A and B registers are unknown prior to time=0, causing the inputs to the adders to be unknown, causing the outputs of the adder to be unknown, causing the input to the S register to be unknown. Since the inputs to the S register are unknown when the clock edge arrives at time=0, the outputs of the S register remain unknown after that clock edge.

It might seem as if the new outputs of registers A/B available just after the clock edge at time=0 would be able to propagate to the S-register's input, allowing it to latch a valid value on the time=0 clock edge. This is incorrect because all the registers in the datapath latch their values at exactly the same instant in time and then output their new values. This assertion is simply a restatement of the observation made on page 96: the propagation delay of a flip flop should be larger than the hold time in order to allow flip flops to be daisy-chained together.

The A and B signals are changed at time=5, on a negative edge of the clock, in order to keep the changes on the register inputs as far away from a positive edge of the clock as possible.

After the rising edge of the clock at time=0, the outputs of the A and B registers become valid, causing the inputs to the adder to become valid, causing the output to become valid, causing the input to the S register to become valid. Thus, the S_{in} signal becomes valid after the positive clock edge at time=0.

When the positive clock edge at time=10 arrives at the circuit in Figure 6.10, registers $A/B/S$ latch the values on their inputs, 6/3/7, respectively. The new outputs of A and B are sent to the adder causing $S_{in} = 9$. This value cannot be latched into the S register until the next rising edge of the clock at time=20, because the rising edge at time=10 is long gone.

The positive clock edge at time=20 causes registers $A/B/S$ to latch 1/5/9, respectively. Once the analysis is started, it is a matter of waiting for a positive clock edge, latch all the register inputs simultaneously, propagating outputs through the combinational logic, and waiting for the next positive edge.

6.6 Combinations

The basic building blocks in Chapters 4 and 6 can be combined in interesting ways to produce complex behavior. The behavior of these digital systems can be described using a programming-language-like syntax, initially presented in Chapter 4. First, examine the counter, counting over a subinterval of its possible range.

```
while(1) {
 if (count<10) count += 1;
 else count = 3;
}
```

The `while(1)` statement means that the statements contained in `while`-brackets should be executed forever. In other words, it is a never-ending loop. The `if` statement checks the count value. If it is less than 10, `count` is incremented, otherwise the value is reset back to 3. The circuit is implemented with a counter to perform the count-up function and a comparator to perform the magnitude comparison between `count` and 9. Why 9? Because if the value is compared against 10, then the `count` value would have to reach 10 in order for the L output of the comparator to change, by which time it would be too late to stop the `count` value from reaching 10. The completed circuit and a timing diagram is shown in Figure 6.11.

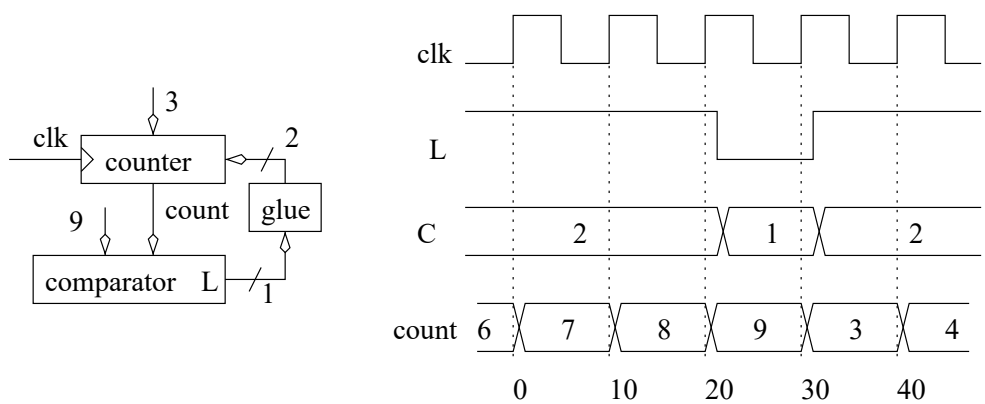


Figure 6.11: A circuit to count between 3 and 9 and its associated timing diagram.

The L output of the comparator is used because the condition checked is `count < 10`. This single-bit output cannot be directly connected to the counter's 2-bit control input because there just are not enough bits. The "glue" box shown in Figure 6.11 contains glue-logic, a combinational logic circuit which interfaces or glues together two pieces of logic. When the counter's output is less than 10, $L = 1$, and the counter is supposed to count up by 1. According to page 107, this requires the control to be $c_1c_0 = 10$.

When the counter's output is greater than or equal to than 10, $L = 0$, the counter should load 3. According to page 107, a load is elicited by asserting $c_1c_0 = 01$ on the counter's control input. The resulting truth table for the glue logic is shown in the margins. From this table, it is easy to ascertain that $c_1 = L$ and $c_0 = L'$.

L	c_1	c_0
0	0	1
1	1	0

The timing diagram shows the counter starting at 6. Since 6 is less than 10, then $L = 1$. When $L = 1$ is present on the input of the glue logic it will output $C = c_1c_0 = 10$, causing the counter to count up when the clock input arrives at time=0. Successive clock edges see the count value increment to 9 at time=20. At this moment, the L output of the comparator

changes to 0 causing the glue logic box to output $C = c_1c_0 = 01$ telling the counter to load 3 on the next positive clock edge at time=30. When the value of 3 is loaded into the counter at time=30, the L output of the comparator changes to 1, causing the glue logic box to tell the counter to count up on the next positive clock edge at time=40. Without a finite state machine, it is difficult to get a combination of basic building blocks to perform a sequence of actions. Difficult, but not impossible as the following circuit shows.

Design a circuit which searches the first 100 memory locations of an eight bit wide random access memory for the smallest value is examined. Assume the RAM is preloaded with data values, so only the memory needs to be read. The approach reads each value and checks if it is smaller than the smallest 8-bit value found thus far. This smallest value is stored in a register, `min`, which will be assumed as initialized to the largest possible value, 0xFF. The "0x" in front of "0xFF" is used in many program languages to signify that "FF" is a hexadecimal number.

```
1.    // assume that min is initialized to 0xFF
2.    for(i=0; i<100; i++) {
3.        MBR = RAM[i];
4.        if (MBR < min) MBR = min;
5.    }
```

The complete solution is shown in Figure 6.12. When working through a solution, you should work your way through the algorithm from beginning to the end adding and connecting hardware as you proceed. Before starting this process a few notes are in order. The variable `MBR`, standing for memory buffer register, a generic term applied to a register used to buffer data operations between a RAM and a datapath. The enb and wen lines on the RAM are connected to logic levels because we only need to read from the RAM. This is called "hardwiring" the inputs. This is done when the input never changes, so simplifying the design. This sort of things occurs often enough in digital design we have a name for it - you may have to hardwire inputs in your homework solutions.

Line 2: is implemented with a counter and a comparator. The comparator examines the counter's output, called i, and stops the counter when the count value reaches 99. The truth table for the glue logic box is given by:

L	counter	MBR register
0	count up	load
1	hold	hold

Note, you do not need to look-up the control values for the counter and register. It is easier to understand if you write the action that the counter and register in the truth table. Only if you wanted to determine the logic gates inside the glue logic block, you would need to look up the control word values to perform these actions.

Line 3: is implemented with a RAM and a register. The address of the RAM `[i]` comes from the counter, and the data output of the RAM is sent to the data input of a register whose output is called MBR.

Line 4: is realized with a comparator and a register. The comparator compares the output of the MBR and the min registers and asserts its L output when MBR is less than min. This L output runs to the control input of the min register. The data input of the min register comes from the data output of the MBR register. The control of the MBR register should stop loading when the count value is greater or equal to 99 - this was completed in the truth table described in Line 2.

The circuit diagram is shown in Figure 6.12. Since the problem statement did not include information about the word sizes, we have ignored them in our design.

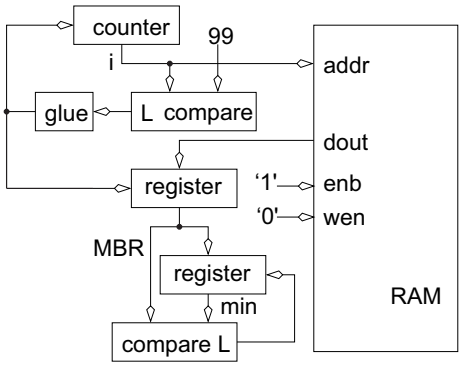


Figure 6.12: A circuit to find the smallest value in a RAM. The min register is initialized to 0xFF.

6.7 Exercises

1. **(8 points)** Build a 4-bit universal shift register in Table 6.1 using D flip-flops and 8:1 multiplexers.

S2	S1	S0	Operation
0	0	0	Hold
0	0	1	Load
0	1	0	ASR
0	1	1	ASL
1	0	0	LSR
1	0	1	LSL
1	1	0	CSR
1	1	1	CSL

Table 6.1: The truth table for a universal shift register.

2. **(8 pts.)** Use a counter and a comparator to implement the following circuits.

   a) Show how to modify the counter (by adding some external logic) to implement a mod-10 counter. A mod-10 counter counts from 0 to 9 and then goes back to 0. It spends one full clock cycle on each of these count values.

   b) Use four mod-10 counters to build a 4-digit decimal counter which counts up from 0 to 9999. Draw a schematic for the 4-digit decimal counter.

3. **(8 pts.)** Design a circuit which contains three 8-bit registers X,Y,Z. The behavior of the circuit is determined by the statement:

```
if (X > Y) then Z = X+X else Z = Y+Y
```

   The registers are preloaded with values in them. Submit a circuit diagram showing the building blocks uses, their interconnections and any miscellaneous logic required to make them operate together.

4. **(8 pts.)** Design a circuit which contains three registers X,Y,Z. The behavior of the circuit is determined by the statements:

```
1. while (X > 0) {
2.     Z = Z+Y;
3.     X = X-1;
4. }
```

   The registers are preloaded with values in them. Submit a circuit diagram showing the building blocks used, their interconnections and any miscellaneous logic required to make them operate together. The design should use an adder and an adder subtractor plus some other building blocks. Hint, use the enable inputs of the registers to control when they latch information.

5. **(8 pts.)** Given three 32-bit registers A,B,PC, design a circuit which adds PC and A (putting the result back into PC) when A is equal to B. Otherwise, add 1 to PC. The contents of A and B are to remain unchanged.

6. **(8 pts.)** Build a circuit that performs the following:

```
for(i=0; i<100; i++)
    total = total + i;
```

Use the counter described in this chapter for the i variable; assume that the counter is initialized to 0. `total` is stored in a register and its initialized to 0. Use a comparator to shut down the counter and put the register in hold when the count value reaches a critical value. Until this critical value is reached the comparator should allow the counter to count and the register to load.

7. **(8 pts.)** Build a circuit that performs the following:

```
for(i=0; i<100; i++)
    total = total + 1;
```

8. **(8 pts.)** Design a circuit that can shift (circular to the right) the contents of register X by an amount given in register Y. X is stored in the circular shift register described in this chapter. The solution will require a comparator and a counter.

9. **(8 pts.)** Assume a 32kx8 RAM is full of data. Show the hardware required to realize the following algorithm.

```
for(i=0; i<32767; i++)
    total = total + M[i];
```

Where `M[i]` is the 8-bit word stored at address `i`. Assume the total register is initialized to 0. The i variable should be the output of a counter. Use a comparator to shut down the counter and to put the register in hold when the count value reaches a critical value.

10. **(8 pts.)** Show how to initialize a 32kx8 RAM in the following manner.

```
for(i=0; i<32767; i++)
    M[i] = i mod 256;
```

Where the "i mod 256" statement means store the least significant eight bits of the i variable into the RAM.

11. **(5 pts.)** Complete the timing diagram in Figure 6.13 for a 4-bit arithmetic shift register. Use the control setting from the truth table on page 104.

12. **(5 pts.)** Complete the timing diagram in Figure 6.14 for a 4-bit counter. Use the control setting from the truth table on page 107.

13. **(5 pts.)** Complete the timing waveforms for A_1, A_0, Q_1, Q_0 based on the circuit diagram shown in Figure 6.15. Use the truth table on page 102 for the register. Put the decimal representation of the signals in the timing diagram (like the timing diagram in Figure 6.1 of the textbook).

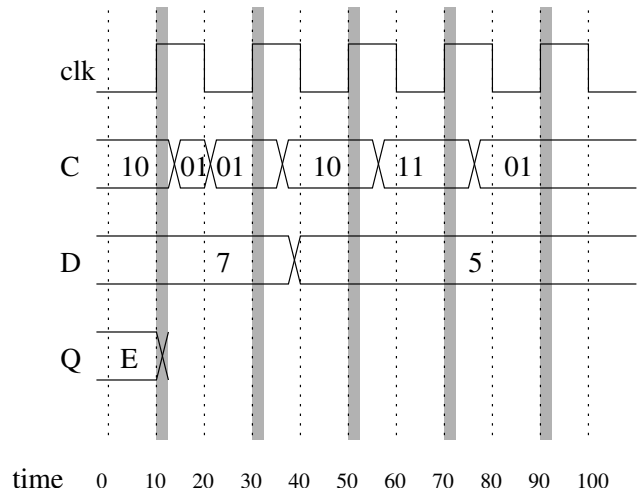


Figure 6.13: The timing diagram for a 4-bit arithmetic shift register in Problem 11.

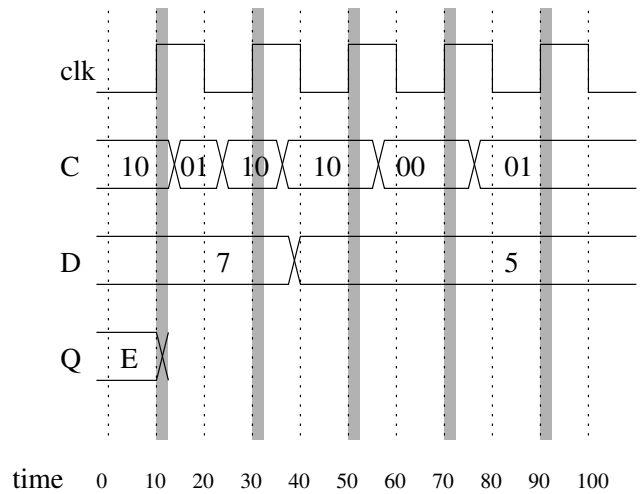


Figure 6.14: The timing diagram for a 4-bit counter in Problem 12.

14. **(4 pts.)** The circuit shown in Figure 6.16 generates a Fibonacci sequence, a sequence starting with 1,1,2,3... The next number in the sequence is the sum of the preceding two numbers. Complete the timing diagram, assuming the circuit starts with the values shown. Identify the signal which generates a complete Fibonacci sequence.

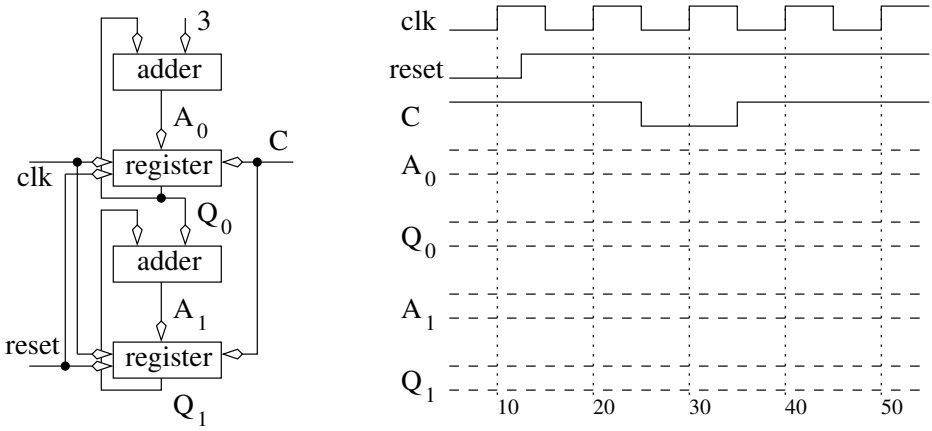


Figure 6.15: The circuit diagram and incomplete timing diagram for Problem 13.

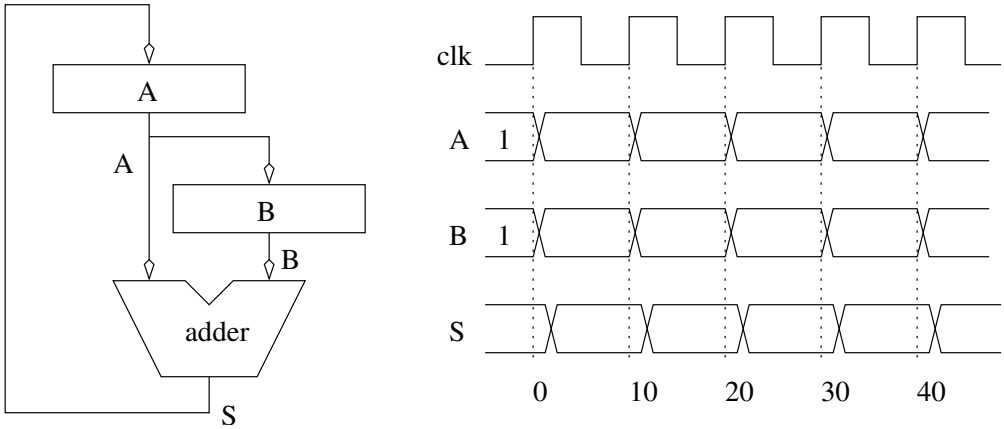


Figure 6.16: A circuit which generates a Fibonacci sequence.

Chapter 7

Finite State Machines

In Chapter 5 the exploration began with sequential circuits, circuits whose output is a function of the input and current state, and by examining the basic memory elements. Now, utilization of these basic memory elements to build general sequential circuit is considered.

The sequential design process starts the same as the combination design process, with a word statement. From this word statement, a state diagram is created. The logic to control the transitions between the states is derived directly from the state diagram. In cases where the finite state machine (FSM) controls a complex system, it may be necessary to build a control word table to determine the output equations. Otherwise, the output equations are derived directly from the state diagram. In order to make this discussion more concrete, all the FSMs to be designed in this chapter have the structure shown in Figure 7.1.

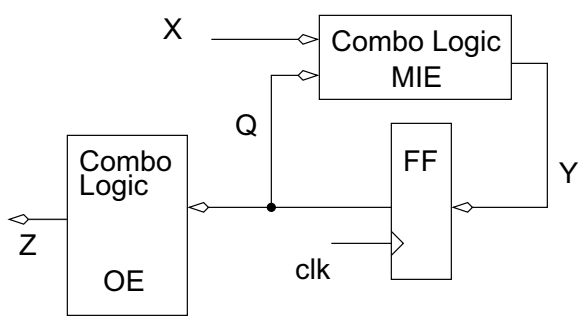


Figure 7.1: The standard structure of a FSM for the current chapter.

Each of the signals X, Y, Q, Z is a vector, consisting of zero or more bits. The X signal is the input to the FSM from the system being controlled. The Z signal is the output from the FSM to the system being controlled. The combinational logic circuit generating Z is called the *output equations* (OE). The state of the FSM is carried on the Q lines. Each bit of Q is the output of a D flip flop. Thus, if Q is six bits wide, then the FSM has six D flip flops. The Y signals are called the memory inputs; they are the data inputs to the D flip flops. The combinational logic circuit generating the Y signals is called the *memory input equations*

(MIEs). In order to improve the readability of circuit diagrams, from now on, the clock signal will not be shown. With respect to Figure 7.1, the design of a FSM requires three questions be answered: What are the MIEs; what are the OEs; and how many D flip flops are required? These questions are answered by first understanding how to convert a word statement into a state diagram.

7.1 Word Statement to State Diagram

Since the design of a FSM starts with the conversion of a word statement into a state diagram, it is imperative to understand what information is captured in the states of a state diagram.

The states of a FSM can be thought of as the history of previous inputs, the operating modes of a system, or the steps in a process.

A state can be used to capture the history of previous inputs. For example, consider a FSM controlling a vending machine which dispenses 35¢sodas. The FSM has two bits of inputs, N and D which equal 1 when a nickel or dime is inserted. It has one bit of output which equals when to dispense a soda. The states in this FSM would represent how much money has been deposited so far.

A state can be used to capture the operating mode of a system. For example, the FSM which controls a furnace in a house might have two states; on or off.

A state can be used to capture which step a process is in. For example, consider a FSM which controls the movement of milking cows through a chute in order to read their ID tags. The FSM has two inputs, one which detects if a cow is present in the chute and the other if the ID tag has been read correctly. The FSM has two outputs, each controlling the position of either the entrance or exit gates. The FSM has states such as CowPresent, WaitToConfirmRead, WaitForCowToLeave, etc..

Regardless of what the states represent, the state diagram must be drawn in the same standardized way. Each state name is drawn inside a circle. One of the states is selected to be the state the FSM starts in when first powered-up. This state, referred to as the reset state, is denoted by putting an asterisk inside its circle. The FSM is in exactly one state at a given time. When the clock edge arrives, the FSM samples its inputs and transitions to a next state. Outgoing arcs from the state are labeled with the input condition which elicits the corresponding transition. As mentioned in Chapter 5, the collection of outgoing arcs from a state should be complete and unequivocal. Complete means that every possible combination of the variables used on the outgoing arcs has been described. Unequivocal means no ambiguity in the decision of the next state exists.

7.2 Design Using Ones Hot Encoding

In order to transform an abstract state diagram into a real circuit, the symbolic state names in the state diagram must be given some binary encoding. When the FSM is in a particular state, S, the binary coding for S is present on the outputs of the flip flops; Q in Figure 7.1. Many different ways can be proposed to choose the encoding of states. The two most popular are dense and one-hot.

A dense encoding minimizes the number of bits used to encode the states. This approach is a reasonable goal as minimizing the number of bits to encode the states has the effect of minimizing the number of flip flops in the FSM. According to the arguments made on page 3, if a state diagram has N states, then each can be assigned a unique binary code using $log_2(N)$ bits. For example, if the vending machine state diagram mentioned earlier had eight states,

then its FSM designed using a dense encoding of states requires three bits. These three bits could be arranged in eight different ways, each representing one of the states. Consequently, its FSM would have three flip flops.

A one-hot encoding assigns one flip flop to each state. When the FSM is in state S, the flip flop associated with state S outputs 1. Since a FSM can only be in one state at a time, exactly one flip flop in a one-hot encoded FSM outputs 1 and all the others output 0. The name "one-hot" refers to the fact that one of the flip flop outputs is "hot" or at a logic 1.

Each encoding technique has its own strength and weakness which should influence the choice of which to use. The strength of the dense encoding is it minimizes the number of flip flops in the design. The biggest draw-back of using a dense encoding is it requires a significant effort to minimize the circuits in the MIEs and OEs. The strength of the one-hot encoding is the fact that the process of determining the MIEs and OEs is quick and simple. Furthermore, when compared to a dense encoding, the MIEs and OEs of a one-hot encoded FSM generally require far less logic. The weakness of the one-hot encoding is it requires far more flip flops than the dense encoding.

The choice of which encoding boils down to deciding the medium to use to implement the digital circuit. Field programmable gate arrays (FPGAs) contain a large number of standard cells which can be configured and interconnected in a variety of ways. A one-hot encoding makes sense for FPGAs because the automated design tools would have a difficult time optimizing a dense encoding and there is an abundance of flip flops available. In a custom-designed VLSI chip, it would make sense to utilize a dense encoding in order to minimize the number of gates in the implementation, and hence, the size of the resulting circuit. Given the prevalence of FPGAs, one-hot encoding is the focus for the states of the chapter's FSMs.

Three steps transform a state diagram into a circuit diagram when using a one-hot encoding of the states. First, determine the number of flip flops. Second, determine the MIEs. Third, determine the OEs.

The first step is easy; assign one flip flop to each state. For each state S, label the input of the flip flop D_S and the output of the flip flop Q_S.

The second step requires the derivation of the MIEs. Since each state gets its own flip flop and there is one MIE for each flip flop, then there is one MIE for each state. The MIE for state S should output 1 when the FSM transitions to state S in the next clock cycle. In other words, the MIE for state S is the answer to the question, "How does the FSM get into state S?" State S is entered by starting at some state P and meeting the condition c on its arc leading to state S. The transition arc contributes a term $P * c$ to the MIE for state S. If more than one arc terminates at state S, then the MIE for S is the logical OR of the terms from each arc.

For example, in the state diagram shown in Figure 7.2, three arcs terminate at state S. Consequently, the MIE for state S consists of three terms, one from each of the arcs. The arc from state A contributes the term $Q_A x' y$, the arc from state B contributes $Q_B x'$, and the arc from state C contributes $Q_C(x+y')$. Putting all these terms together yields the memory input equation $D_S = Q_A x' y + Q_B x' + Q_C(x + y')$.

The third step in the transformation of a state diagram into a circuit diagram requires the elaboration of the output bits and the definition of their value for each state forming an output table. From Figure 7.1, see that the OEs are dependent only on the state. Remember the output from the flip flop associated with S, Q_S, equals 1 when the FSM is in state S. Consequently, if an output Z equals 1 when the FSM is in state S, then the output equation for Z will contain the term Q_S. Consulting the output table for a FSM reveals a list of states for which an output Z equals 1. Since the output equals 1 when the FSM is in any of these states, the output equation for Z is the logical OR of these states.

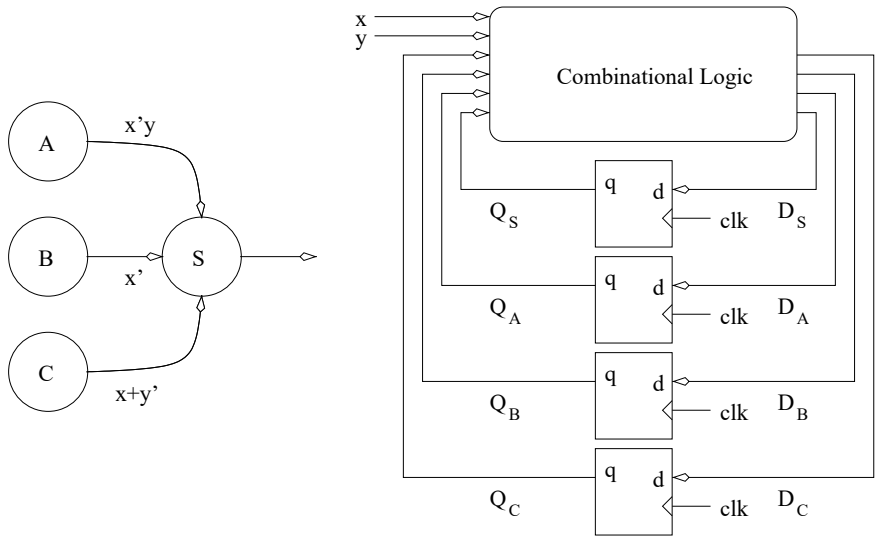


Figure 7.2: A state diagram and its FSM.

The furnace controller circuit first introduced in Chapter 5 is revisited to better understand the concepts illustrated in Figure 7.2.

Word Statement Design a FSM which controls a furnace in order to regulate the temperature in a house. The FSM has two bits of input from thermometer, T_{hi} and T_{low}. When the temperature inside the house is greater than the high threshold, then T_{hi} outputs 1; otherwise it outputs 0. When the temperature inside the house is greater than the low threshold, then T_{low} outputs 1; otherwise it outputs 0. The output from the FSM controls a furnace. When the FSM outputs a 1, the furnace turns on, warming the house. When the FSM outputs a 0, the furnace is turned off, causing the house to cool down. A graph of this controller's behavior is given in Figure 5.2.

State Diagram The FSM has two bits of input and one bit of output. The state diagram, shown in Figure 7.3, has an asterisk in the state **OFF** implying that this state is the reset state. Hence, for start-up, the furnace controller is set in the **OFF** state.

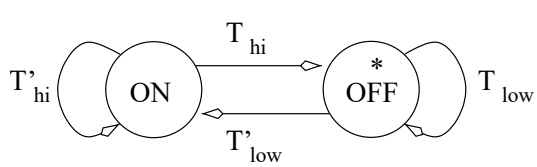


Figure 7.3: A state diagram describing the furnace controller.

Circuit Diagram The circuit for the FSM shown in Figure 7.3 requires two flip flops because it has two states. These flip flops can be called ON and OFF, their inputs D_{on} and D_{off}, and their outputs Q_{on} and Q_{off}.

The MIEs are derived directly from the state diagram. Two arcs terminate at the **ON** state, thus the MIE for this state has two terms, $D_{on} = Q_{off} * T'_{low} + Q_{on} * T'_{hi}$. Likewise, the MIE for the **OFF** state has two terms, $D_{off} = Q_{on} * T_{hi} + Q_{off} * T_{low}$.

In order to determine the OEs, an output table is constructed. The table is organized by listing each output as a column and each state as a row. In the case of the furnace controller, this construction is quite simple.

State	Furnace
	0 off
	1 on
ON	1
OFF	0

Since the single output equals 1 when the FSM is in the **ON** state, $Z_{furnace} = Q_{on}$. Putting all this together yields the complete circuit diagram for the furnace controller FSM as shown in Figure 7.4.

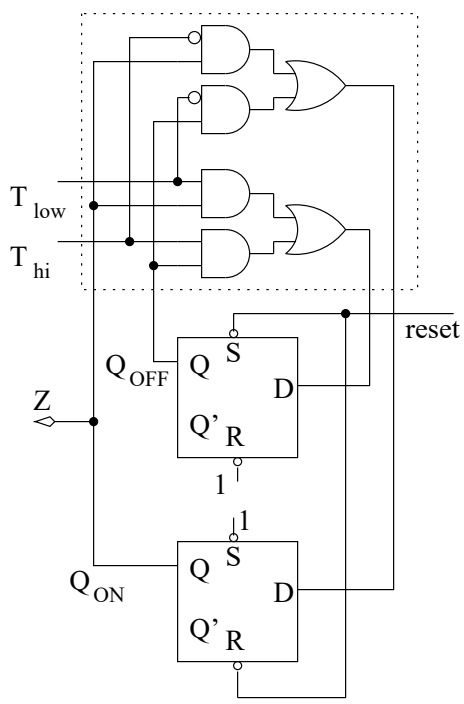


Figure 7.4: The circuit diagram for the furnace controller.

One final note needs to be made about the circuit diagram in Figure 7.4; its reset configuration. According to the state diagram shown in Figure 7.3, the **OFF** state is the reset state. Hence, the reset signal is wired to the asynchronous active low set input of the OFF D flip flop so that its output goes to logic 1 when the reset is activated. All the other flip flops are connected to the reset signal through their asynchronous active low reset input so that their outputs all go to logic 0 on a reset event.

Since the furnace controller operates through time on real inputs its behavior is better understood in light of applying inputs and examining the outputs in a timing diagram.

7.3 Timing

Once the FSM is realized, the only way to inspect the behavior of a physical realization is by applying inputs and observing outputs. A timing diagram is used to examine a sequence of inputs and the resulting behavior of the FSM. The timing diagram is compared to the state diagram of the FSM in order to verify that it is operating as originally designed. Before diving into an example timing diagram, the sequence of events occurring inside the FSM needs to be understood.

The events occurring in the FSM are referenced to the clock input of the D flip flops inside the FSM; see Figure 7.5. Since flip flops sample their inputs on the positive edge of the clock, this point is the beginning of the timing analysis, denoted as Event 1 in Figure 7.5. The propagation delay of the flip flops means a small delay occurs between the clock edge and the flip flop outputs, Q, becoming valid, Event 2 in Figure 7.5. The propagation delay of the flip flops is denoted T_{FF}. In order to maximize the clocking frequency of the FSM, the new inputs, X, to the FSM should be applied at the same moment that the flip flop outputs are becoming valid. This event is Event 3 in Figure 7.5. According to Figure 7.1, changing Q and X has two effects, the outputs Z change and the memory inputs Y change. The delay between the application of the new inputs to the MIE logic and Y is the propagation delay of the combination logic, denoted T_{combo} in Figure 7.5. Event 4 denotes the instant in time when Y becomes valid. When the Y values are valid, a small delay occurs while the flip flops register their new inputs, denoted T_{su}. After this setup time, the FSM is ready for another clock edge.

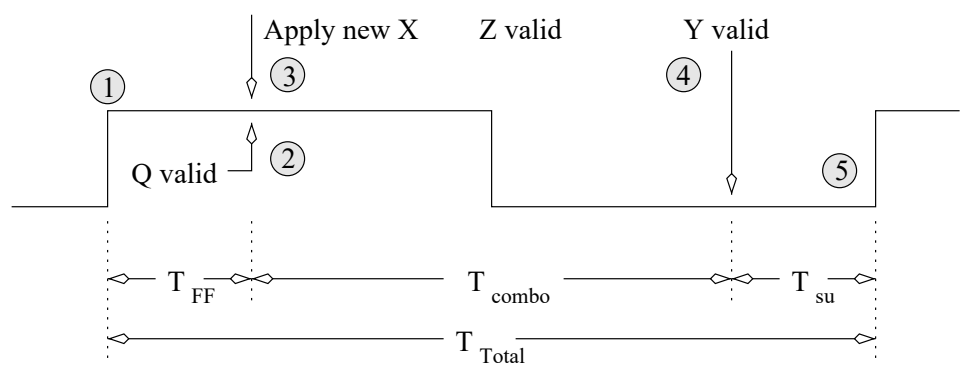


Figure 7.5: A timing diagram showing the sequence of events in a FSM. The circled numbers refer to the sequence of events in a FSM.

The total time between the clock rising and the FSM being ready for another clock edge, $T_{total} = T_{FF} + T_{combo} + T_{su}$, is the minimal amount of time between successive clock edges. If a clock edge arrived sooner than T_{total}, then at the very least, the setup time of the flip flops would be violated. The maximum clocking frequency of a FSM is $F_{max} = 1/T_{total}$.

Consider the operation of the furnace controller through time. To do this, consult the memory input equations of the FSM in Figure 7.4. A timing diagram, Figure 7.6, shows a sequence of inputs applied and the behavior of the circuit. At some time before time=0, the reset signal is asserted to logic 0. This reset event causes the FSM to go into the **OFF** state. The reset signal is released prior to time=0, but the FSM stays in the **OFF** state because it has not seen a positive clock edge. At time=0, a positive clock edge arrives while the temperature is below the low threshold. Hence, the furnace transitions into the **ON** state. The furnace starts producing heat, so the temperature in the house increases. At time=15

the temperature goes above the lower threshold, causing $T_{low} = 1$, but the temperature is still below the higher threshold, so $T_{hi} = 0$. The $Q_{on} * T'_{hi}$ term in D_{on} will equal 1, causing the FSM to transition into the **ON** state, during the next clock edge at time=20; not much of a transition really. Since the furnace remains on, it is still producing heat and increasing the temperature in the house. At time=25, the high temperature threshold is exceeded causing $T_{hi} = 1$ as well as $T_{low} = 1$. The $Q_{on} * T_{hi}$ term of D_{off} will equal 1, causing the FSM to transition into the **OFF** state at the next positive clock edge at time=30. With the furnace off, the temperature in the house starts to drop. At time=35, the temperature drops below the high threshold causing $T_{high} = 0$, but still remains above the low threshold causing $T_{low} = 1$. The $Q_{off} * T_{low}$ term of D_{off} equals 1 causing the FSM to transition into the **OFF** state at the next clock edge at time=40. Again, this change is not much of a transition because the FSM stays in the same state. At time=45, the temperature drops below the low threshold. This change causes the $Q_{off} * T'_{low}$ term of D_{on} to equal 1, causing the FSM to transition to the **ON** state at the next clock edge at time=50.

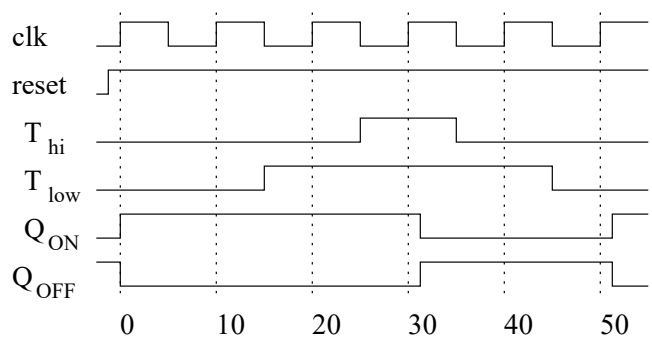


Figure 7.6: A timing diagram for the furnace controller.

The furnace controller is an example of a FSM whose states capture the operating modes of a system. The next example, the vending machine, is a FSM whose states capture the history of the applied inputs.

7.4 Vending Machine

Consider the design of a FSM controlling the operation of a very basic vending machine. This machine is not a very user-friendly design as it does not give out change.

Word Statement Design a FSM which has two inputs N, D representing a nickel and a dime, respectively. $N = 1$ when a nickel has been deposited and $D = 1$ when a dime has been deposited. N and D never simultaneously are equal to 1 because a nickel and a dime cannot be simultaneously inserted into the coin slot. The FSM has one output; it equals 1 when 35¢ or more has been deposited. After dispensing a soda, the FSM should start collecting money for another soda.

State Diagram Usually two natural choices can be phrased for what the FSM is to remember.

1. The number of nickels and dimes deposited.
2. The total amount of money deposited so far.

Clearly, the second approach is going to yield a FSM with far fewer states, fewer flip flops, and in all likelihood, less logic for the MIEs. Hence, use the second approach and move on to defining the states. The FSM has eight states: 0¢, 5¢, ... 35¢ each representing the total amount of money deposited so far. Clearly, 0¢ should be the reset state. The complete FSM is shown in Figure 7.7.

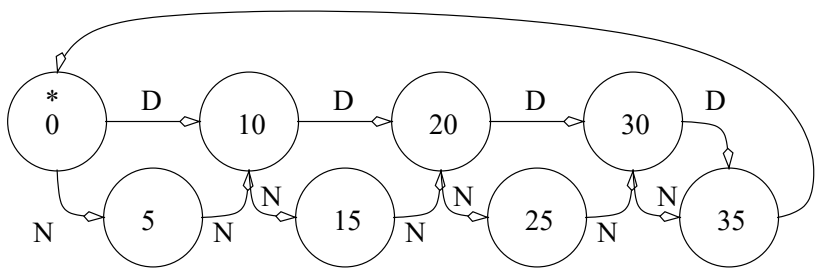


Figure 7.7: The state diagram for the vending machine.

The state diagram is not complete in the sense of the definition given on page 87 because it does not answer the question, "What happen when no money is deposited?" It should be obvious that from any state S, the transition arcs labeled N' or D' point back to S. In other words, if no money is deposited, then the total amount of money deposited does not change. These arcs are omitted from the diagram for the sake of clarity.

Another interesting feature of the state diagram is the arc leaving state 35¢. The arc is not labeled because this arc represents an unconditional state transition; the condition on the arc is assumed to be true regardless of the input condition. The FSM spends only one clock cycle in this state, dispensing a single soda, before transitioning back to state 0¢.

Finally, the FSM is not very user-friendly. A user who deposits 10¢ after depositing 30¢ gets a soda, but no change. The improvement of the FSM is left for a homework problem.

Circuit Diagram The design process of a FSM consists of three steps, determining the number of flip flops, determining the MIEs, and determining the OEs. Since the FSM has eight states, the FSM will have eight flip flops, labeled 0¢, 5¢, ... 35¢.

In writing the MIEs, remember every state **S** that has an outgoing arc labeled N, also has an undrawn arc labeled N'. This undrawn arc leaves state **S** and returns to state **S**. That is, if a nickel is not deposited, then the FSM stays in the same state. Likewise every state which has an outgoing arc labeled D, has an undrawn self arc labeled D'. For example, three arcs terminate at state 5¢. They are the arc from state 0¢labeled N and the two arcs leaving from state 5¢ labeled N' and D'. Hence the MIE for state 5¢ is $D_5 = Q_0N + Q_5N' + Q_5D'$ which can be simplified to $D_5 = Q_0N + Q_5(N' + D')$. The complete list of MIEs is given below.

$D_{35} = Q_{25}D + Q_{30}N + Q_{30}D$
$D_{30} = Q_{20}D + Q_{25}N + Q_{30}(N' + D')$
$D_{25} = Q_{15}D + Q_{20}N + Q_{25}(N' + D')$
$D_{20} = Q_{10}D + Q_{15}N + Q_{20}(N' + D')$
$D_{15} = Q_5D + Q_{10}N + Q_{15}(N' + D')$
$D_{10} = Q_0D + Q_5N + Q_{10}(N' + D')$
$D_5 = Q_0N + Q_5(N' + D')$
$D_0 = Q35 + Q_0(N' + D')$

A table listing all the states and their output values could be built, but an inspection of the word statement reveals that the FSM only outputs 1 when 35¢ has been deposited. Hence,

the FSM should output 1 when it is in state 35¢. Consequently, $Z = Q_{35}$.

7.5 Waits States

When a FSM needs to interface with an entity operating much slower than the FSM or with an event taking an unknown amount of time to happen, it is often necessary to include wait states in the FSM. A wait state is exactly what its name implies, a state in which the FSM waits for some input event to occur.

For example, imagine constructing a FSM to control a vending machine dispenser. The FSM asserts its dispense output, enabling the vending machine to dispense a bag of chips, until its photodetector detects a bag of chips passing down the dispense chute. The photodetector input to the FSM is called *photo* and the dispense signal is an output from the FSM.

This FSM has a (wait) state with a self loop labeled $photo'$ in which it asserts the dispense output. The FSM is waiting for the *photo* input to go to 1, signaling the bag of chips is being dispensed. While the `photo` signal is equal to 0, the FSM asserts the dispense output. The wait state should have a second transition arc labeled `photo` which takes the FSM out of the wait state when *photo* equals 1.

The next section illustrates how to incorporate wait states into a FSM so that it waits patiently for cows to make their way through a cattle chute.

7.6 DAISY

Consider the design of a FSM capturing the current step of a process. The task is to design a high-tech cow-tracking system for a local dairy. The system is called the Dairy Automated Information SYstem, or DAISY for short. The system operates as follows.

Word Statement Cows have a RFID tag attached to their collars. When the cow passes through the cattle chute on their way into the barn, a RFID reader reads the unique ID stored on the RFID tag and logs the cow into the barn. The RFID system outputs a single bit: a 1 means the system has read an RFID tag and has successfully checked a cow back into the barn; a 0 means the RFID system is either still processing a tag or is not currently reading a tag.

In order to ensure each cow is scanned, the flow of cows into the barn is controlled by two gates at either end of the chute. Each gate is controlled by a single bit. To lift a gate, this input must be held at logic 1; to lower a gate, the input must be held at a logic 0. The sequence of raising and lowering the gates in order to control the flow of cows is illustrated in Figure 7.8.

In Step 1, $gate1$ is lifted allowing cow A to enter the chute. In Step 2, the DAISY system has detected cow A is in the chute and closes $gate1$. The cow waits in the closed off chute until the RFID reader signals that it has read the tag and checked in cow A. The DAISY system then raises $gate2$ and waits for cow A to leave the chute before closing $gate2$ and transitioning back to Step 1. If the cow takes more than 30 seconds to leave the chute, then the cow is "goosed" by a three-second burst of compressed air. The compressed air is released by an electronically-controlled valve; asserting a 1 on the valve's input holds the value open, asserting a 0 closes the air valve. The system then waits another 30 seconds before firing the air valve again. This process continues until the cow leaves the chute. At any time when the cow leaves the chute, the cycle is interrupted and the DAISY system transitions back to Step 1.

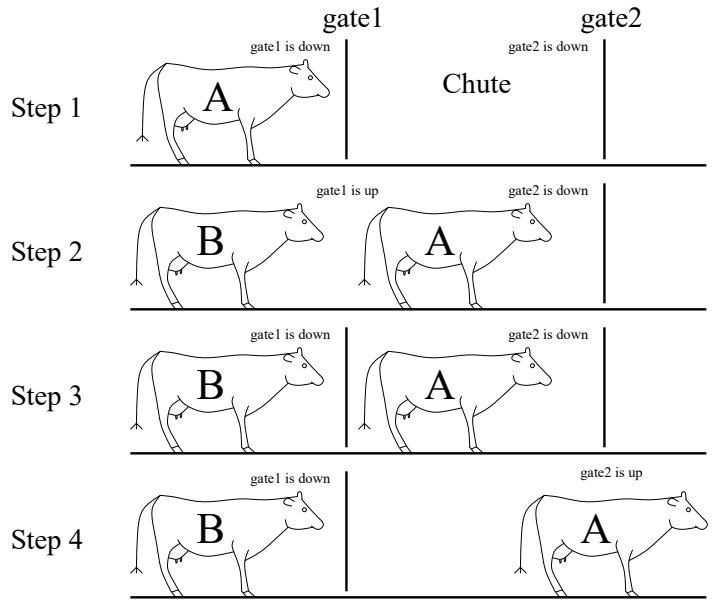


Figure 7.8: The sequence of steps to gate a cow through the RFID reader chute.

In order to give DAISY an accurate sense of time, the system is provided with a single timer with two bits of input and one bit of output. To use this timer, set the timer to either 3 or 30 seconds. By asserting a set command for a single clock cycle, the timer is set. Then, the control input commanding the timer to count down is continuously applied. The output of the timer will equal 0 until the set time limit has expired at which times its output will stay at 1 until a new time interval is set.

Before deriving the state diagram, the inputs and outputs of the DAISY system are categorized.

System Inputs and Outputs The word statement infers the existence of three inputs. The RFID scanner sends the DAISY system a single bit which indicates if the cow has been processed. A second input tells the DAISY system if a cow is in the chute. The final system input comes from a timer used to inform the DAISY system when 3 or 30 seconds have expired.

RFID Scanner = r	Cow Present = c	Timer Status = t
1 - Cow checked in	1 - cow present	1 - timer up
0 - Cow not processed	0 - no cow	0 - timer running

The word statement infers the existence of four, separate outputs. The gates in the DAISY system are controlled by a single bit each. Assume a logic 1 must be continuously applied to a gate in order to keep it raised. In order to use the timer, set it to either 3 or 30 seconds by applying 01 or 10 for a single clock cycle. Then, the timer is run by applying 11. The timer outputs a status signal (a DAISY input) which should be monitored to tell DAISY when the set time limit has expired. When the electronic valve controlling the compressed air is open, air rushes out, goosing the cow.

Gate1	Gate2	Timer Control	Air Valve
1 - gate up	1 - gate up	00 Stop timer	0 closed
0 - gate down	0 - gate down	01 Set timer to 30 seconds	1 open
		10 Set timer to 3 seconds	
		11 Run timer	

State Diagram The process of creating the state diagram for the DAISY system requires considering movement through the steps of the process required to get a single cow through the gated chute. Each step in this process then becomes a state or a set of states. Each state asserts some output to control the devices connected to the DAISY system. Below is one possible list; other arrangements are possible.

1. Open gate1
2. Wait for cow to enter chute
3. Close gate1
4. Wait for RFID to read cow
5. Open gate2
6. Wait for cow to leave
7. If 30 seconds has transpired, then "goose" cow; goto Step 6
8. Else if the cow has left, then close gate2; goto Step 1

In order to simplify the labels on the state diagram arcs, the inputs are abbreviated. The RFID scanner input is represented by the variable r, the cow present input is represented by the variable c, and the timer status input is represented by the variable t.

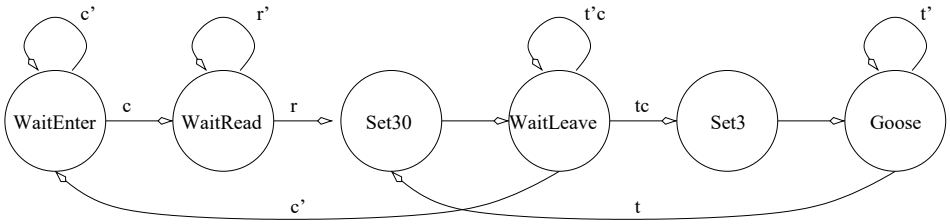


Figure 7.9: The state diagram for the DAISY system.

In some cases the FSM combines a couple of the items in the list together into a single state. For example, Item 1 and Item 2 on the list, "open gate1" and "wait for cow to enter chute" are performed in the **WaitEnter** state. The "Open gate1" action is performed by asserting the $gate1$ output while in this state. The "Wait for cow" action is performed by the self arc labeled c'. That is, while c' is true (while $c = 0$), the FSM waits in the **WaitEnter** state. After a cow moves into the chute, the system moves into state **WaitRead**. This state handles Item 3 and Item 4 in the list, "Close gate1" and "Wait for RFID to read cow". Closing the gate is handled by asserting 0 on the $gate1$ output, while waiting for the RFID reader is handled by the self arc labeled r'. When the RFID reader has checked in the cow, the FSM transitions to the state **Set30** where a 30-second count-down is set. The FSM then waits for the timer to time-out or for the cow to leave.

Timer Status	Cow Present	Action	Next State
0	0	close gate2	WaitEnter
0	1	wait	WaitLeave
1	0	close gate2	WaitEnter
1	1	goose the cow	Set3, Goose

The combinations of timer status and cow present while in the state **WaitLeave** deserves some attention. The state has four combinations of these two inputs which effect the next state. In order to reason out the consequences of each input combination, structuring the analysis can be achieved by putting all four combinations of inputs into a truth table. Then, each combination can be carefully analyzed to determine which next state occurs as well as to simplify the expressions placed on the arcs.

The pair of transitions which lead to the **WaitEnter** state can be simplified by noting that in both cases, the cow present status bit is 0 (denoted by c' on the state diagram).

Memory Input Equations Since the FSM is built with a one-hot encoding of the states, then the MIEs are formed by answering the question, "How did I get into this state?" The answer to this question, answered for each state is given below.

$D_{WaitEnter} = Q_{WaitEnter}c' + Q_{WaitLeave}c'$
$D_{WaitRead} = Q_{WaitEnter}c + Q_{WaitRead}r'$
$D_{Set30} = Q_{WaitRead}r + Q_{Goose}t$
$D_{WaitLeave} = Q_{Set30} + Q_{WaitLeave}t'c$
$D_{Set3} = Q_{WaitLeave}tc$
$D_{Goose} = Q_{Set3} + Q_{Goose}t'$

Output Equations The output table is shown in Table 7.1. The table is referred to as the *control word* table. The term "control" is used because the table describes how the FSM controls its associated hardware. The term "word" is used to indicate the control is formed from a collection of bits.

State	Gate1	Gate2	Timer Control	Air
	1 - gate up	1 - gate up	00 Stop timer	0 - closed
	0 - gate down	0 - gate down	01 Set timer to 30 seconds	1 - open
			10 Set timer to 3 seconds	
			11 Run timer	
WaitEnter	1	0	00	0
WaitRead	0	0	00	0
Set30	0	0	01	0
WaitLeave	0	1	00	0
Set3	0	0	10	0
Goose	0	0	11	1

Table 7.1: The control word table for the DAISY system.

The output equations are derived by looking at the columns of the control word table. Each column is given its own output equation by asking, "What states cause this output to go to 1?" ORing together these states produced the output equation for the corresponding output. For example, the $Gate1$ output only goes to logic 1 when the FSM is in the **WaitEnter** state, hence $Z_{Gate1} = Q_{WaitEnter}$. The complete list of output equations is given below.

$Z_{Gate1} = Q_{WaitEnter}$
$Z_{Gate2} = Q_{WaitLeave}$
$Z_{Timer1} = Q_{Set3} + Q_{Goose}$
$Z_{Timer0} = Q_{Set30} + Q_{Goose}$
$Z_{Air} = Q_{Goose}$

7.7 Exercises

1. A finite state machine has been implemented with four flip-flops, two inputs and three outputs.

   a) What are the minimum and maximum number of states in the diagram?

   b) What are the minimum and maximum number of transition arrows starting at a particular state?

   c) What are the minimum and maximum number of transition arrows ending in a particular state?

   d) What are the minimum and maximum number of different binary patterns that are displayed on the outputs?

2. **(20 points)** The state assignment for a FSM influences the amount of combinational logic required in the realization. In the following problem this phenomena is investigated. Determine the MIEs for the following state table using both the state assignments.

State Table

CS x	0	1
A	A,0	B,0
B	C,1	F,1
C	D,0	C,0
D	A,1	H,1
E	F,1	E,1
F	G,0	F,0
G	G,1	C,1
H	D,1	E,1

State Assignment 1

State	Q_2	Q_1	Q_0
A	0	0	0
B	0	0	1
C	0	1	1
D	0	1	0
E	1	0	0
F	1	0	1
G	1	1	1
H	1	1	0

State Assignment 2

State	Q_2	Q_1	Q_0
A	0	0	0
B	1	1	1
C	0	0	1
D	1	1	0
E	1	0	1
F	0	1	1
G	1	0	0
H	0	1	0

After obtaining the MIEs for both realizations, determine the cost of each solution according to the following formula: $C(FSM) = A + O + 6 * F$. Where $C(FSM)$ denotes the cost of the FSM, A is the cost of the AND gates, O is the cost of the OR gates, and F is the number of flip flops. The cost of an AND gate is equal to the number of inputs to the AND gate. Likewise the cost of an OR gate is equal to the number of inputs. NOT gates are free. For example, the circuit $A'B + ABC'$ costs $(2 + 3) + 2 + 0 = 7$.

   Submit:

   a) Shared steps of the design process.

   b) Derive the MIEs for each of the two realizations.

   c) Determine the cost of each of the two realizations.

   d) Determine the cost of each of the two realizations using Espresso.

3. **(8 pts.)** Realize the FSM in the previous problem using a one-hot encoding. Determine the MIEs and the cost of the circuit using the same metric. It is helpful to convert the state table into a state diagram.

4. **(8 points)** Enhance the vending machine discussed in this chapter as follows. Add two buttons for a beverage selection; *regular* soda and *diet* soda, see Figure 7.10. This machine will have a change dispenser. If the user deposits more than 35¢, the circuit should send a signal to either the *nickel change* dispenser or the *dime change* dispenser, a single bit sent for one clock cycle to a dispenser will yield a single coin. When the user

deposits 35¢(or more) the machine gives any change and then waits for one of the two buttons to be depressed. Depending on the selection, the circuit should send a signal to either the *regular dispenser* or the *diet dispenser* mechanism. The dispenser need only get a signal for one clock cycle. After the dispensing, go back to the reset state.

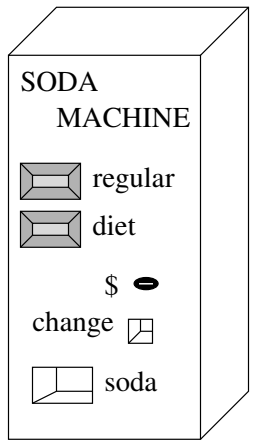


Figure 7.10: A basic vending machine.

5. **(6 pts.)** Build a FSM for a car alarm. The input to the FSM comes from a tilt sensor. The tilt-sensor outputs 1 when the car has been physically displaced by a preset amount, otherwise the tilt sensor outputs 0. The output of the circuit drives an alarm, when the alarm output equals 1 the alarm sounds, otherwise the alarm does not sound. Once the alarm has been set off, it will continue sounding until a reset input equals 1, at which point the alarm will stop sounding.

   Draw the state diagram and from this determine the MIEs and OEs.

6. **(12 pts.)** Build a digital circuit which controls an automatic garage door opener. The garage door circuit has three bits of input. The first input, called *button*, comes from a the main control button used to open or close the garage door. When pressed $button = 1$ otherwise $button = 0$. The garage door rides in a track, at the top and bottom of of which are two limit switches. The top limit switch equals 1 when the garage door is all the way up, otherwise its output equals 0. The bottom limit switch equals 1 when the garage door is all the way down, otherwise its output equals 0. The garage door circuit has two bits of output called *motor*. When $motor = 01$, the motor moves the door in a downward motion, closing the door. When $motor = 10$, the motor moves the door upward, opening the garage door. When $motor = 00$, the motor is turned off.

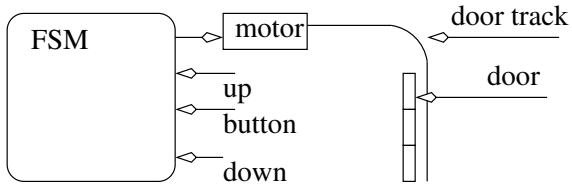


Figure 7.11: The garage door and the circuit controlling it.

Construct the FSM assuming a one-hot encoding of the states. Determine the memory input equations and output equations.

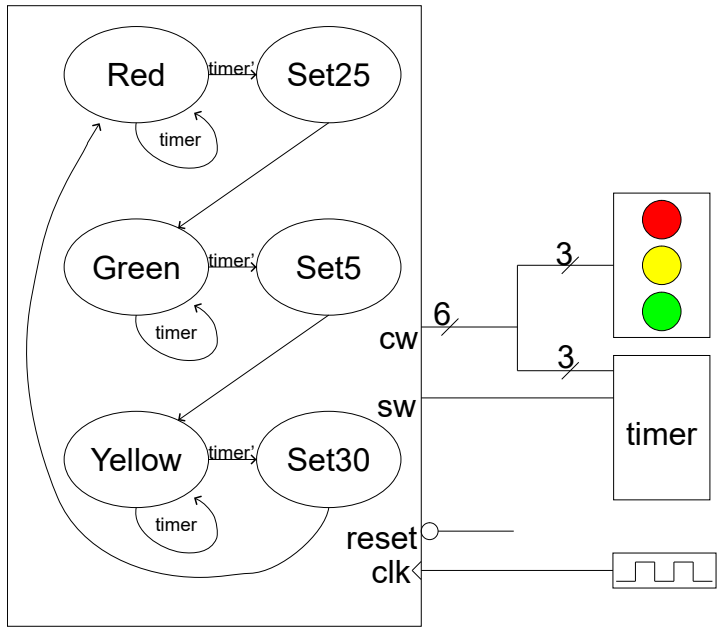


Figure 7.12

7. **(8 pts.)** Build a digital circuit to control a single traffic light. The circuit has three outputs, $Rlight, Ylight$ and $Glight$. When $Rlight = 1$ the red light illuminates otherwise the light is off. The same behavior holds true for $Ylight$ and $Glight$. In order to sequence the lights, the circuit has three timers, Rtimer, Gtimer and Ytimer. Each timer controls the length of time that its light should be illuminated. Each timer has one bit of input and one bit of output. When a timer's input is 0, the timer is reset. When a timer's one bit input is 1, the timer counts down its preset timer interval. When a timer counts all the way down, its output goes to 1 and stays there until the timer is reset (by applying an input of 0). The state diagram of the circuit is shown in the figure below. As shown in figure 7.12 the FSM receives input from the three timers, while the output of the FSM controls the counters. Complete the following three tasks.

   a) Label the arcs of the FSM with the input values (Rt, Yt or Gt) needed to make the circuit operate correctly.

   b) Next, determine what the output should be in each of the states. Instead of writing the output in each state, the outputs are organized in a separate table. In this table, each row will contain the output associated with a particular state. Each column in the table will be associated with one bit of the output.

state	Rlight	Ylight	Glight	Rtimer	Ytimer	Gtimer
	0 off	0 off	0 off	0 rst	0 rst	0 rst
	1 on	1 on	1 on	1 run	1 run	1 run
R						
Y						
G						

   c) Finally, write the memory input equations and output equations for the traffic light controller. In order to write the memory input equations use the labels on the state

transitions from the state diagram. In order to write the output equations use the output table.

$$Q_{red} =$$
$$Q_{yellow} =$$
$$Q_{green} =$$

$$Z_{Rlight} =$$
$$Z_{Ylight} =$$
$$Z_{Glight} =$$
$$Z_{Rtimer} =$$
$$Z_{Ytimer} =$$
$$Z_{Gtimer} =$$

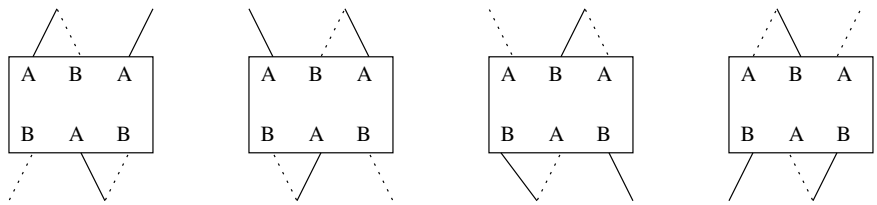


Figure 7.14: The walking gait of the hexapod robot.

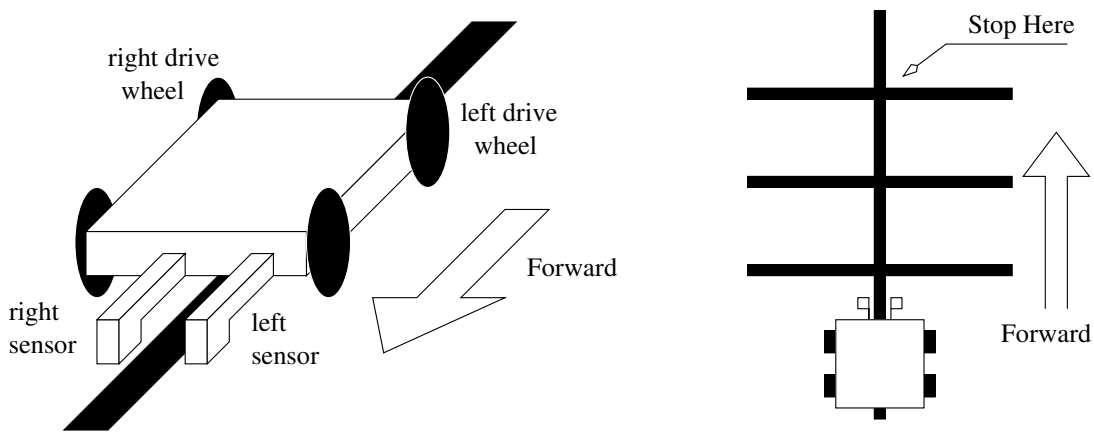


Figure 7.15: A simple line-tracking robot that must cross two intersections.

8. **(16 pts.)** Build a FSM which make the hexapod robot shown in Figure 7.13 walk forward.

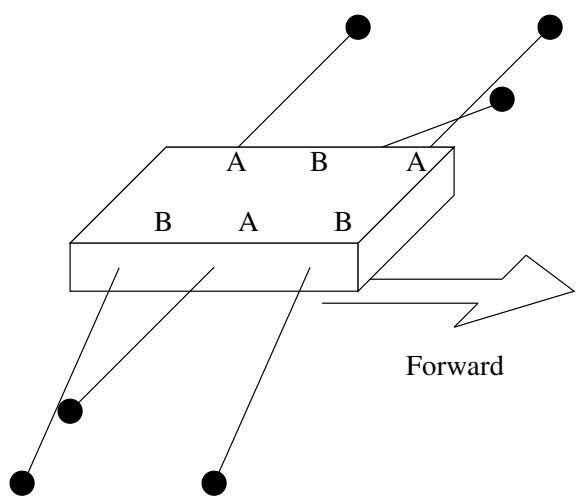


Figure 7.13: A hexapod walking robot.

Each leg of the hexapod robot is moved by two nitinol (Flexinol) wires. At rest, nitinol wire is straight. When 5 volts (logic 1) is applied to the wire, it bends in a particular direction. The two wires making up a particular leg are positioned so that they move in

perpendicular directions. One wire moves a leg up or down and the other will move the wire forward or backwards. The table below elaborates.

wire	logic 0	logic 1
w_0	down	up
w_1	forward	backward

The hexapod robot walks by moving three legs in unison; see A and B in Figure 7.13. The movements of A and B are coordinated so that, at times, the hexapod is balanced on three legs. A portion of the walking gait is shown in Figure 7.14; note that in this figure the viewer is looking down at the top of the robot which is moving to the right. The dotted legs are assumed to be in the air, solid legs are in contact with the ground.

Assume that the legs can move to their correct position in one clock cycle. Define each state as a position of the legs in Figure 7.14. Draw the state diagram, and determine the memory input and output equations.

9. **(12 pts.)** Build a FSM which makes the simple robot shown in Figure 7.15 move along (track) the black line crossing two intersections.

   The FSM has two inputs, a left sensor, denoted *ls*, and a right sensor, denoted *rs*. These two sensors look down at the ground. When a sensor sees white, it outputs 0; when it see black, it outputs 1. The sensors are spaced far enough apart that they can straddle the black line and see white on either side. The FSM has two outputs, a left motor, denoted *lm* and a right motor, denoted *rm*. A motor rotates when it is sent a 1 and does not rotate when it is sent a 0. The FSM should constantly check that the robot is straddling the line. If it is not the FSM should take corrective action by stopping one of the wheels.

   Submit the state diagram for the FSM, MIEs and OEs using a one-hot encoding of the states.

10. **(16 pts.)** Make the robot from Problem 9 cross 63 intersections. The problem is that the number of states will grow to large to handle with a FSM by itself. Additional hardware, in the form of a counter and comparator, are added to the FSM to address this problem.

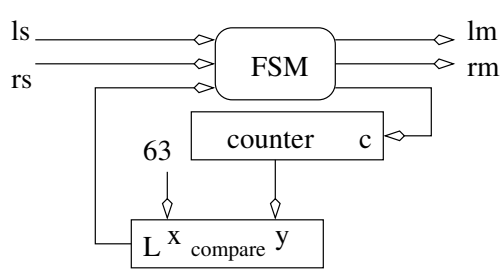


Figure 7.16: The innards of a intersection counting, line tracking robot.

Assume that the counter is reset to 0 when the circuit is first turned on. The robot must still track the line, but must also count up once every time it crosses an intersection. Remember the digital circuit shown in Figure 7.16 is operating much faster than the robot is crossing intersections.

The state diagram needs to have wait states while it crosses the intersection, similar to those in the DAISY example. Assume the counter counts up when the control input is 1 and the clock rises. When the control input is 0, then the counter holds its current count value.

Submit the state diagram for the FSM, OEs and MIEs for a one-hot encoding of the states.

11. **(24 pts.)** Construct a digital circuit to control the operation of a simple washing machine, see Figure 7.17.

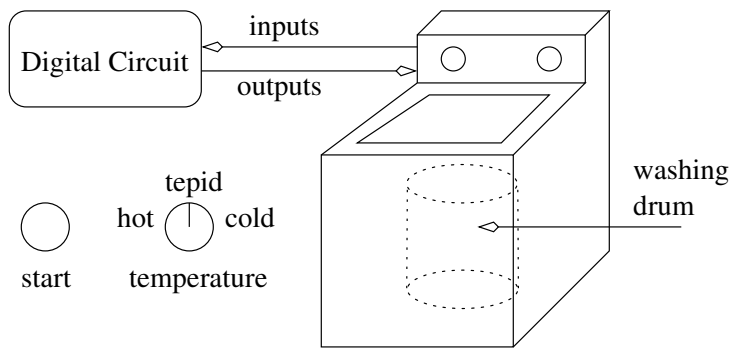


Figure 7.17: A humble washing machine with a close-up of the start button and temperature switch.

To use the simple washing machine set the temperature switch to either hot, tepid or cold and then press the start button. To build a digital circuit to control the washing of clothes its necessary to understand the washing cycle. When the start button is pressed water of the selected temperature pours into the washing drum. The simple washing machine has two electronically controlled water valves, the hot valve admits hot water into the washing drum and the cold valve admits cold water. Water continues to pour into the drum until it fills. There are two water level sensors; the full switch signals when the drum is full of water and the empty switch signals when the drum is empty. After the drum is full of water the simple washing machine starts to agitate the clothes. The simple washing machine has a motor controlled by two bits, which agitates (a rapid back and forth motion), spins (a rapid rotation in one direction) or does nothing. After agitating for 15 minutes, the agitation cycle stops and the machine drains its water. Water leaves the drum through a drain valve. When the drum is emptied of water the washing machine enters the rinse cycle. The rinse cycle fills the drum with cold water and agitates for 5 minutes. The rinse cycle concludes by draining the water from the drum. When the drum is emptied of water the washing machine enters the spin cycle. This lasts for 5 minutes. The simple washing machine keeps track of time using a 5 minute timer. To use the timer it must first be reset for one clock cycle. After being reset the timer will count down as long as the timer input is set to run. After 5 minutes have elapsed the timer output will go to logic 1 and stay there until the timer is reset. In order to get longer time intervals, the timer should be reset for another 5 minutes and count down again. When the spin cycle is done, the washing is complete. The inputs from the washing machine to the digital circuit have the following meaning.

inputs to the digital circuit				
start	temperature	empty	full	timer out
0 off	00 hot	0 not empty	0 not full	0 nothing
1 on	01 cold	1 empty	1 full	1 5 minutes elapsed
	10 tepid			

The outputs from the digital to the washing machine have the following meaning. The left most column is explained below.

outputs from the digital circuit					
state	hot	cold	motor	timer in	drain
	0 close	0 close	00 off	00 hold	0 close
	1 open	1 open	01 agitate	01 reset	1 open
			10 spin	10 run	
S1					

Draw the state diagram for the FSM to control the washing machine. Label the arcs of the state diagram with the input (or its negation) that causes the transition. Use simple Boolean expressions on these arcs, for example (start and hot). For each state define the output using a table similar to the one above. For example, if **S1** is a state fill in the bit values for the o outputs depending on what state **S1** is supposed to do. Determine the memory input equations and output equations assuming a one-hot encoding.

12. **(36 pts.)** Construct a digital circuit to control the movement of an elevator in a four-story building. The elevator will always wait on its current floor until a call button is pressed; see Figure 7.18. The elevator then moves to the floor that was called. The elevator then opens its doors and waits for an elevator control button to be pressed. If a a call to another floor is received before an elevator control button is pressed, then the elevator closes the doors and goes to the new floor. When a floor is selected on the elevator control panel, then the door close and the elevator moves to the desired floor.

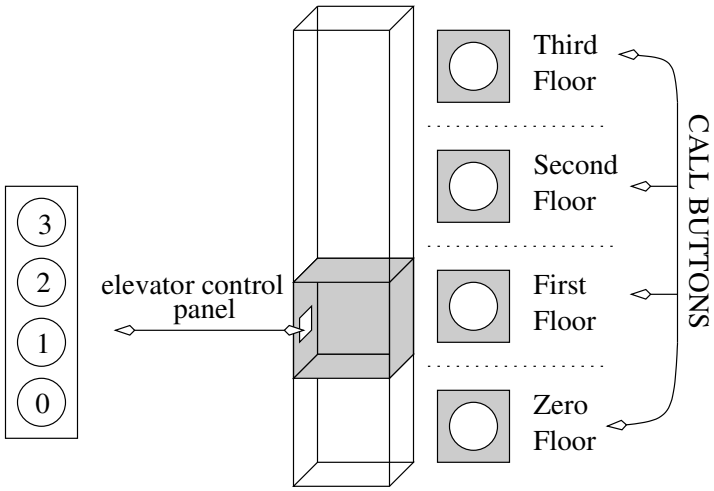


Figure 7.18: The layout of an elevator in a four story tall building.

The inputs to the digital circuit clearly include all the buttons. When a button is pressed on the elevators control panel two things happen. A 2-bit binary value representing the

button pressed becomes valid and a 1-bit panel request becomes valid. The panel request line will remain valid until acknowledged.

When a call button is pressed two things happen. A 2-bit binary value representing which call button was pressed becomes valid and a 1-bit call request becomes valid. The call request line will remain valid until acknowledged.

Another input tells the circuit when the elevator is or is not aligned with a floor. For example, consider an elevator moving from the first to the third floor. Initially, the align variable is 1. When the elevator starts to move away from the first floor towards the second, the align variable goes to 0. When the elevator reaches the second floor, the align variable will go to logic 1 and remain there for a short while (at least several milliseconds) because there is some slack allowed in what is considered "aligned". After the elevator passes the second floor, the align variable goes back to 0 and stays there until the elevator reaches the third floor.

Here is the table of inputs to the FSM, their abbreviations, to be used in the FSM, and their meaning.

Control panel floor	Pfloor	2-bit floor number	
Panel request	Preq	The panel has a valid floor	
Call floor	Cfloor	2-bit floor number	
Call request	Creq	The call buttons have a valid floor	
Align	Align	0 not aligned	1 Aligned

The outputs from the digital circuit to control the door and the movement of the elevator.

Panel acknowledge	Pack	Acknowledge the panel request	
Call acknowledge	Cack	Acknowledge the call request	
Door	0 close	1 open	
Motor	00 stop	01 up	10 down

Submit; an algorithm for the datapath and control unit, the control word table, the memory input equations, and output equations.

13. **(36 pts.)** Construct a digital circuit to control the movement of traffic at the four way intersection shown in Figure 7.19.

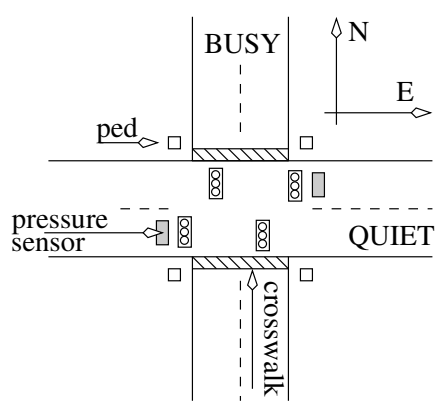


Figure 7.19: The layout of a four way intersection.

The circuit comes equipped with timers. The timer has 2 inputs which set the timer to some preset amount of time or allows the timer to count down. When the timer reaches

0 then the output of the timer goes to 1. When the timer is not at 0 then the output equals 1. The specific inputs and behavior are described in the following truth table.

timer input	behavior
00	count down
01	set to timer to 5 seconds
10	set to timer to 15 seconds
11	set to timer to 30 seconds

In addition to the timer there are a variety of real world inputs sent to the circuit described in the following table.

Name	Abbreviation	Function
E or W Pressure Sensor	EW-PS	1 if 250 lb. or more on E or W sensor
Ped button	ped	1 if any pedestrian crosswalk button

The outputs from the digital circuit to control the lights are:

light	0x red	10 yellow	11 green

The main sequence of events is outlined below;

```
while(1) {
    Nlight = Slight = green;
    Elight = Wlight = red;
    wait 30 seconds;
    while ((EW-PS == 0) && (ped == 0));
    Nlight = Slight = yellow;
    wait 5 seconds;
    if (ped == 1) {
        Nlight = Slight = red;
        Elight = Wlight = red;
        wait 15 seconds;
    }
    Nlight = Slight = red;
    Elight = Wlight = green;
    wait 15 seconds;
    Elight = Wlight = yellow;
    wait 5 seconds;
    if (ped == 1) {
        Nlight = Slight = red;
        Elight = Wlight = red;
        wait 15 seconds;
}   }
```

Submit; the control unit, the control word table, the memory input equations, and output equations.

Chapter 8

Datapath and Control

The datapath and control design methodology break the design of digital systems into two components: a datapath and a control unit. The datapath is responsible for all the data manipulations and the control unit is responsible for sequencing the actions of the datapath. The datapath is constructed from the basic building blocks presented in Chapters 4 and 6. The control unit is a FSM.

A digital system built using the datapath and control design approach is still a digital system whose inputs and outputs can be categorized using the terminology introduced Figure 4.1. The digital system shown in this figure is broken down into two components, a datapath and a control unit. The addition of a clock and a reset signal for the sequential logic elements yields Figure 8.1.

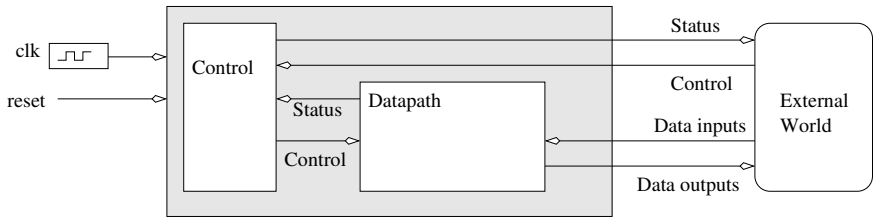


Figure 8.1: An abstract digital system constructed from a datapath and a control unit.

The datapath can perform a variety of data transformations. The control unit instructs the datapath which transformation to perform using a set of control signals called the control word. An often overlooked portion of the control word are signals provided by the external world such as the acknowledge signal in a two-line handshake. The datapath provides status information about the state of its transformation to the control unit. Then, the control unit uses this information to determine the next instruction to the datapath.

A structured approach to designing digital systems with a datapath and control architecture is offered next.

8.1 Conversion

Building digital systems using the datapath and control approach is a three-step process.

1. Write an algorithmic description for the solution to the problem.
2. Parse the algorithmic description into datapath building blocks and control states.
3. Define the MIEs and OEs for the control unit.

The algorithmic descriptions are written in a simple programming language. The algorithmic description is then transformed into hardware by parsing the algorithm one line at a time. Each statement in the algorithm introduces additional building blocks in the datapath and additional states in the control unit. After the algorithm is finished being parsed, the design is completed by deriving the MIEs and OEs of the FSM.

Word Statement to Algorithm

The programming language used to formalize an algorithmic solution to design problem is a derivative of the popular C-programming language referred to as mini-C. The mini-C programming language contains four types of statements.

- `if (condition) then BODY_1 else BODY_2`
- `for (i=A; i<B; i+= 1) BODY`
- `while(condition) BODY`
- `X = value`

The term "BODY" is a place holder for 0 or more statements. In this way, statements can be nested. For example, the body of a for loop may contain a for loop, the body of which may contain a while loop, etc... In addition to statements, the mini-C language also requires variables to hold the state of the program.

In digital circuit design, the variable types in the mini-C language are limited to be either binary or 2's-complement integers. Arrays of these types are common. Limiting the discussion to integer types is not an inherent limitation of the mini-C language, rather it limits the discussion to the essential points of the design process. Complex types like floating-point numbers can be accommodated if the necessary representations and hardware are developed.

No effort is made to explain the process of transforming a word statement into an algorithm; the process should be a familiar task from writing programs. Rather, consideration of the transformation of the algorithm into hardware is presented.

Algorithm to Circuit

From an algorithmic statement, its conversion into hardware is desired. The conversion is accomplished by parsing the algorithm. In computer science, parsing is the process of analyzing a program for its structure. Here, parsing means analyzing a program line-by-line, sequentially, from the first line to the last line, to determine its hardware structure. The analysis process takes a line of code, a mini-C statement, and transforms it into some additional building blocks in the datapath and some additional states in the control unit. When the parsing is complete,

the datapath has all the functionality present in the algorithm, and the control unit has all the control structures present in the algorithm.

As an example of the process, a familiar statement first introduced in Chapter 4, the if/then/else statement, is transformed.

```
if (condition) then BODY_1 else BODY_2
```

When an if/then/else statement is encountered in a program, `BODY_1` is executed when the condition is true. If condition is false, then `BODY_2` is executed. `BODY_1` and `BODY_2` contain 0 or more statements. Typically, the datapath computes the condition using a comparator. In such a case, the datapath requires a comparator, the output of which is the status signal shown in Figure 8.2.

While the control unit is in state **IfThen**, the condition is being evaluated, the status signal is being communicated to the control unit, and the control unit is deciding whether to transition to either the $\mathbf{BODY}_1$ or $\mathbf{BODY}_2$ states. When the clock edge arrives, the control unit will transition to its next state. The $\mathbf{BODY}_1$ or $\mathbf{BODY}_2$ states contain the entire collection of states derived by parsing all the statements in their respective bodies. Regardless of which path the control unit takes, both threads return to the **Next** state, which is the next statement after the if/then/else statement in the algorithm.

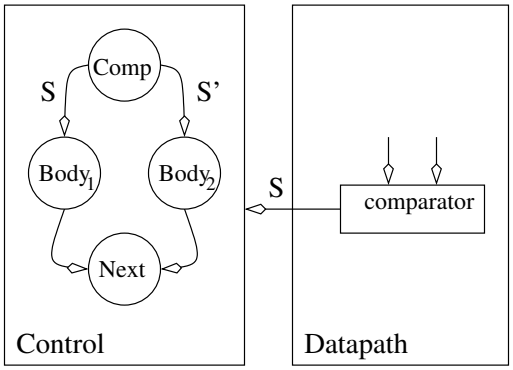


Figure 8.2: The datapath and control components required to realize an if/then/else structure.

```
for(i=A; i<B; i+=1) BODY
```

When a for loop is encountered in a program, `BODY` is executed B-A times and the value of `i` is available for use inside `BODY`. `BODY` contains zero or more statements. A for statement requires a counter and a comparator arranged as shown in Figure 8.3. The initial value of the for loop is the data input to the counter. The output of the counter is the `i` variable of the for loop. The `i` variable is compared to the terminal value of the for loop. The status of this comparison is passed to the control unit so that the control unit knows to terminate the for loop when the counter has reached its terminal value.

The control unit sequences the actions of the hardware in the datapath. The execution of the for loop begins with an initialization of the counter in the **Init** state. In this state, the control unit asserts a load signal on the control lines to the counter causing the counter to be initialized to A. On the next clock edge, the counter loads A and the control unit transitions to the **Comp** state. In this state, the control unit does nothing, giving the comparator time to determine the relative magnitude of i and B, and to assert its L output to the control unit in the form of a status signal. The control unit uses

the status signal to either execute the body of the for loop, or to exit the for loop and proceed with the next instruction after the for loop. The **Body** state represents the collection of states derived by parsing all the statements in the body of the for loop. At the end of the for loop's body, the control unit enters the **Inc** state where the control unit asserts an increment signal on the control lines to the counter. This assertion causes the counter to count up on the next edge which also causes the control unit to transition back to the **Comp** state.

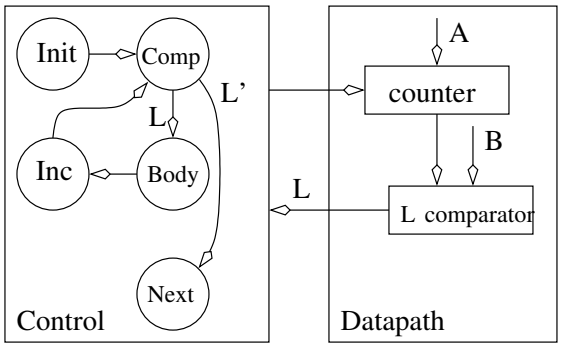


Figure 8.3: The datapath and control components required to realize a for loop.

`while(condition) BODY`

When a while loop is encountered in a program, `Body` is executed while the condition is true. Typically, the datapath computes the condition using a comparator, the output of which is the status signal shown in Figure 8.4. While the control unit is in state **Comp**, the condition is being evaluated, the status signal is being communicated to the control unit, and the control unit deciding on whether to transition to either the **Body** or **Next** states. The **Body** state represents the collection of states derived by parsing all the statements in the body of the while loop.

In some cases the status signal may be determined by some external source. Then, the status line shown in Figure 8.4 as emanating from the datapath would in fact be sent in from the external world as shown in Figure 8.1.

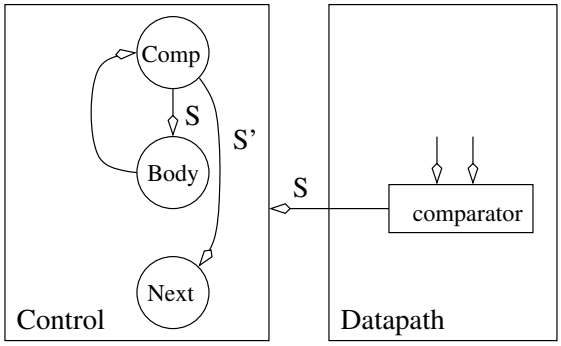


Figure 8.4: The datapath and control components required to realize a **while** statement.

`X = value`

When an assignment statement is encountered in a program, `X` is assigned a new value. This statement is realized by placing a register in the datapath whose input is the value on the right-hand side of the assignment statement. In order to make the assignment, the control unit enters the **Op** state, where it asserts a load signal on the registers control input. On the next positive edge of the clock, the register loads its value and the control unit moves on to the **Next** state.

The size of the register storing X is determined by the range of values required to be stored in X. In some cases, this size is defined by the word statement; in other cases, the designer must make a judgment call on a reasonable value range.

Statements like `X = X+Y` often occur in algorithms. In cases when a variable appears on both the left-hand and right-hand side of an assignment statement, feedback must be employed as shown in Figure 8.5. In this case, the output from the X register is added to Y, the output of the summation is sent to the data input of the X register. The control unit asserts a load signal on the X register's control input when it is in the **Op** state. Most likely, the control unit would assert a hold signal on the Y register's control input while it was in the **Op** state.

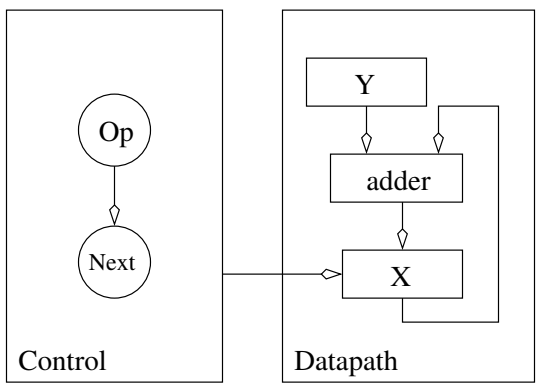


Figure 8.5: The datapath and control components required to realize an assignment statement of the form X+Y.

What prevents the X register from rapidly adding Y to itself multiple times? The answer is that the X register will only latch X+Y on the positive edge of the clock. So X+Y cannot "get into" the X register until the positive clock edge.

A variable often appears on the left-hand side (LHS) of two or more assignment statements. For example, consider a algorithmic description which contains the statements `X=Y` and `X=Z`. In this case, the variable `X` appears on the LHS of two assignments. Since the variable `X` is stored in a register which has a single input, a problem occurs because there are two different sources for the input. This conflict is resolved by inserting a mux between the two data sources and the single data input of the X register as shown in Figure 8.6. The control unit aids in resolving this conflict by asserting control_1 to route the correct value to the data input of register X when the control unit asserts a load signal on the control_2 line.

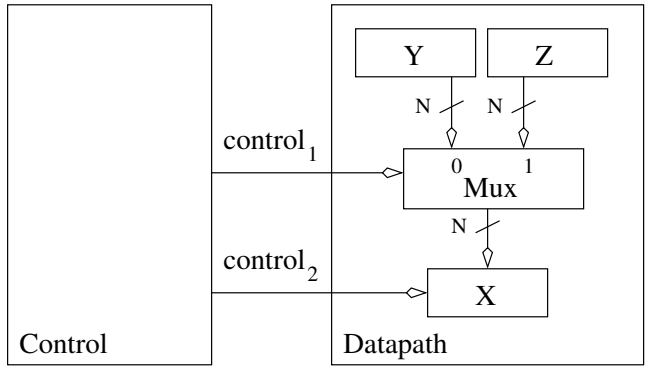


Figure 8.6: The datapath and control components required to resolve the problem of a register requiring two different sources of data input.

Device	Page	Data in	Data out	Status	Control
N:M Decoder	66	1 bit	M bits		N bits
N:1 Mux	67	N bits	1 bit		$\log_2(N)$ bits
M-bit N:1 Mux	69	N, each M-bits	M bits		$\log_2(N)$ bits
N-bit adder	72	2, each N-bits	N bits	Overflow	
N-bit add/sub	73	2, each N-bits	N bits	Overflow	1 bit
N-bit comparator	74	2, each N-bits		3 bits	
BCD to 7-segment	79	4 bits	7 bits		
N-bit priority encoder	81	N bits	$\log(N)$-bits		
N-bit register	102	N bits	N-bits		1 bit
N-bit shift register	104	N bits	N-bits		2 bits
N-bit counter	107	N bits	N bits		2 bits
N:M RAM	110	$\log_2(N)$ bits, M bits	M bit		2 bits

Table 8.1: The list of all the basic building blocks and some of their attributes.

Control Word

After the algorithm is parsed, the design of the datapath is complete and the architecture of the control unit is complete. The details of the control unit, its MIEs and OEs, remain to be defined. A one-hot encoding of the states means the MIEs can be derived directly from the state diagram constituting the control unit. The real work comes from the definition of the control word for each state.

> A control word is a complete listing of the names and meanings of all the control signals sent **from** the control unit **to** the datapath.

Chapters 4 and 6 introduced a variety of basic building blocks with a variety of inputs and outputs. Table 8.1 summarizes all inputs and outputs from these basic building blocks as well as their page numbers.

The control word is defined by listing the control inputs and their effects for every basic building block in the datapath. The control word defines the language of the datapath; any task performed by the datapath must be expressed using this collection of bits. The list of control inputs forms the header of the control word table, the table which contains the control word for every state. The rows of the control word table are labeled with all the state names

used in the control unit. Then, the actions each state needs to perform in the datapath are translated into the language defined by the control word.

In order to give this discussion concrete meaning, a circuit from Chapter 6, the minimum search problem, is reexamined.

8.2 Minimum Search

The minimum search problem on page 116 is reexamined for two reasons. First, the solution presented was unable to initialize the min register. Second, the problem will now be solved using the datapath and control approach, allowing the comparison of two different control approaches. All control decisions in a datapath and control circuit are centralized in the control unit, whereas the control strategy used in the Chapter 6 solution was distributed throughout the circuit.

> Design a digital circuit that looks for the smallest 8-bit integer in a 128x8 RAM. The numbers are stored at addresses 0 . . . 99. Assume the RAM is preloaded with data.

The strategy used to solve this problem is the same as the strategy outlined on page 116: Compare successive elements of the RAM to the smallest value found so far, and update the smallest value if the RAM value is smaller. The caveat of the strategy is to initialize the value of the register holding the minimum value found so far to the largest possible 8-bit value, 0xFF.

```
1.  min = 0xFF;                // Set the min reg to largest 8-bit value
2.  for (i=0; i<100; i++) {    // Search through the entire array
3.      MBR=RAM[i];            // Read an 8-bit value from the RAM
4.      if (MBR<min) then      // If MBR is smaller than min
5.          min = MBR;         //   then set min to the smallest value
6.  } // end for
```

Now the translation of this algorithm to hardware proceeds by examining line-by-line the algorithm and transforming each statement into some datapath and control.

Line 1. The min register is initialized to the largest 8-bit integer, in this case hexadecimal FF. When this assignment statement is parsed according to Figure 8.5, one state is added to the control unit and a register to the datapath. The state is called **InitMin**. Other initialization states in the control unit are to be expected and each should be given a distinct name. An 8-bit register, labeled min, is placed in the datapath. Since min is on the LHS of two assignment statements (Line 1 and Line 5), place a 8-bit 2:1 mux in front of the min register; see Figure 8.6. When the control unit is in the **InitMin** state, it asserts a load signal on the min register's control input and should route 0xFF through the min register's mux.

Line 2. The for loop is used to search every address in the RAM looking for the smallest value. When this for loop is parsed according to Figure 8.3, three states are added to the control unit and a counter and comparator is added to the datapath. In Figure 8.7, the **InitI, CompC, Inc** states correspond to the **Init, Comp, Inc** in Figure 8.3, respectively. The body of the for loop is composed of Lines 3-5. The **Body** state of the for loop in Figure 8.3 is composed of the **Read, CompM, NewMin** states in Figure 8.7. The output of the counter is sent to a comparator whose output, labeled IC, is sent to the control unit so, allowing the for loop to acknowledge when completed.

When the control unit is in the **InitI** state, it asserts a load signal on the counter's control input. In the **CompC** state, the control unit should have the counter hold its value. In the **Inc** state, the control unit should assert an increment signal on the counter's control input.

Line 3. In this line, the RAM is read and its value is stored in the MBR. When this assignment statement is parsed according to Figure 8.5, one state is added to the control unit, and a register and RAM is added to the datapath. The state **Read** in Figure 8.7 corresponds to the state **OP** in Figure 8.5. Since the expression `RAM[i]` is on the RHS of the assignment statement, the RAM provides the data to the MBR register. When the control unit is in the **Read** state, the enb signal is asserted on the RAM's control input and a load signal is asserted on the MBR register's control input.

Line 4. The output of MBR is compared to min. When the if/then/else statement is parsed according to Figure 8.2, one state is added to the control unit and a comparator is added to the datapath. The state **CompM** in Figure 8.7 plays the role of the state **IfThen** in Figure 8.2. This state uses the output of the comparator labeled MC to determine which state to enter next. If min is less than the MBR, MC=1 and the control unit goes to the state **NewMin** in the next clock cycle. Otherwise, the control goes to the state **Inc**.

Line 5. The line of code is only executed if MBR is less than min. When the assignment statement is parsed according to Figure 8.5, one state is added to the control unit and no additional hardware to the datapath because the min and MBR registers are already in the datapath. The state **NewMin** in Figure 8.7 corresponds to the state **OP** in Figure 8.5. When the control unit is in the **NewMin** state, a load signal is asserted on the min registers control input and routes MBR through the min register's mux.

Line 6. The line of code halts the machine. When the while loop is parsed, one state is added to the control unit and no hardware to the datapath. The **Done** state in Figure 8.7 corresponds to the state **Comp** in Figure 8.4. The unconditional self arc to/from the **Done** state halts a FSM when it gets to the **Done** state. However, in most cases the circuit will restart its primary operation from the beginning when it has completed one iteration.

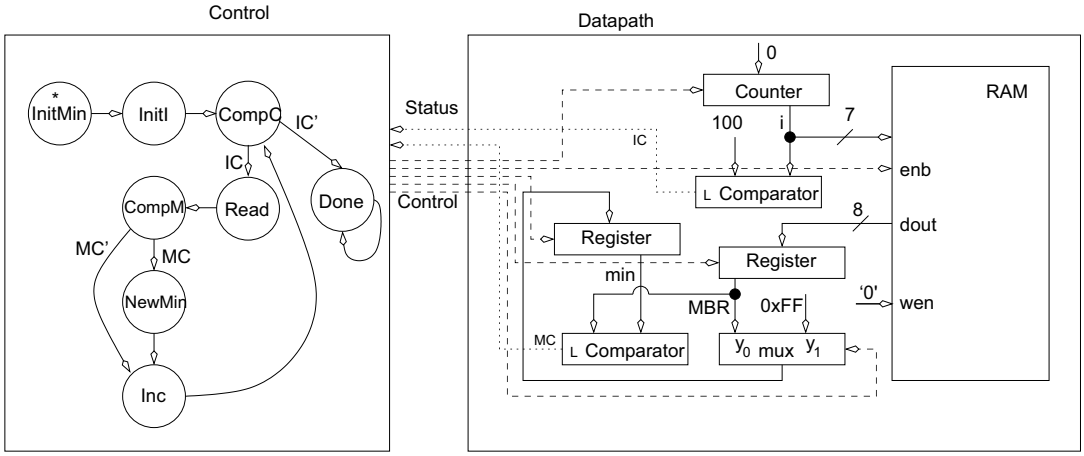


Figure 8.7: The datapath and control components required to implement the minimum search circuit.

The dotted lines representing the status information are generated in the datapath by comparators and sent to the control unit which uses them to decide which course of action

State	enb	Min Reg	Min mux	Counter	MBR Reg
	0	0 hold	0 load FF	00 hold	0 hold
	1 read	1 load	1 load RAM	01 load	1 load
				10 count	
				11 reset	
InitMin	0	1	0	00	0
InitI	0	0	x	01	0
CompC	0	0	x	00	0
Read	1	0	x	00	1
CompM	0	0	x	00	0
NewMin	0	1	1	01	0
Inc	0	0	x	10	0
Done	0	0	x	00	0

Table 8.2: The control word for the minimum search circuit and its values for each state.

to take next. The dashed lines represent control information generated by the control unit to instruct the datapath which actions to perform. The control lines should not be drawn on the datapath and control circuits as they unnecessarily clutter the figure and do not add any useful information. Since the basic building blocks are assumed to be controlled by the control unit, assume these connections are present even when they are not drawn. Since the control unit handles all processing of the status bits, the status outputs from all the comparators in the datapath must be sent to the datapath. Since it is implicit that the status bits are sent to the control unit, they do not need to be drawn all the way to the control unit. Draw just enough of the status output in order to fit the name of the status signal.

Since a one-hot encoding of the states is employed, MIEs are determined for the control unit by inspection as shown below.

$D_{IM} = 0$
$D_{II} = Q_{IM}$
$D_{CC} = Q_{II} + Q_I$
$D_R = Q_{CC} * IC$
$D_{CM} = Q_R$
$D_D = Q_{CC} * IC'$
$D_{NM} = Q_{CM} * MC$
$D_I = Q_{NM} + Q_{CM} * MC'$

The control word for the circuit is determined by listing every control input in the datapath along with its associated action. In Figure 8.7, the collection of dashed lines forms the control word. It is a good idea to form the control word while parsing the algorithm's statements. By using the table on page 152, the control input to each type of box can be determined.

The header of Table 8.2 is the control word table for the minimum search circuit. Each state is listed as a row in the control word table. The values filled in for each state are determined by examining the actions each state performs and is discussed next.

Determining the values of the control-word bits for each state requires understanding what is supposed to happen in each state. For example, parsing Line 1 resulted in adding the **NewMin** state. In this state, the min register was to be assigned the value 0xFF. The datapath can accomplish this by asserting a "load" to min register and a "load FF" on the Min mux control lines. All the other basic building blocks in the datapath must be inactive, thus the RAM is "turned off" and the counter and MBR are told to "hold" their values. Closely

reading how each line of code is parsed should clearly indicate which control bits need to be set in the remaining states of the control unit.

A note about "don't cares" is appropriate at this point. As a general rule, never set the control input of a sequential device to "don't care." The reason is fairly obvious; since sequential devices have memory, the effects of a spurious operation will be remembered and may result in an erroneous operation later on. On the other hand, a combinational logic block like an mux can have its control input set to "don't care" when its output is not being used because it will not remember this decision in the future. For example, when the min register is holding its value, the Min mux's control input is set to a "don't care." In general, always set a combinational logic device's control input to "don't care" whenever possible to communicate the associated device's output is not being used. While this will not help reduce the complexity of the control unit when implemented with a one-hot encoding, it may be helpful under other encoding schemes. Once the control word is defined for all the states, it is time to determine the output equations for the FSM which is the control unit.

One output equation occurs for each bit in the control word table. For example, there are eight bits of control in the min search control. While seven bits of control may seem correct, remember the counter requires two bits of control, giving eight, total control bits. The output equation for a control bit is the OR of the states which cause the output to equal 1. For example, the ENB output of the control unit equals 1 when the control unit is in the **Read** state. Hence, $Z_{ENB} = Q_R$. All the OEs are summarized in the list below.

$Z_{ENB} = Q_R$
$Z_{RM} = Q_{IM} + Q_{NM}$
$Z_{MM} = Q_{NM}$
$Z_{C1} = Q_I$
$Z_{C0} = Q_{II} + Q_{NM}$
$Z_{MBR} = Q_R$

In order to understand how the elements of the minimum search circuit operate together to complete its task, examine its behavior through time using a sequence of modified circuit diagrams. The state diagram representation of the control unit is replaced with the circuit diagram representation of the one-hot encoded control unit. When a component is active it is shaded. For example, when a local signal is asserted on the min register's control input, the min register is shaded. When a control or status line is active, it is drawn as a solid line instead of a dotted or dashed line. Each diagram represents the circuit in one state; the name of the state is written underneath the circuit diagram.

The operation of the minimum search circuit starts at time=1 in the **InitMin** state, shown in the upper left of Figure 8.8. The flip flop labeled IM (**InitMin**) is shaded grey because its output, $Q_{IM} = 1$; it is the current state. The control word for the **InitMin** state asserts "load" on the min register' control input and "route FF" to the min mux. Hence, these control signals are drawn as solid lines. The MIE, $D_{II} = Q_{IM}$, means that when the clock edge arrives, the control unit transitions to the **InitI** state and the min register will latch the value FF.

When the control unit is in the **InitI** state, $Q_{II} = 1$ causes the OEs to assert a "load" on the counter's control input. The control unit transitions into state **CompC**, where the IC output of the comparator is used to send the control unit to the **Read** state. During this state, the counter is used as the address to the RAM, which sends its data output to the MBR. When the positive edge of the clock arrives, the MBR latches its value and the control unit transitions into the **CompM** state. While in this state, the MC output of the comparator is being used by the control unit to determine whether it should transition into the **NewMin** or the **Inc** state (see Figure 8.7). It is assumed that the value stored at address 0 in the RAM is less than 0xFF, so the MC output is 1 and the control unit transitions to the **NewMin** state.

In this state, the control unit loads the MBR into the min register. When it is in the **Inc** state, the control unit is getting ready to enter another loop of the for loop, by incrementing the counter. It then transitions back into the **CompC** state on the next positive clock edge.

Before moving on with another example, a more detailed examination of the minimum search circuit's timing with the goal of determining the maximum clocking frequency of the circuit is considered.

8.3 Timing

The timing analysis of a datapath and control circuit is based on the behavior of the general model of its organization shown in Figure 8.1. Since the control unit in this figure is just a FSM, understanding the timing analysis of the FSM presented in Figure 7.5 on page 128 is imperative. The goal of this analysis is to determine the maximum clock frequency at which a datapath and control circuit can operate. In order to do this, the worst case delay, called the critical path, between successive clock edges offers insight.

The positive edge of the clock is used as the reference point of the timing analysis since this is when the primary source of change in the circuit, the D flip flops latching their values, occurs. The positive clock edge has two primary effects: It causes the FSM to transition into a new state and it causes the registers in the datapath to latch new values. The propagation delay of the flip flops is referred to as $T_p(A)$ in Figure 8.9. Since the datapath requires a valid control word before it can begin, the critical path includes the output equation logic.

The D flip flops which store the state of the control unit's FSM are the input of the OEs; see Figure 7.1. The delay between the application of a valid Q to when the OEs assert their new values is referred to as $T_p(B)$ in Figure 8.9.

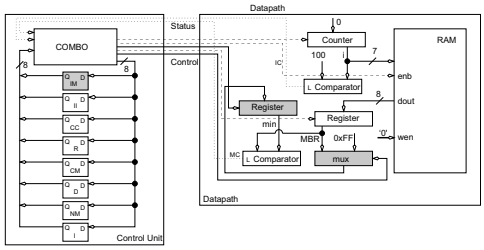
The OEs of the FSM are the control word of the datapath and control circuit, telling the elements in the datapath what operation to perform. Sequential logic components do not actually perform their instructed operations until the next clock edge arrives. On the other hand, combinational logic components perform their operations immediately. It is easy to construct datapath instances where the control word effects the status input to the control unit. For example, the control word selects an input of a mux, whose output is routed to a comparator, whose status output is sent to the control unit. Thus, the combinational logic is on the critical path because its delay constrains the maximum clocking frequency. The time difference between the application of a valid control word to the datapath and the status input to the control unit becoming valid is referred to as $T_p(C)$ in Figure 8.9.

The status input to the control unit are routed to the MIE logic; see Figure 7.1. The delay between the status inputs becoming valid and the MIEs becoming valid is referred to as $T_p(D)$ in Figure 8.9.

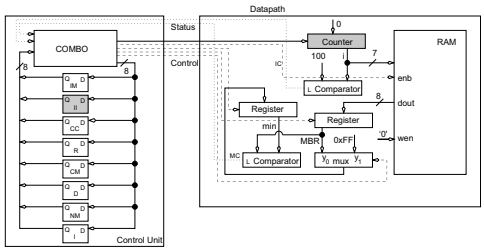
Once the memory inputs have stabilized, they must be allowed some setup time, (see page 95), before the next clock edge. The setup time is referred to as $T_p(E)$ in Figure 8.9.

After the setup time, the outputs of all the circuit elements are stable. Thus, all the flip flops should be ready to latch new values on the next clock edge. Adding together all the delays on the critical path yields the minimum clocking frequency $T_{CRITICAL} = T_p(A) + T_p(B) + T_p(C) + T_p(D) + T_p(E)$. The maximum clocking frequency is the reciprocal of the minimum period.

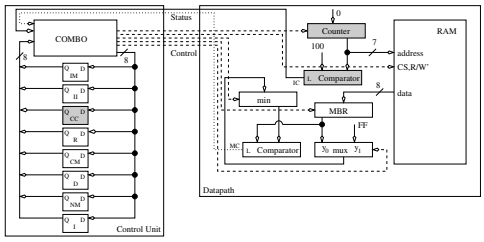
This timing analysis assumes that there are no combinational devices in the datapath requiring more than $T_p(C) + T_p(D)$ time to compute their values. Components like large adders require a lot of time to compute their values and may exceed these time bounds. When this happen these components become part of the critical path in the timing analysis.



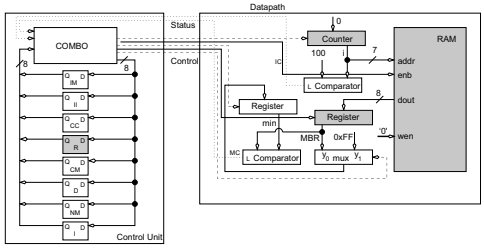
1. State **InitMin**



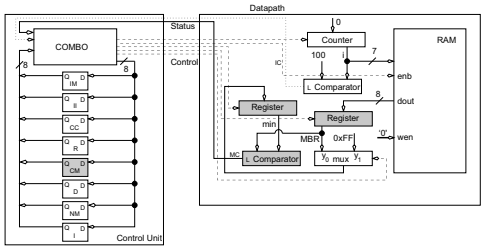
2. State **InitI**



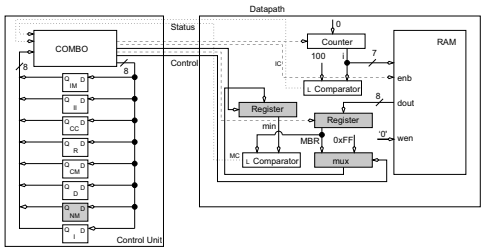
3. State **CompC**



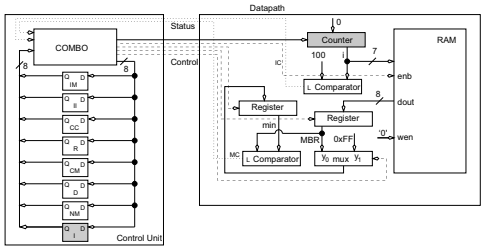
4. State **Read**



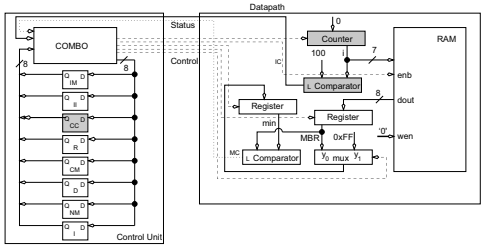
5. State **CompM**



6. State **NewMin**



7. State **Inc**



8. State **CompC**

Figure 8.8: A sequence of figures showing the operation of the minimum search circuit through time. Active components are shaded.

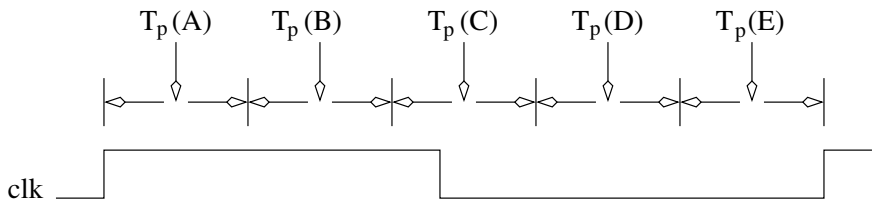


Figure 8.9: A clock waveform annotated with the delays on the critical path of a datapath and control circuit.

Increasing Parallelism

Two approaches are used to decrease the amount of time for a datapath and control circuit to perform a task: To increase the clocking frequency of the circuit, or to decrease the number of steps required to perform a task. The first method relies on a combination of technology and organization of the components in the datapath. The second method relies on increasing the utilization of the hardware components in the datapath and consequently decreasing the number of steps required to perform the task. The second approach follows.

In order to have a datapath perform a task in fewer steps, it is necessary to have the datapath perform multiple steps at the same time. In other words, the goal is to increase the parallelism of the datapath. This modification can be done by combining one or more states of the control unit together. Following some common sense rules accomplishes the change.

Two or more assignment statements can be combined if there are no conflicts in the hardware resources required for the operations or if there is no conflict in the order of operations. For example, the **InitMin** and **InitI** states in the minimum search circuit can be performed at the same time because they operate on separate pieces of hardware.

Successive "branches" of the control unit can be combined when their values are available as shown in Figure 8.10. In Figure 8.10, notice when $X = 1$ and $Y = 1$, the control unit moves from state **A** to state **D** via the state **B**. If states **B** and **C** do not perform any operations, then they can be eliminated by transitioning directly from state **A** to state **D** via an arc labeled XY.

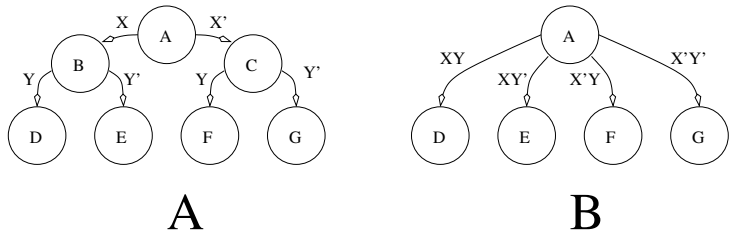


Figure 8.10: Consecutive branches by the control unit can be combined using AND.

8.4 Two-Line Handshake

In most cases, digital systems require data from the external world in order to perform their tasks. In cases where the digital system and the outside word operate on independent clocks, the transfer of data is complicated by the lack of a common clock. To understand how a reliable transfer of data can be performed in this circumstance, consider the following scenario of a producer trying to deliver a packet of candies to a consumer.

Two participants associate in the scenario called *producer* and *consumer*. The producer has a bag of 32 candies; (the number of candies in the bag really does not matter). The candies are to be given to the consumer and the producer is to receive an acknowledgment from the consumer of their receipt. Unfortunately, the producer is blind-folded and is wearing a rather thick pair of ski mittens so they do not know when the consumer has actually taken the candies. The producer and consumer must synchronize the transfer of candies using signals sent with their voices. The transfer protocol is described in the following four steps:

1. The producer stumbles into consumer's room and calls out, "Candies, Candies, ..." (non-stop).

2. The consumer gets up, takes the box of candies and then calls out, "Received, Received, ..." (non-stop).

3. The producer upon hearing the consumer has received the candies stops calling "Candies" and walks out of the room.

4. The consumer upon hearing the producer has stopped calling out "Candies," stops calling out "Received."

Figure 8.11 shows a timing diagram for this scenario. During the time interval labeled 1, both the producer and consumer are quiet. Perhaps the producer is negotiating through the consumer's rooms. During time interval 2, the producer is calling out and the consumer is quiet. The consumer, may be busy with some other task, and is not able to attend to the producer. At the end of time interval 2, the consumer has taken the candies. During time interval 3, perhaps the most annoying time in the scenario, both the producer and consumer are calling out. At the end of time interval 3, the producer has heard the consumer and is about to stop offering candies. At the beginning of time interval 4, the producer becomes quiet; the producer knows for certain the consumer has received the box because the consumer is calling out "Received." At the end of time interval 4, the consumer hears the producer has stopped calling out "Candies." At the beginning of time interval 5, the consumer stops calling out. The consumer knows that the producer knows that the consumer got the box of candies because the producer has acknowledged the consumer's thanks by being quiet.

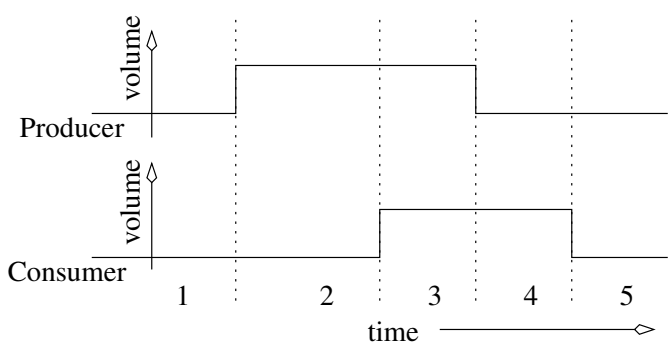


Figure 8.11: A timing diagram of a data transfer between a producer and a consumer.

In the above scenario, the producer is the active agent, the entity initiating the exchange of candies and the consumer is the passive agent, the agent that waited for the candies. This protocol, regardless of who is the producer or consumer, is called a two-line handshake because

the communicating agents must have two, coordinating signals, Request (REQ) and Acknowledge (ACK) and at least one data line. The REQ signal is used by the active agent to signal a readiness to perform a data transfer. The ACK signal is used by the passive agent to acknowledge the data has been transferred. An algorithm description of the two-line handshake for a digital circuit which is the passive consumer is shown below.

```
1.  while(REQ==0);          // Do nothing but wait
2.  register = DATA         // Latch the data
3.  ACK=1;                  // Acknowledge the producer
4.  while(REQ==1);          // Do nothing but wait
5.  ACK=0;                  // Acknowledge the producer
```

In Line 1 and Line 4, the body of the while loops are empty; there is nothing to do but wait. Furthermore, with respect to the external world, (see Figure 8.1), the ACK and REQ signals act as status and command bits, respectively. The algorithm above is translated into datapath and control in Figure 8.12.

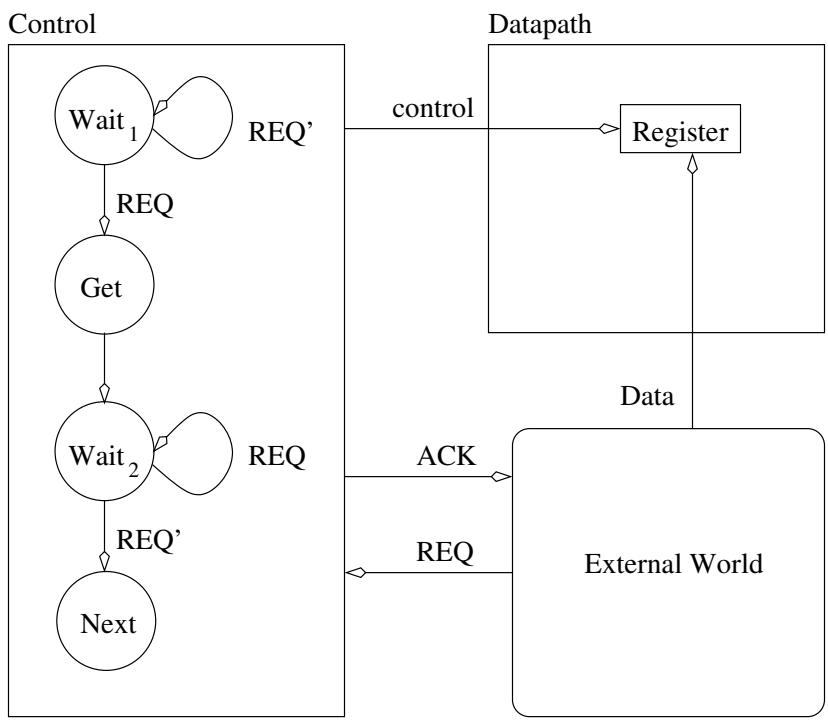


Figure 8.12: The datapath and control components required to implement a two-line handshake where the digital system is the passive consumer.

The most important feature of the control unit are the two self-arcs at states **Wait**$_1$ **and Wait**$_2$. In state **Wait**$_1$, the control unit does nothing except check the value of the REQ signal. As long as REQ=0, the control unit waits. As soon as REQ=1, the control unit proceeds to state **Get** where it enables the register to load the external data. The control unit spends a single clock cycle in this state before moving to state **Wait**$_2$. In state **Wait**$_2$, the control unit asserts and acknowledges, (ACK=1), and waits for the REQ signal to drop. It is important to assert an acknowledge only after latching the data into the register. If an

acknowledge is sent in the **Get** state, then it is possible for a very fast external world to be able to change the data signal before the end of the circuit's clock cycle, giving the wrong data. When the REQ signal is dropped, the control unit goes to state **Next** which represents some further actions expected of the digital system to perform. In state **Next**, (and in all other states except **Wait**$_2$), the ACK signal should be set to 0.

Notice that no matter how different the clock speeds are between the producer and the consumer, this circuit transfers data correctly. If the consumer is faster, it will wait patiently for the producer. If the consumer is slower, it will work as fast as possible to latch the data.

8.5 RAM counter

The following circuit uses a two-line handshake to transfer a data item (called the key) to a digital circuit which scans a RAM, counting the number of words which match the key. The word statement for this problem is:

> Build a circuit to read in an 8-bit KEY using a two-line handshake; the circuit is a passive consumer. The circuit should search an 8kx8 RAM, counting the number of words that match KEY. Assume the RAM is preloaded with data and it can respond to a read request with valid data within one clock cycle. After counting the number of matches, the circuit should wait for another key and repeat.

The algorithm for this circuit needs to read in the key using a two-line handshake and then needs to read through the RAM, one word at a time. Similar to the minimum search algorithm, the RAM is depicted as an array. Since it takes a full clock cycle to read the RAM, then it is best to store the currently read word in a register for further use. A register which buffers the contents of a memory is often called a memory buffer register, or MBR for short. Once the RAM value is in the MBR, the value is compared against the key. If there is a match, then a register called match is incremented.

```
1.  while(1) {
2.      while(REQ == 0);
3.      KEY = data;
4.      ACK = 1;
5.      while(REQ == 1);
6.      ACK = 0;
7.      match = 0;
8.      for(i=0; i<8191; i++) {
9.          MBR = RAM[i];
10.         if (MBR == KEY) {
11.             match=match+1;
12.         } // end if
13.     } // end for
14. } // end while
```

Lines 2-6. The two-line handshake is formed. The datapath and control are similar to Figure 8.12 with the register being named key.

Line 8. The for loop generates the address of each word in RAM. This line of code adds several states to the control unit and adds a counter and a comparator to the datapath. The data output from the counter provides the address input of the RAM.

State	ACK	mux	Reg match	Reg KEY	Counter	MBR	enb
	0	0 pass 0	0 hold	0 hold	00 hold	0 hold	0 inactive
					01 load		
	1	1 match+1	1 load	1 load	10 count	1 load	1 read
Wait$_1$	0	x	0	0	00	0	0
Get	0	x	0	1	00	0	0
Wait$_2$	1	x	0	0	00	0	0
match	0	0	1	0	00	0	0
Init	0	x	0	0	01	0	0
For	0	x	0	0	10	0	0
Read	0	x	0	0	00	1	1
Comp	0	x	0	0	00	0	0
Inc	0	1	1	0	00	0	0

Table 8.3: The control word for the RAM match circuit and its value for each state.

Line 9. The assignment statement takes the data output from the RAM and sends it to the MBR register. While in this state, the control unit should read from the memory and assert load on the MBR's control input.

Line 10. The comparison adds a state to the control unit in which the E output from the comparator is used in the datapath to determine if it should increment the match.

Line 11. The assignment statement increments the number of matches if the key is equal to the current memory word. There are two reasonable hardware solutions for the match register, a counter or a register with an adder. The solution chosen largely is a matter of preference or of available hardware. In the solution presented, the register with an adder combination is used. The completed datapath and control is shown in Figure 8.13.

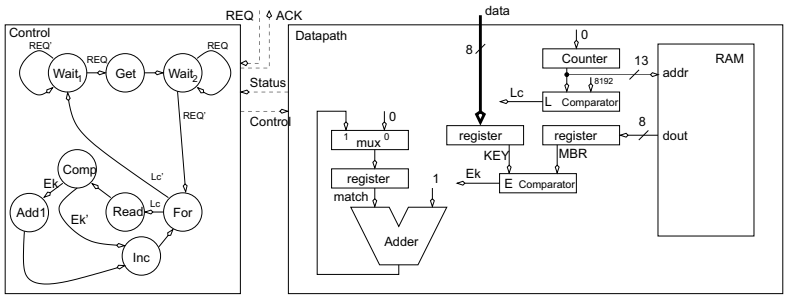


Figure 8.13: The datapath and control for the RAM match circuit.

The control word, shown in Table 8.3, is formed by enumerating the control inputs in the datapath, and by listing their values for each state in the control unit.

8.6 Keyboard Scancode Reader

Even though PS/2 keyboards are being replaced with their USB counterparts, they are still a common piece of technology that can easily be incorporated into digital systems with the help of a scan code reader. Before building a scan code reader, understanding how a keyboard works must come first. Using the input, output, and behavior tables utilized in Chapters 4 and 6 supports this goal.

Nomenclature:	PS/2 Keyboard
Data Input:	none
Data Output:	1-bit data, nominally logic 1
Control:	none
Status:	none
Others:	1-bit clk, nominally logic 1
Physical Input:	key press and key release events
Physical Output:	none
Behavior:	When a key is pressed, its 8-bit make code is transmitted. When a key is released, an 8-bit break code is transmitted, immediately followed by the key's 8-bit scan code.

While the table implies a keyboard is an output-only device, the truth is the clock and data lines are open collector signals. In other words, the clock and data lines can safely be manipulated by the external world to configure a keyboard. A common example of such bidirectional communication occurs every time the "Caps Lock" key is pressed on a keyboard. When this happens, the keyboard sends the "Caps Lock" scan code to the PC and the PC in return writes a "Toggle Caps Lock LED" command to the keyboard. Since the keyboard scan code reader does not write to the keyboard, it assumes that the clock and data signals are outputs from the keyboard.

When a keyboard key is pressed, the keyboard sends one packet of information as shown at the top of Figure 8.14. The 8-bit data contained in this make code is the scan code of the key pressed. The relationship between the keys and their scan codes is not at all obvious and is not based on ASCII. The exact codes are immaterial to the discussion; the curious reader can perform a quick Internet search on "PS/2 keyboard scan codes" to get a complete listing. When a key is released, two packets are transmitted as shown at the top of Figure 8.14. The break code is almost always equal to 0xF0. The final packet is the scan code of the released key.

While there is a scan code for "a" key, there is not a scan code for "A". The device reading the keyboard interprets the make code for "shift," and then sees a make code for "a". From this, the device reading the keyboard should understand that the user wants a capital "A". More than likely, the user will release the "a", first causing its break code and scan code to be transmitted, followed by the break and scan code for the "shift" key.

Each of these packets consists of 11 bits as shown in the lower half of Figure 8.14. The data from the keyboard is always valid on the falling edge of the clock signal. The keyboard asserts new data on or around the rising edge of the clock. The 11-bit data packet always begins with a start bit equal to 0. Following the start bit are 8-bits of data, transmitted least-significant bit first. Following the data bits is an odd-parity bit, whose value is set by the keyboard so that the total number of 1s transmitted in the eight data bits plus the parity bit equals an odd number. For example, if the eight data bits are 01100011, then the parity bit would equal 1 so that the total number of 1s would be an odd number, in this case 5. By adding some additional circuitry, the parity bit can be used to detect errors in transmission. Following the parity bit, the final bit of the data packet, the stop bit, is sent and is always equal to 1.

The keyboard scan code reader circuit assumes only one key is pressed at a time. That is, while a key is being held down, no other key is assumed to be pressed. Thus, typing in "A" may result in the keyboard scan code reader generating an invalid output. Furthermore, the circuit ignores the parity bit, forgoing any possibility of checking for errors in the transmitted data. The input, output, and behavior of the keyboard scan code reader is summarized in the following table.

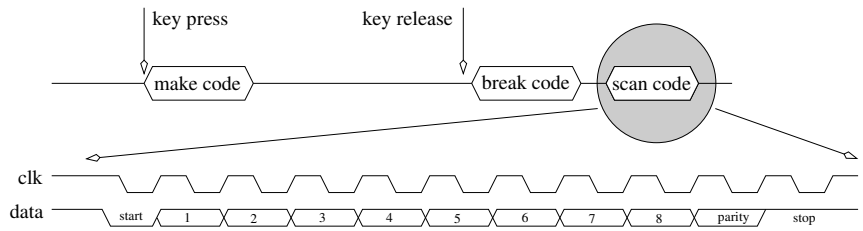


Figure 8.14: The behavior of the keyboard clock and data lines to a key press event.

Nomenclature:	Keyboard scan code reader
Data Input:	1-bit kd_data, nominally logic 1 1-bit kd_clk, nominally logic 1
Data Output:	8-bit scan code
Control:	none
Status:	1-bit busy, nominally logic 0
Others:	1-bit clk, nominally logic 1
Behavior:	Interprets the PS/2 keyboard clk and data signal from a keypress event and outputs the associated scan code. The busy signal goes high when the first data bit arrives and stays high until the last data bit is received. Busy is low only when there is a valid scan code on the output.

The keyboard scan code reader algorithm just shifts in the 33 bits sent from the keyboard and holds the busy signal high while doing this. The scan code is extracted from the low-order bits of the shift register outputs.

```
1.  while(1) {
2.      busy=0;
3.      while (kb_clk == 1);
4.      busy=1;
5.      for (count=0 count<33; count++) {
6.          while(kb_clk == 1);
7.          shift = (shift << 1) | kb_data;
8.          while(kb_clk == 0);
9.      }
10.     scan = shift[9-2]
11. }
```

Note the keyboard clock, kb_clk, signal is being treated as an ordinary data input signal, rather than as a clock signal. This decision seems to increase the complexity of the solution, but actually makes implementing the circuit easier on FPGAs because the clock does not require special clock net resources and it makes integrating clock debouncing circuits easier. The keyboard scan algorithm is converted into datapath and control by parsing it line-by-line.

Line 1. The while loop formed by Line 1 and Line 9 means the scan code reader is expected to operate forever. This statement is responsible for the **start** state in Figure 8.15.

Line 2. The assignment statement can be handled with the control unit because busy is a single bit. Thus, the assignment state does not introduce any states in the control unit.

Line 3. The delay loop waits for the falling edge of the keyboard clock. Introducing a 1-bit comparator in the datapath to check if kb_clk is equal to 1 is wasteful because the E

output of the comparator is equal to kb_clk itself. Hence, the kb_clk signal is sent directly to the control unit as a status signal. The while loop is responsible for the **while** state in Figure 8.15.

Line 4. The statement is incorporated into the control word. Its value is assigned in the states which make up the surrounding statements.

Line 5. This line of code requires a 6-bit counter and a 6-bit comparator to be added to the datapath. The statement is responsible for the **clear**, **inc** and **done?** states in Figure 8.15. Note that the **done?** state closes the infinite loop started in Line 1.

Line 6. The delay loop in Line 3 is used to distinguish the start of a keypress event and consequently, when to set the busy bit to 1. The delay loop on Line 6 indicates the presence of a negative edge on the keyboard clock and hence a valid data bit. This statement is responsible for the **wait1** state in Figure 8.15.

Line 7. In order to get to Line 7, a negative edge on kb_clk occurred. Hence, the keyboard data is latched up into a shift register. This statement is responsible for the **shift** state in Figure 8.15.

Line 8. A delay loop is waiting for the rising edge of the keyboard clock. As with Line 3 and Line 6, this statement requires no hardware in the datapath. The statement is responsible for the **wait0** state in Figure 8.15.

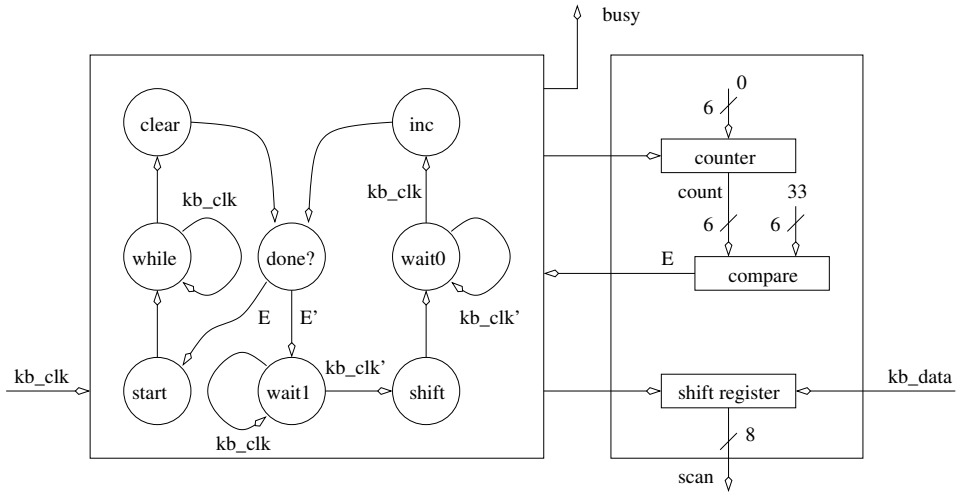


Figure 8.15: The datapath and control for the kbscan circuit.

The control word is formed by writing down the control settings for all the components in the datapath. Next, each state is listed, each in its own row, and the binary control word for each state is defined in Table 8.4.

This concludes the construction of the keyboard scan code reader circuit. The circuit, however, is used in the following section in order to build a circuit to generate a light show.

State	Counter	Shift Reg	Busy
	00 hold	00 hold	0 valid output
	01 load	01 load	1 invalid output
	10 count	10 shift left	
		11 shift right	
start	01	00	0
while	00	00	0
start	00	00	0
clear	01	00	1
done?	00	00	1
wait1	00	00	1
shift	00	11	1
wait0	00	00	1
inc	10	00	1

Table 8.4: The control word for the keyboard scan circuit and its values for each state.

8.7 Light Show

A light show consists of an endlessly repeating sequence of up to 16 frames. A frame is an illuminated pattern of LEDs on a LED bar graph. The user creates a light show by specifying the number of frames in the show, editing those frames, and then instructing the circuit to cycle through the frames. The input to the circuit comes from a standard PS/2 keyboard. The output of the circuit is displayed on a LED bar graph and a 7-segment display. The behavior of the Light Show circuit is given in Figure 8.16.

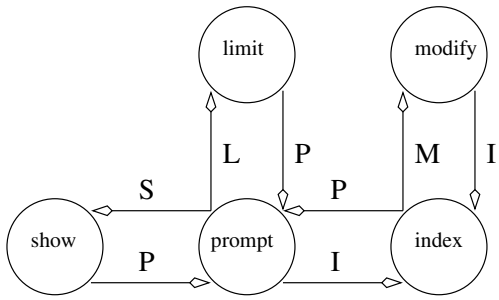


Figure 8.16: A state diagram describing the behavior of the Light Show circuit.

The Light Show circuit changes state when the users presses a key on the keyboard. For example, pressing "M" while in the **index** state causes the circuit to transition into the **modify state**. The behavior of the Light Show circuit in each of its states is described in Table 8.5.

The algorithm for the light show circuit continually scans the busy signal and the keyboard scan code. When the user presses either a "L", "I", or "S", the algorithm drops into one of the subfunctions described in Table 8.5. Assume the Light Show circuit is clocked at 16MHz. The clock rate is needed in order to create a delay of 0.25 seconds required to pace the frames during the show phase. The delay is created by having the circuit wait for a counter to count up from 0 to 4,000,000. The function calls in the algorithm are explained in the subsequent line-by-line analysis.

State	LED bar graph	7-segment	Behavior
reset	blank	blank	All sequential elements are reset
prompt	blank	"P"	Waiting for user input
limit	blank	current limit	The number of frames in the light show is called the limit. If the user enters a hex value between 0-F it is stored as the new limit.
index	current frame	current index	Each frame in the show has an index which defines its position in the show. If the user enters a hex value between 0-F, this frame will be edited in the **modify** state.
modify	current frame	current index	Each frame has eight bits which specify the state of the 8 LEDs on the bar graph. These bits are toggled by pressing the corresponding key. For example, if LED 5 is on, pressing "5", causes LED 5 to go off.
show	cycle through frames	current index	Consecutive frames are displayed on the LED bar graph at around 4Hz. After the last frame is displayed, the circuit loops back to the 0th frame.

Table 8.5: The behavior of the Light Show circuit in each of its states.

```
while(1) {
    HexDisplay = ``P"
    if (!busy and IsL(ScanCode)) {
        while (!busy' and !IsP(ScanCode)) {
           HexDisplay = Hex2Seven(limit);
           BarGraph = 0x00;
           if (!busy and IsHex(ScanCode)) {
                limit = Scan2Hex(ScanCode);
                while(!busy);
    }   }  }

    if (!busy and IsI(ScanCode)) {
        while (busy or !IsP(ScanCode)) {
            HexDisplay = Hex2Seven(index);
            if (!busy and IsHex(ScanCode)) {
               index = Scan2Hex(ScanCode);
                while(!busy);
            }

            if (!busy and IsM(ScanCode)) {
                while (busy or !IsI(ScanCode)) {
```

```
20.                     HexDisplay = Hex2Seven(index);
21.                     BarGraph = RAM[index];
22                         while(!busy);
23                         while(busy);
24.                     if (!busy and IsOct(ScanCode)) {
25.                         RAM[index] = Flip(IsOct(ScanCode),RAM[index]);
26.    }   }   }   }   }

27.     if (!busy and IsS(ScanCode)) {
28.        index = 0;
29.         while(!busy and !IsP(ScanCode)) {
30.             BarGraph = RAM[index];
31.             HexDisplay = Hex2Seven(index);
32.             for (timer=0; timer<2^22; timer++);
33.             index += 1;
34.             if (index == limit) index=0;
35.  }  }   }
```

As each line is parsed, new states are added to the control unit and basic building blocks are added to the datapath. In several cases, brand new building blocks need to be created in order to perform the tasks required of the datapath. Descriptions of these components are provided, but the structure of their internal organization is left to the reader to determine.

Line 1. The infinite loop formed by Line 1 and Line 35 means that the outermost loop in the control unit cycles in the **prompt** state in Figure 8.17 until something happens.

Line 2. From the word statement, the 7-segment display shows "P" or a numerical value. A 2:1 mux in the datapath resolves this conflict. Since the assignment statement is performed without the need to actually assign a value to a register, no explicit state is required. The control unit selects the "P" input when in the **prompt** state.

Line 3. The busy signal is generated by the kbscan component in the datapath. The functional notation "IsL(ScanCode)" is just short-hand for the IsL output from the ScanDecode component in the datapath. The ScanDecode component takes in the 8-bit scan code from the kbscan component and outputs 1 when the scan code corresponds to the scan code for the letter "l". The ANDing of busy' and IsL signals is handled by the internal logic in the control unit. When the condition is true, the control unit transitions to the **set limit** state.

Line 4. The condition of the while statement is checked in the **set limit** state. When the user presses a "p" in the **set limit** state, the control unit transitions back to the **prompt** state. See page 83 for more details regarding the scan code recognizer circuit.

Line 5. Two different numerical values can be displayed on the 7-segment display, limit and index (Line 13). The 2:1 multiplexer in front of the Hex2Seven component allows the control unit to select which of these two values is converted and sent onto the display. The control unit asserts its control outputs to route the limit register's output to the 7-segment display in the **set limit** state.

Line 6. The LED bargraph displays two different values on its output, 0x00 and RAM[index], (Line 21). The 2:1 multiplexer placed in front of the LED bargraph resolves this conflict. The control unit selects 0x00 when it is in the **set limit** state.

Line 7. If the control unit detects a hexadecimal character has arrived while in the **set limit** state, the control unit transitions to the **load limit** state. The datapath already contains the hardware necessary to check this condition.

Line 8. This assignment is performed in the **load limit** state where the control unit asserts a load command to the limit register. The data input to the limit register comes from the converted scan code via the Scan2Hex component. The Scan2Hex component takes in an 8-bit scan code corresponding to a key representing a hexadecimal character and converts it into its 4-bit value. See page 84 for more details.

Line 9. After loading the limit register, the control unit immediately transitions to the **wait limit** state. Then, the control unit waits for some keyboard activity before transitioning back to the **set limit** state. This action is done in order to prevent the limit register from being continuously loaded while waiting for keyboard activity. Such repetitive loading is considered bad form. While waiting in this state the control unit should display the new limit register on the 7-segment display.

Line 10. The If/Then statement started on Line 3 is terminated.

Line 11. This statement causes the control unit to transition to the **set index** state.

Line 12. The while statement returns the control unit transition to the **prompt** state when "p" is pressed.

Line 13. When the control unit is in the **set index** state, the 7-segment display should show the current index by asserting the correct control signals.

Line 14. When the control unit is in the **set index** state and a hexadecimal character is typed, the control unit should transition to the **load index** state.

Line 15. Since the primary purpose of the index is to walk through the memory during a light show, it is stored in a counter. The control unit signals the counter to load the index in the **load index** state.

Line 16. After loading the index register, the control unit transitions to the **wait limit** state where it waits for keyboard activity before transitioning back to the **set index** state.

Line 17. Closes the if/then statement from Line 14.

Line 18. When the control unit is in the **set index** state, a keypress of "m" causes it to transition to the **modify** state.

Line 19. When the control unit is in the **modify** state, a keypress of "i" causes it to transition to the **set index** state. Otherwise the control unit transitions to the **read** state.

Line 20. When in the **read** state, the control unit should route the current index to the 7-segment display.

Line 21. When in the **read** state the control unit instructs the bargraph register to load the current frame from the RAM.

Line 22. After reading the current frame, the control unit transitions to the **wait! busy** state where it displays the frame on the LED bargraph and waits for a keypress event.

Line 23. The control unit waits for a key press event in order to leave the **wait busy** state.

Line 24. If the keypress event was an octal digit, the control unit transitions to the **write** state, otherwise it goes to the **modify** state to check if the keypress was an "i".

Line 25. In the **write** state, one of the current frame's bits are flipped and then the altered frame is stored back into the RAM. The lower three bits of the converted scan code tell the flip component which bit of the frame to invert. Refer to page 83 for further details regarding the flip component. After this, the control unit goes to the **read** state which started the while loop on Line 19.

Line 26. The preceding control structures are terminated, ending with the if/then on Line 11.

Line 27. This statement causes the control unit to transition to the **reset index** state.

Line 28. When in the **reset index** state, the control unit instructs the index counter to synchronously reset its value to 0.

Line 29. Pressing "p" stops the light show and returns the control unit to the **prompt** state. This check is performed during the first state, **load frame**, of the while loop.

Line 30. The bargraph register is loaded with the current frame during the **load frame** state.

Line 31. The control unit should display the current index on the 7-segment display during all the states which make up the while loop of the light show. In preparation for the loop in the next line, the timer counter is reset to 0 in the **load frame** state.

Line 32. In the **wait frame** state, the control unit increments the delay counter from 0 to 4,000,000. The comparator on the delay counters output signals to the control unit when this happens. Since the clock is running at 16MHz, this count value will hold the control unit in the **wait frame** state for 1/4 of a second.

Line 33. When in the **inc index** state, the control unit increments the index of the current frame.

Line 34. When the control unit is in the **comp index** state, it checks the index equals the limit. If it does, the control unit transitions to the **reset index** state, otherwise the next frame is loaded in the **load frame** state.

Line 35. The program is closed.

The net accumulation of the states and hardware added in the parsing of the Light Show algorithm are shown in Figure 8.17.

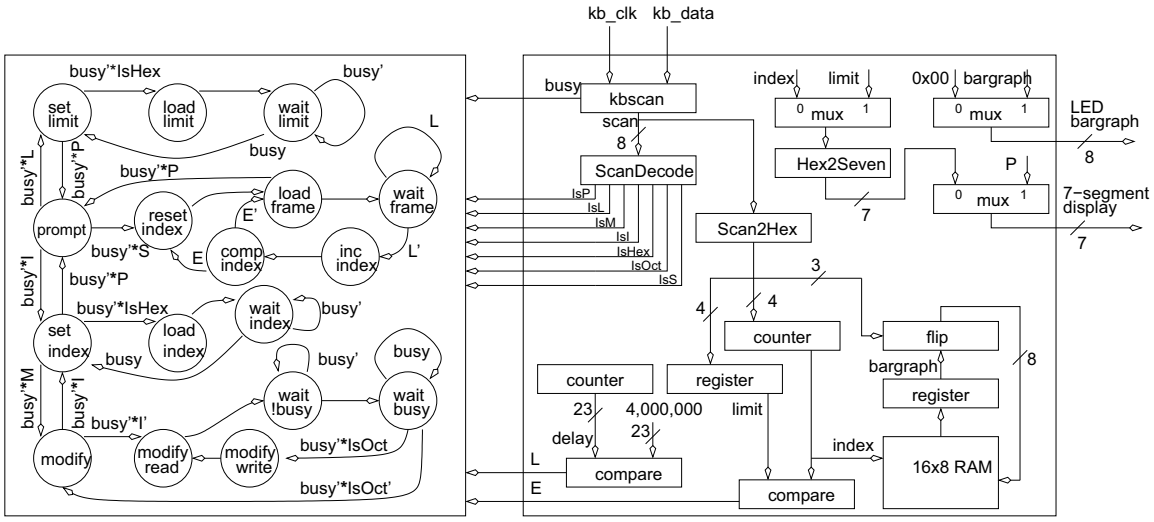


Figure 8.17: The datapath and control for the Light show circuit.

The next step in the design of Light Show circuit is to define the control word and its value for each of the states shown in Figure 8.17. The control word is the set of signals from the control unit to the control inputs of the components in the datapath. The top row of Table 8.6 lists all the components in the datapath which have control inputs. The resulting 11-bit control word is then defined for each of the 17 states. The actions performed in each state are determined from the line-by-line analysis of the light show algorithm.

The reader is left to derive the MIEs and OEs for the control unit.

State	bar mux	7-seg mux	hex mux	index count	delay count	limit register	bargraph register	enb	wen	flip
	0 0x00 1 bar	0 index 1 limit	0 7-seg 1 “P”	00 hold 01 cnt 10 load 11 reset	00 hold 01 cnt 10 load 11 reset	0 hold 1 load	0 hold 1 load	0 1 read	0 write	0 pass 1 flip
prompt	0	x	1	00	00	0	0	0	0	x
set limit	0	1	0	00	00	0	0	0	0	x
load limit	0	1	0	00	00	1	0	0	0	x
waitlimit	0	1	0	00	00	0	0	0	0	x
reset index	1	0	0	11	00	0	0	0	0	x
load frame	1	0	0	00	11	0	1	1	0	x
wait frame	1	0	0	00	01	0	0	0	0	x
inc index	1	0	0	01	00	0	0	0	0	x
comp index	1	0	0	00	00	0	0	0	0	x
set index	0	0	0	00	00	0	0	0	0	x
load index	0	0	0	10	00	0	0	0	0	x
wait index	0	0	0	00	00	0	0	0	0	x
modify	0	0	0	00	00	0	0	0	0	x
modify read	1	0	0	00	00	0	1	1	0	x
modify write	1	0	0	00	00	0	0	1	1	1
wait !busy	1	0	0	00	00	0	0	0	0	x
wait busy	1	0	0	00	00	0	0	0	0	x

Table 8.6: The control word for the LightShow circuit and its value for each state.

8.8 Exercises

1. **(4 pts.)** Show how to eliminate the 4-bit 2:1 mux in the bit counter by assuming that the Y register had an asynchronous active low reset input. Consider the fact that the external world still needs the ability to hit a single button to reset the state of the entire circuit.

2. **(6 pts.)** A control unit has been built with the following control word:

Reg A	Reg B	Reg P	Mux M
00 hold	00 hold	1 hold	1 Load 0
11 lsr	11 lsr		
10 lsl	10 lsl	0 load	0 Load Add
01 load	01 load		

   Regrettably, these setting were completely wrong. In reality here is what the control word should have been:

Reg A	Reg B	Reg P	Mux M
00 hold	00 hold	0 hold	0 Load 0
01 lsr	01 lsr		
10 lsl	10 lsl	1 load	1 Load Add
11 load	11 load		

   The design team is in a total panic. The design team thinks that it will take weeks to straighten out the error, they claim that the control unit needs to be redesigned. However, there is a cheap and easy solution. Design some combinational logic to insert between the faulty control unit and the datapath in order to straighten out the bum control signals. There is one error can be fixed by changing something in the datapath, no extra hardware is required. Identify this error and its solution.

3. **(8 pts.)** Modify the algorithm for the bit counting circuit so that it uses a two-line handshake to transmit the Y register. The circuit should take the role of an active producer in the transmission of Y. The circuit has four handshaking lines and two data lines. Hint, a common error of students is to insert a three-state buffer on the output of the Y register to the outside world to prevent its transmission to the outside world until the value of Y is finalized. Don't do this! If the outside world reads the value of Y before the circuit's signals are valid (via the send_REQ signal) then its their own dumb fault. Just send the Y signal outside the datapath as is.

4. **(16 pts.)** A 8kx32 RAM is full of integer data. Design a circuit to scan the RAM and find its smallest value.

   Turn in; an algorithm the datapath and control unit, the control word table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding.

5. **(16 pts.)** A 8kx32 RAM is full of integer data. Design a circuit that determines the sum of the integers *between* addresses A and B. The values of A and B are to be read in using a two-line handshake where the circuit is to act as a passive consumer. The sum is to be placed in a 32-bit register S. Turn in; an algorithm the datapath and control unit, the control word table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding.

6. **(16 pts.)** Design a circuit that repetitively looks at a 1-bit input X. Anytime X changes logic values increment an 8-bit register Y. Turn in; an algorithm the datapath and control unit, the control word table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding.

7. **(16 pts.)** A 256x8 RAM is full of data. Design a circuit that jumps around in memory. It does this by fetching a word and using the retrieved word as the next address to jump to. The circuit is to start at address 0. Turn in; an algorithm the datapath and control unit, the control word table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding.

   To desired behavior of the circuit is illustrates in Figure 8.18. If the address=0 then the circuit will jump to address 3F then 28, 53, 3F and continue cycling for ever amount these three addresses.

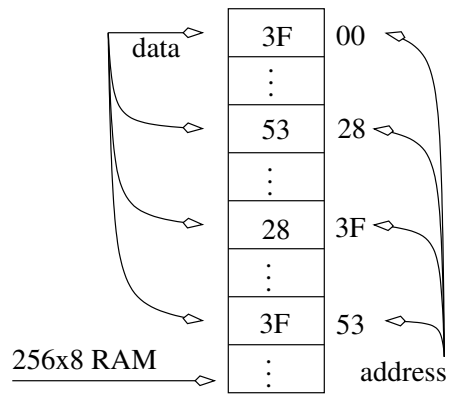


Figure 8.18: A 256x8 RAM loaded with some data.

8. **(16 pts.)** A 256x8 RAM is full of data. Design a circuit that jumps around in memory. The current address should be stored in a register called PC. If the MSB of the fetched word is 1, then the remaining seven bits represent a 7-bit 2's complement number; add these seven bits to the PC. If the MSB of the fetched word is 0 then just increment the PC. Repeat this process forever. Turn in; an algorithm the datapath and control unit, the control word table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding.

   The desired behavior of the circuit is illustrated in Figure 8.19. In this figure if PC=0 then the word at that address (3F) has a MSB of 0 so the PC is incremented to 1. The word at address 1 is fetched (BC) and has an MSB of 1 so the least significant seven bits of BC are added to the PC, making its new value 3D. repeating this process sees the PC goto address 21, 22, 21, 22 into a never ending cycle. Make sure the solution identifies how to add the least significant seven bits to an 8-bit PC.

9. **(16 pts.)** Modify the circuit in the previous problem as follows. Anytime an external input, called IRQ, is asserted the circuit is to stop jumping around and assert and ACK. The outside world will then read the PC (which must be routed outside the datapath) and then drop the IRQ. The circuit should then drop the ACK and resume jumping. Turn in; an algorithm the datapath and control unit, the control word table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding.

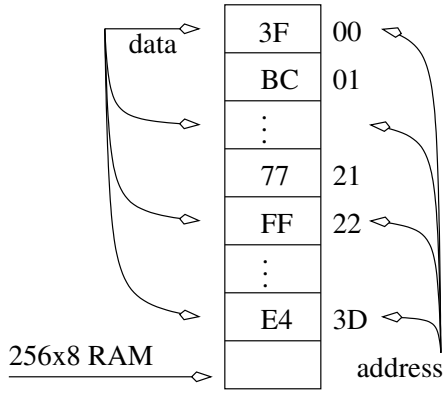


Figure 8.19: A 256x8 RAM loaded with some data.

10. **(16 pts.)** Design a circuit that reads successive words from a 1kx12 RAM and updates a 12-bit register called **ACC** based on the upper two bits of the memory word. The address of the current memory word should be contained in a register called PC (Program Counter). Since the words read from the RAM will tell us what operation to perform on the ACC, the memory word will be stored in a register called IR (Instruction Register). If the upper two bits of IR are:

    a) 00 then add the lower 10 bits of the IR to ACC. Pad the upper two bits of the IR with 0's before adding to the ACC.

    b) 01 then store the ACC to the address specified by the lower 10 bits of the IR.

    c) 10 then load the ACC from from the address specified by the lower 10 bits of the IR.

    d) 11 then clear the value of ACC to 0.

    The PC is to be initialized to 0. After the each memory word is read and the appropriate operation performed on ACC, the PC should be incremented. Turn in; an algorithm the datapath and control unit, the control word table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding.

11. **(16 pts.)** Design a circuit that moves M consecutive words from address S (source) to address D (destination). For example, if $M = 4$, $S = 3EA$ and $D = 1FE$ then the circuit would move words 3EA, 3EB, 3EC and 3ED to address 1FE, 1FF, 200 and 201. Each of M, S, D is preloaded into a register. While this problem appears simple, its really rather treacherous. The circuit will have to handle cases where $S + M > D$. In such a case the order of the data movement must be carefully planned. In order to simplify the design, assume that $S < D$. Turn in; an algorithm the datapath and control unit, the control word table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding. Do not worry about the sizes of the registers or RAM.

12. **(16 pts.)** Design a circuit that determines how many times a user specified 8-bit value, called **key**, occurs in an 1kx8 RAM. **key** is to be read using a two-line handshake; the circuit is the passive consumer. Turn in; an algorithm the datapath and control unit, the control word table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding.

13. **(16 pts.)** Design a circuit that records the number of times that it has seen an 8-bit, user specified value, **key**. The key will is shown to the circuit, at most, 16 times. The collection of keys is stored in a 1kx12 RAM. The RAM is larger than it needs to be because it is thought that in the future the number of keys will be increased. Each word of the RAM is organized as follows; The upper eight bits hold the key and the lower four bits hold the "hit count", the number of times that this key has been seen. The circuit should read in the key using a two-line handshake; the circuit is the passive consumer. The circuit should then scan the RAM looking for a matching key; a match, if it exists, will only occur once in the RAM. If a match is found then increment the lower four bit and store the key and the incremented hit count back to RAM. Turn in; an algorithm the datapath and control unit, the control word table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding.

14. **(36 pts.)** Design a digital circuit to control access to an automated parking garage containing 828 parking spaces. Drivers pull up to the garages gate and insert their pass card into a card reader. The card reader sends the pass card ID number to the digital circuit. If their pass card has a valid code then the gate opens. There is a pressure sensor just inside the entry way which sends a signal to the circuit whenever a significant load is present (over 150 lbs). The exit procedure is similar, the users have to insert their pass card into a card reader. The digital circuit then raises the exit gate bar, a pressure sensor at the exit tells the circuit when it is OK to close the exit gate. See Figure 8.20.

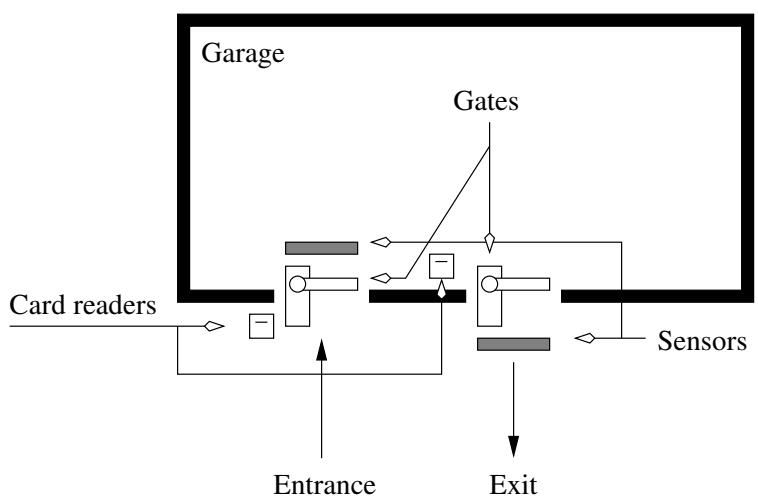


Figure 8.20: The layout of an automated garage.

The signal names are defined in the following table:

Entrance gate	InGate	0 Close gate	1 Open gate
Entrance sensor	InSen	0 No weight	1 Weight present
Entrance REQ	InREQ	0 No card read data	1 Card reader has data
Entrance ACK	InACK	Circuit control	
Entrance ID	InID	Card ID	
Exit gate	OutGate	0 Close gate	1 Open gate
Exit sensor	OutSen	0 No weight	1 Weight present
Exit REQ	OutREQ	0 No card read data	1 Card reader has data
Exit ACK	OutACK	Circuit control	
Exit ID	OutID	Card ID	

The gate requires a logic 1 to start and to stay open. The sensor will generate a logic 1 while there is more than 150 lbs. on the sensor. Only close the gate when the rear wheels of the car activate the sensor (hope no unicycle use the garage). The entrance card reader will provide InID or OutID using a two-line handshake, where the circuit is the passive consumer. Assume that at any point in time only one car is entering or leaving the garage. That is, deal with only one direction at a time.

In addition to controlling access to the garage, the clients would also like to keep track track of how many times a pass ID has been used to gain access in-to and out-of the garage. The count will be checked and reset once a month. Cars pass into and out of the garage at most 4 times a day.

To implement this circuit use a *single* RAM. Each word of the RAM must be divided into three fields; ID, Ins and Outs corresponding to the pass ID number, number of times into the garage and number of times out of the garage respectively. The digital circuit will scan successive IDs in the RAM looking for a match. If a match is found then either increment the Ins or Outs field then store this item back into the RAM. A major issue in this design is determining the sizes of the data items. Use the information in the word statement to make the design as space efficient as possible. Turn in; an algorithm

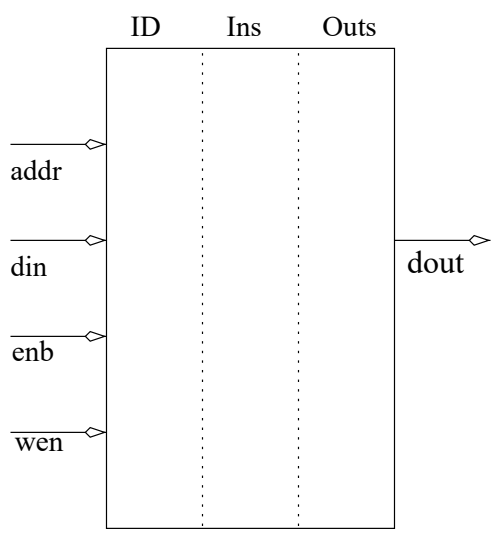


Figure 8.21: The format of the RAM in the garage circuit problem.

the datapath and control unit, the control word table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding.

15. **(16 pts.)** Design a circuit that converts a 6-bit binary number into a 2 digit BCD representation. The circuit acquires a 6-bit number through a two-line handshake where the circuit is a passive consumer. The circuit is then to convert this 6-bit number into two BCD digits and signals it completion via a DONE signal.

    A number X can be converted from binary into BCD digits by iteratively checking that X is greater than 10, then subtracting 10 from X. Each subtraction should increment a tens digit counter.

    Make sure to identify the size of all the signals in the datapath and the size of any register, counters, etc... Turn in; an algorithm the datapath and control unit, the control word table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding.

16. **(8 pts.)** Design a circuit that converts a 2 digit BCD number into a binary number. The circuit acquires the BCD digits through 2 read operations most significant digit first. Each read operation takes the form of a two-line handshake where the circuit is a passive consumer.

    A 2 digit BCD number can be converted into binary by multiplying the most significant digit by 10 then adding it to the least significant BCD digit. A number can be multiplied by 10 using the shift-and-add technique presented on page 104. Note, this task can be accomplished without using a single shift register. For example, the adder in Figure 8.22 generates the value of 9*X from a 4-bit register by adding X, shifted left by three bits, to X.

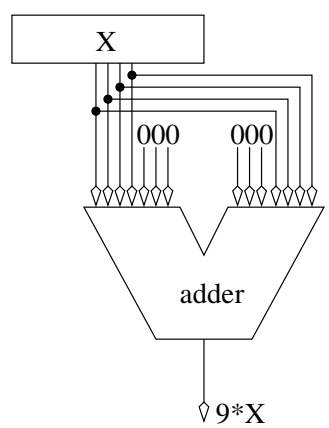


Figure 8.22: A simple circuit to compute 9*X.

    Make sure to identify the size of all the signals in the datapath and the size of any register, counters, etc... Turn in; an algorithm the datapath and control unit, the control word table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding.

17. **(36 pts.)** Design a digital circuit that plays a game of roulette, allows betting and keeps track of total earnings. The roulette wheel has 8 slots, labeled $1 \ldots 8$. The player can play one of the numbers straight or play even or odd. The player starts with \$10. The layout of the machine is shown in Figure 8.23.

    The sequence of events is as follows:

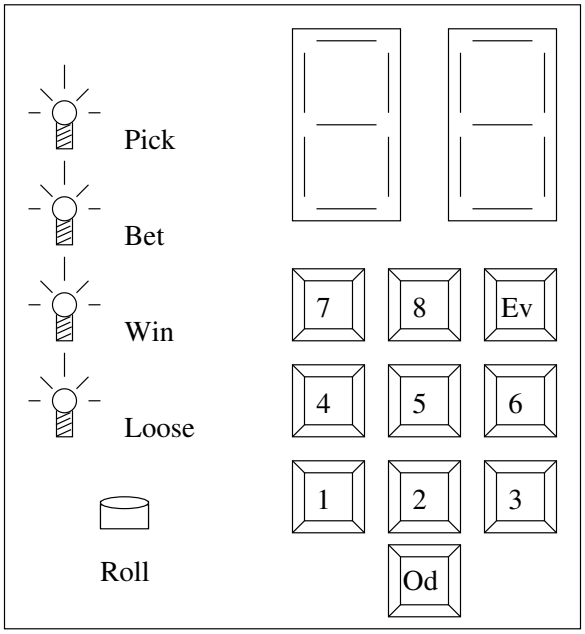


Figure 8.23: The layout of the roulette playing machine. The two 7-segment displays at the top are used for a variety of purposes.

a) The circuit lights up the PICK LED. The player enters their guess; a number between 1-8, even or odd. While holding down their guess they press the roll button.

b) The circuit displays the picked number in the left most 7-segment display. The circuit lights up the BET LED. The player enters a one digit bet between 1 to 8. While holding down their bet they press the roll button.

c) The circuit displays the bet on the rightmost 7-segment display. The player pushes and holds down the roll button. The circuit increments a mod 8 counter while the roll button is depressed. It would be nice to display the current count value on right 7-seven segment display. Since the clock cycle is on the order of milliseconds, then the user would not be able to anticipate the roll.

d) The player releases the roll button. The final roll is displayed on the rightmost 7-segment display. The circuit stops incrementing the counter and checks to see if the final value matches the players guess. If the match is correct then light the WIN LED and increment the players earnings. If the match is incorrect then light the LOOSE LED and decrement the players earnings.

e) The play hits the roll button to clear the roll information from the 7-segment displays.

f) The circuit displays the players earnings on the 7-segment display.

g) When the user pushes the roll button then go to step 1.

Set reasonable bounds on the maximum winnings. Values may be displayed in hexadecimal (assume there is a hex to 7-segment display converter available). See page 79 for more information. Turn in; an algorithm the datapath and control unit, the control word

table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding.

18. **(20 pts.)** Design a tone generator. The tone generator is a box with two buttons on it labeled "Up" and "Down" and a 1-bit output. At start-up the tone generator outputs a 440Hz square wave (clock-like signal). Every time that the Up button is pressed the tone generator should increase the frequency of the square wave by $\sqrt[12]{2} - 1.0 = 0.059463094 \approx 7/128$ of its current frequency. To determine the fraction $7/128$ of X, shift X left by 7-bits (dividing by 128) then multiplying it by 4+2+1. Every time that the down button is pressed the circuit should decrease the frequency by $7/128$ of its current value. Assume that the master clock frequency of the circuit is 4Mhz. Turn in any relevant calculations, algorithm, datapath and control, control word, MIEs, OEs and the maximum tone frequency of the circuit. Turn in; an algorithm the datapath and control unit, the control word table, the memory input equations, and output equations. The control unit is to be implemented using a ones hot encoding.

Index