9. Proposition Logic

9.1 Introduction

This chapter describes proposition logic and some of the roles this form of logic plays in computer hardware and software. From early on, computers have been known as “logic machines”. Indeed, “logic” plays a central role in the design, programming, and use of computers. Generally, “logic” suggests a system for reasoning. But in computer science, reasoning is only one use of logic. We also use logic in a fairly mechanistic way in the basic construction of computers. This form of logic is called “proposition logic”, “switching logic”, or sometimes (not quite correctly) “Boolean algebra”. There are also several other, much more powerful, types of logic that are used in other aspects of computer science. “Predicate logic”, also called “predicate-calculus” or “first-order logic” is used in programming and in databases. Predicate logic, and “temporal logic”, which is built upon predicate logic, are used for reasoning about programs and dynamic systems. Varieties of “modal logic” are used in building artificial intelligence reasoning systems.

In this course, we will be concerned mostly with proposition logic, and to some extent, and mostly informally, predicate logic. Proposition logic is used in the design and construction of computer hardware. It is also related to a simple kind of deductive reasoning. In proposition logic, we deal with relationships among variables over a **two-valued domain**, for these reasons:

- It is simplest to build high-speed calculating devices out of elements that have only two (as opposed to several) stable states, since such devices are the simplest to control. We need only to be able to do two things:

    - sense the current state of the device

    - set the state of the device to either of the two values

- All finite sets of values (such as the control states of a Turing machine or other computer) can be *encoded* as combinations (“tuples”) of two-valued variables.

In this book we shall mostly use 0 and 1 as our two values, although any set of two values, such as true and false, yes and no, *a* and *b*, red and black, would do. When we need to think in terms of truth, we will usually use 1 for true and 0 for false.

9.2 Encodings of Finite Sets

As mentioned above, every finite set can be encoded into tuples of 0 and 1. These symbols are usually called "bits". Technically the word "bit" stands for "binary digit", but it is common to use this term even when we don't have a binary or base-2 numeral system in mind. More precisely, an **encoding** of a set S is a one-to-one function of the form

$$S \rightarrow \{0, 1\}^N$$

for some N. Here $\{0, 1\}^N$ means the set of all N-tuples of 0 and 1. A given tuple in this context is called a "codeword". Remember that "one-to-one" means that no two elements of S map to the same value. This is necessary so that a codeword can be decoded unambiguously. Since $\{0, 1\}^N$ has exactly 2^N elements, in order for the one-to-one property to hold, N would therefore have to be such that

$$2^N \geq |S|$$

where we recall that |S| means the **size**, or number of elements, of S. Put another way, N is an integer such that

$$N \geq \log_2 |S|.$$

This inequality still gives us plenty of leeway in choosing encodings. There are many considerations that come into play when considering an encoding, and this motivates the use of different encodings in different situations. A fair amount of programming ends up being conversion of one encoding to another. Some of the considerations involved in the choice are:

- *Conciseness*: a code that uses as few symbols as possible.

- *Ease in decoding*: a code that is simple for humans, or circuitry, to decode.

- *Difficulty in decoding*: a code that is hard to decode, for example, one to be used in encrypting data for security purposes.

- *Error detection*: a code designed in such a way that if one of the bits changes inadvertently, this fact can be detected.

- *Error correction*: like error detection, except that we can determine the original value, as well as determining whether a change occurred.

- Other special properties, such as the "one change" property found in Gray codes, as discussed below.

We already encountered the binary numerals in our earlier discussion of encoding the infinite set of natural numbers. Obviously, the same idea can be used to encode a finite

set. It is common to use a fixed number of bits and include all of the leading zeroes when encoding a finite set.

Binary Code Example

Encode the set {0, 1, 2, 3, 4, 5, 6, 7} in binary: We give the encoding by a table showing the correspondence between elements of the set and $\{0, 1\}^3$, the set of all 3-tuples of bits:

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Figure 121: The binary encoding of {0, ..., 7}

It is easy to see that this is a code, i.e. that the correspondence is one-to-one.

In general, N bits is adequate to encode the set of numbers $\{0, \ldots, 2^N-1\}$. There are many other ways to encode this same set with the same number of bits. Not all have any particular pattern to them. One that does is the following:

Gray Code Example

A Gray Code is also called "reflected binary". Encoding the same set as above, this code starts like binary:

0	000
1	001

However, once we get to 2, we change the second bit from the right only:

1	001	
2	011	(instead of 010 as in straight binary)

In general, we wish to change only one bit in moving from the encoding from a number K to K+1. The trick is to do this without returning to 000 prematurely. By using the pattern of "reflecting" a certain number of bits on the right, we can achieve coverage of all bit patterns while maintaining the one-change property.

```
0    000
1    001 ↑
       ---    last bit reflected above and below
2    011 ↓ read up above the line, copy bit downward
3    010
     -----    last two bits reflected
4    110
5    111
6    101
7    100
0    000
```

Figure 122: The Gray encoding of {0, ..., 7}

One-Hot Code Example

This code is far from using the fewest bits, but is very easy to decode. To encode a set of N elements, it uses N bits, only one of which is 1 in any codeword. Thus, to encode {0, ..., 5}, a one-hot code is:

0	000001
1	000010
2	000100
3	001000
4	010000
5	100000

Figure 123: A one-hot encoding of {0, ..., 5}

Examples of one-hot codes include a push-button telephone (one out of 12) and a standard traffic light (one out of 3). An electronic piano keyboard would not be one-hot, because it allows chords to be struck.

Subset Code Example

This code is useful when the set to be encoded is, in fact, the set of all subsets of a given set. If the latter has N elements, then exactly 2^N elements are in the set of subsets. This suggests an N-bit code, where one bit is used to represent the presence of each distinct element. For example, if the set is {a, b, c}, then the set of all subsets is {{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}. The code would appear as:

{}	000
{a}	100
{b}	010
{c}	001
{a, b}	110
{a, c}	101
{b, c}	011
{a, b, c}	111

Figure 124: An encoding of all subsets of {a, b, c}

Note that we could use this code and the binary code to establish a relationship between subsets and the numbers {0, ..., 2^N-1}:

number	subset	binary
0	{}	000
1	{c}	001
2	{b}	010
3	{b, c}	011
4	{a}	100
5	{a, c}	101
6	{a, b}	110
7	{a, b, c}	111

Figure 125: Correspondence between binary encoding subset encoding

This consideration is of importance in the construction of computers and also in programming, as will be seen later.

The Pascal language has a built-in subset code feature in the form of a *set* type, which can be derived from any enumeration type.

Binary-Coded Decimal Example

This code, abbreviated BCD, is sometimes used to make it easy to decode into decimal representations. The number to be encoded is first represented in decimal (radix 10). Each digit is separately coded in 4-bit binary. The resulting 4-bit codes are concatenated to get the codeword. For example, the number 497 would encode as 0100 1001 0111.

The Cartesian Encoding Principle

The BCD encoding illustrates a general principle: We can achieve economy in the description of an encoding when we can decompose the set to be encoded as a Cartesian Product of smaller sets. In this case, we can separately encode each set, then take the

overall code to be a tuple of the individual codes. Suppose that the set S to be encoded contains M elements, where M is fairly large. Without further decomposition, it would take a table of M entries to show the encoding. Suppose that S can be represented as the Cartesian product T x U, where T contains N and U contains P elements. Then M = NP. Knowing that we are using a product encoding, we need only give two tables, one of size N and the other of size P, respectively, to present the encoding. For example, if N and P are roughly the same, the table we have to give is on the order of 2 times the square root of M, rather than M. For large M this can be a substantial saving.

Error-Correcting Code Principle (Advanced)

Richard W. Hamming invented a technique that provides one basis for a family of error-correcting codes. Any finite set of data can be encoded by adding sufficiently many additional bits to handle the error correction. Moreover, the number of error correction bits added grows as the logarithm of the number of data bits.

The underlying principle can be viewed as the multiple application of *parity bits*.

> With the parity bit scheme, a single bit is attached to the transmitted data bits so that the sum modulo 2 (i.e. the exclusive-or) of all bits is always 0. In this way, the corruption of any single bit, *including the parity bit itself*, can be detected. If there are N other bits, then the parity bit is computed as $b_1 \oplus b_2 \oplus .... \oplus b_N$, where $\oplus$ indicates modulo-2 addition (defined by $0 \oplus 1 = 1 \oplus 0 = 1$, and $0 \oplus 0 = 1 \oplus 1 = 0$). If the sum of the bits is required to be 0, this is called "even parity". If it is required to be 1, it is called "odd parity".

The Hamming Code extends the parity error- detection principle to provide single-bit error *correction* as follows: Designate the ultimate codewords as $\{c_0, c_1, c_2, ..., c_K\}$. (We haven't said precisely what they are yet.) Suppose that N is the number of bits used in the encoding. Number the bits of a generic codeword as b_1, b_2, b_3, ... The code is to be designed such that the sum of various sets of bits of each codeword is always 0. In particular, for each appropriate i, the sum of all bits having 1 as the ith bit of their binary expansion will be 0 in a proper codeword. In symbolic terms, supposing that there are 7 bits in each codeword, the code requires the following even parities:

$$b_1 \oplus b_3 \oplus b_5 \oplus b_7 = 0$$
$$b_2 \oplus b_3 \oplus b_6 \oplus b_7 = 0$$
$$b_4 \oplus b_5 \oplus b_6 \oplus b_7 = 0$$
....

There is a simple way to guarantee that these properties hold for the code words: Reserve bits b_1, b_2, b_4, b_8, etc. as parity bits, leaving the others b_3, b_5, b_6, b_7, ... for the actual data. Observe that each equation above entails only one parity bit. Hence each equation can be *solved* for that bit, thus determining the parity bits in terms of the data bits:

$b_1 = b_3 \oplus b_5 \oplus b_7$
$b_2 = b_3 \oplus b_6 \oplus b_7$
$b_4 = b_5 \oplus b_6 \oplus b_7$
....

To construct a Hamming code for M data bits, we would begin allocating the bits between parity (the powers of 2) and data, until M data bits were covered. In particular, the highest order bit can be a data bit. For example, if M = 4 data bits were desired, we would allocate b_1 as parity, b_2 as parity, b_3 as data, b_4 as parity, b_5 as data, b_6 as data, and b_7 as data. The index numbered 7 is the least index that gives us 4 data bits. We construct the code by filling in the data bits according to the ordinary binary encoding, then determining the parity bits by the equations above. The result for M = 4 is shown in the table below. Note that it takes two bits of parity to provide error-correcting support for the first bit of data, but just one more bit of parity to provide support for three more bits of data.

		data	**data**	**data**	**parity**	**data**	**parity**	**parity**
decimal	**binary**	**b_7**	**b_6**	**b_5**	b_4	**b_3**	b_2	b_1
0	0000	**0**	**0**	**0**	0	**0**	0	0
1	0001	**0**	**0**	**0**	0	**1**	1	1
2	0010	**0**	**0**	**1**	1	**0**	0	1
3	0011	**0**	**0**	**1**	1	**1**	1	0
4	0100	**0**	**1**	**0**	1	**0**	1	0
5	0101	**0**	**1**	**0**	1	**1**	0	1
6	0110	**0**	**1**	**1**	0	**0**	1	1
7	0111	**0**	**1**	**1**	0	**1**	0	0
8	1000	**1**	**0**	**0**	1	**0**	1	1
9	1001	**1**	**0**	**0**	1	**1**	0	0
10	1010	**1**	**0**	**1**	0	**0**	1	0
11	1011	**1**	**0**	**1**	0	**1**	0	1
12	1100	**1**	**1**	**0**	0	**0**	0	1
13	1101	**1**	**1**	**0**	0	**1**	1	0
14	1110	**1**	**1**	**1**	1	**0**	0	0
15	1111	**1**	**1**	**1**	1	**1**	1	1

Figure 126: The Hamming code for 4 bits of data, requiring a total of 7 bits. The bold-face bits represent data bits. The plain-face bits are determined from the data bits by the parity equations.

For example, consider row 14. The data bits are 1110 = $b_3\ b_5\ b_6\ b_7$. According to our equations,

$b_1 = b_3 \oplus b_5 \oplus b_7 \quad = 0 \oplus 1 \oplus 1 \quad = 0$
$b_2 = b_3 \oplus b_6 \oplus b_7 \quad = 0 \oplus 1 \oplus 1 \quad = 0$
$b_4 = b_5 \oplus b_6 \oplus b_7 \quad = 1 \oplus 1 \oplus 1 \quad = 1$

Thus we have assigned in row 14 $b_4b_2b_1$ = 1 0 0.

Error-correction rule: A word in the Hamming code is error-free (i.e. is a code-word) iff each parity equation holds. Thus, given the word received, compute the sums

$b_1 \oplus b_3 \oplus b_5 \oplus b_7 = s_0$
$b_2 \oplus b_3 \oplus b_6 \oplus b_7 = s_1$
$b_4 \oplus b_5 \oplus b_6 \oplus b_7 = s_2$
....

If any of these is non-zero, then there is an error. A clever part of Hamming's design is that the sums $s_2s_1s_0$, when interpreted as a binary numeral, **indexes the bit that is incorrect**. For example, consider the codeword for 12:

1 **1** **0** 0 **0** 0 1

Suppose that bit 2 gets changed, resulting in:

1 **1** **0** 0 **0** 1 1

Then $s_2s_1s_0$ will be 0 1 0, indicating b_2 is incorrect. On the other hand, if bit 6 were to change, resulting in

1 **0** **0** 0 **0** 0 1

$s_2s_1s_0$ will be 1 1 0, indicating b_6 is incorrect.

The total number of encodings possible with 7 bits is $2^7 = 128$. For each 7-bit code, we need 7 other codes that translate to the same data word, i.e. 8 codes per group. Fortunately $8 * 16 \leq 128$.

We can visualize what is going with Hamming codes by using a hypercube, a recurrent theme in computer science. To do so, we will use a smaller example. Below is shown a 3-dimensional hypercube. The connections on the hypercube are between points that differ by only one bit-change, Hamming distance 1, as we say. To make an error-correcting code, we need to make sure that no code differs by one-bit change from more than one other code. The diagram shows an error-correcting code for one data bit. The dark nodes 000 and 111 are representative code words for 0 and 1 respectively. The nodes with arrows leaving encode the same data as the nodes to which they point. In order for this to be error correcting, each node must be pointed to by all the nodes Hamming distance1 away, and no node can point to more than one representative.

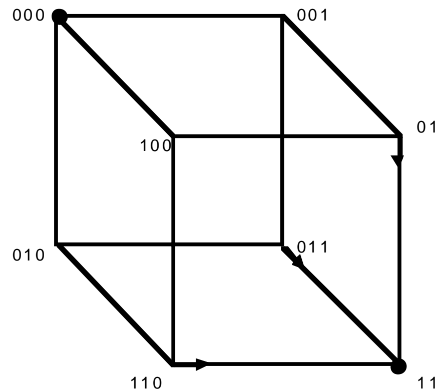


Figure 127: Three-dimensional hypercube for an error-correcting code

The above code is "efficient" in the sense that no node goes unused. If we move to a four-dimensional hypercube, we could try to encode more than one data bit, for example two data bits. A feasibility test for this encoding is that there must be five code words in each group: the main representative and its four immediate neighbors (to account for all 1-bit changes). Also, there are four different combinations of two data bits, so we need at least 5 * 4 == 20 distinct code words. But there are only 16 code words in a four-dimensional hypercube, so with it we cannot do better than one data bit, the same as with a three-dimensional hypercube. We need a 5-dimensional hypercube to achieve error-correction with 2 data bits.

9.3 Encodings in Computer Synthesis

Many problems in synthesis of digital systems involve implementing functions on finite sets. The relationship of this to proposition logic, discussed in the following section, is the following:

> **Combinational Switching Principle**
>
> Once an encoding into bits has been selected for the finite sets of interest, the implementation of functions on those sets reduces to the implementation of functions on bits.

Addition Modulo 3 Example

Suppose we wish to represent the function "addition modulo 3" in terms of functions on bits. Below is the definition table for this function.

	a + b mod 3	b 0	1	2
	0	0	1	2
a	**1**	1	2	0
	2	2	0	1

We select an encoding for the set {0, 1, 2}. Let us choose the binary encoding as an example:

element	encoded element
0	00
1	01
2	10

We then transcribe the original table by replacing each element with its encoding, to get an image of the table using bit encodings, calling the encoding of *a* uv, and the encoding of *b* wx:

		wx 0 0	0 1	1 0
	0 0	0 0	0 1	1 0
uv	**0 1**	0 1	1 0	0 0
	1 0	1 0	0 0	0 1

We then separate this table into two functions, one for each resulting bit value, where we use [u, v] for the bit encodings of *a* and [w, x] for the bit encodings of *b*.

		wx 0 0	0 1	1 0
	0 0	0 0	0 1	1 0
uv	**0 1**	0 1	1 0	0 0
	1 0	1 0	0 0	0 1
		^	^	^

Carats point to "first result bits", used to construct following table.

	f_1	wx 0 0	0 1	1 0
	0 0	0	0	1
uv	0 1	0	1	0
	1 0	1	0	0

Table for the first result bit of encoded modulo 3 addition.

	f2	wx 00	01	10
	00	0	1	0
uv	01	1	0	0
	10	0	0	1

Table for the second result bit of encoded modulo 3 addition.

Each table is then a function on 4 bits, two from the side stub and two from the upper stub, i.e. we have modeled the original function of the form $\{0, 1, 2\}^2 \rightarrow \{0, 1, 2\}$ as two functions of the form $\{0, 1\}^4 \rightarrow \{0, 1\}$. We can compute a + b mod 3 in this encoding by converting a and b to binary, using the two tables to find the first and second bits of the result, then convert the result back to the domain $\{0, 1, 2\}$.

Let's try to model this process in rex. The encoded domain will be represented as lists of two elements, each 0 or 1. We will give a specification for the following functions:

encode: $\{0, 1, 2\} \rightarrow \{0, 1\}^2$ is the encoding of the original domain in binary

add: $\{0, 1, 2\}^2 \rightarrow \{0, 1, 2\}$ is the original mod 3 addition

f_1: $\{0, 1\}^4 \rightarrow \{0, 1\}$ is the function for the first bit of the result

f_2: $\{0, 1\}^4 \rightarrow \{0, 1\}$ is the function for the second bit the result

We expect the following relation to hold, for every value of *a* and *b* in $\{0, 1, 2\}$:

encode(add(a, b)) == [f_1(append(encode(a), encode(b))), f_2(append(encode(a), encode(b)))];

We can write a program that checks this. The rex rules for the above relations are:

```
add(0, 0) => 0;    add(0, 1) => 1;    add(0, 2) => 2;
add(1, 0) => 1;    add(1, 1) => 2;    add(1, 2) => 0;
add(2, 0) => 2;    add(2, 1) => 0;    add(2, 2) => 1;

encode(0) => [0, 0];        encode(1) => [0, 1];        encode(2) => [1, 0];

f1([0, 0, 0, 0]) => 0;      f1([0, 0, 0, 1]) => 0;      f1([0, 0, 1, 0]) => 1;
f1([0, 1, 0, 0]) => 0;      f1([0, 1, 0, 1]) => 1;      f1([0, 1, 1, 0]) => 0;
f1([1, 0, 0, 0]) => 1;      f1([1, 0, 0, 1]) => 0;      f1([1, 0, 1, 0]) => 0;

f2([0, 0, 0, 0]) => 0;      f2([0, 0, 0, 1]) => 1;      f2([0, 0, 1, 0]) => 0;
f2([0, 1, 0, 0]) => 1;      f2([0, 1, 0, 1]) => 0;      f2([0, 1, 1, 0]) => 0;
f2([1, 0, 0, 0]) => 0;      f2([1, 0, 0, 1]) => 0;      f2([1, 0, 1, 0]) => 1;
```

The test program could be:

```
test(A , B ) =>
    encode(add(A , B )) == [f1(append(encode(A ), encode(B ))), f2(append(encode(A ), encode(B )))];
```

```
test_all(_) =>
        (test(0, 0), test(0, 1), test(0, 2),
         test(1, 0), test(1, 1), test(1, 2),
         test(2, 0), test(2, 1), test(2, 2)),
        "test succeeded";

test_all() => "test failed";
```

The first rule for *test_all* has one large guard. If any test fails, then that rule is inapplicable and we use the second rule.

Although the above method of implementing modulo 3 addition is not one we would use in our everyday programming, it is used routinely in design of digital circuits. The area of proposition logic is heavily used in constructing functions on bits, such as f_1 and f_2 above, out of more primitive elements. We turn our attention to this area next.

Exercises

1 •• Suppose that we chose a different encoding for {0, 1, 2}: $0 \rightarrow 00$, $1 \rightarrow 10$, $2 \rightarrow 11$. Construct the corresponding bit functions f_1 and f_2 for modulo-3 addition.

2 •• Choose an encoding and derive the corresponding bit functions for the *less_than* function on the set {0, 1, 2, 3}.

3 •• If A is a finite set, use |A| to denote the *size* of A, i.e. its number of elements (also called the *cardinality* of A). Express |A x B| in terms of |A| and |B|.

4 •• Express $|A_1 \times A_2 \times .... \times A_N|$ in terms of the N quantities $|A_1|$, $|A_2|$, ... $|A_N|$.

5 ••• Let A^B denote the *set of all functions* with B as domain and A as co-domain. Supposing A and B are finite, express $|A^B|$ in terms of |A| and |B|.

6 • What is the fewest number of bits required to encode the English alphabet, assuming that we use only lower-case letters? What if we used both lower and upper case? What if we also included the digits 0 through 9?

7 ••• For large values of N, how does the number of bits required to encode an N element set in binary-coded decimal compare to the number of bits required for binary?

8 •• Show that a Gray code can be constructed for any set of size 2^N. [Hint: Use induction.]

9 ••• Devise a rex program for the function *gray* that, given N > 0, will output a Gray code on N bits. For example, gray(3) ==> [[0,0,0], [0,0,1], [0,1,1], [0,1,0], [1,1,0], [1,1,1], [1,0,1], [1,0,0]].

10 ••• Devise a rex program that will "count" in Gray code, in the sense that given a codeword in the form of a list, it will produce the next codeword in sequence. For example, gray_count([1,1,1]) ==> [1,0,1], *etc*. [Hint: Review the Chinese ring puzzle in *Compute by the Rules*.]

9.4 Propositions

By a "proposition" we mean a statement or condition that can be one of 0 ("false") or 1 ("true"). In computer science, there are two primary ways in which we deal with propositions:

- Expressions that contain proposition variables and logical operators
- Functions, the arguments of which range over proposition values.

These two ways are closely related, but sometimes it is more natural to work with expressions while other times it is simpler to work with functions.

Examples

Let us give some variables representing propositions:

a: Harvey Mudd College is in Claremont.

b: Disneyland is in Claremont.

c: It is raining in Claremont.

Each of these statements can be assigned a truth value, 0 or 1. It turns out that it only makes sense to assign *a* the value 1 and *b* the value 0, but this is irrelevant since proposition logic is concerned mostly with relationships between *hypothetical* truth values. These relationships are expressed by propositional operations or functions. In expressions, we usually deal with 1-ary or 2-ary operators, whereas we can deal with functions of arbitrary arity.

The propositional operators are sometimes called "connectives". A typical example of a connective is $\wedge$, read "and". For example, with the above interpretation,

$$b \wedge c$$

would stand for:

"Disneyland is in Claremont and it is raining in Claremont."

More than wanting to know whether this overall statement is true or not, we want to know how its truth depends on the truth of the constituent proposition variables *b* and *c*. This can be succinctly described by giving the value of the statement for all possible values of b and c in the form of a function table. We have already used such tables before. When we are dealing with functions on propositions or bits, the table is called a "truth table". Such a table can appear many ways. Two common ways are (i) with a stub enumerating all assignments to b and c in a 1-dimensional array, or (ii) with separate stubs for b and c, in a 2-dimensional array.

b	c	b∧c
0	0	0
0	1	0
1	0	0
1	1	1

Figure 128: Representation (i) of ∧

b∧c	c		
		0	**1**
b	**0**	0	0
	1	0	1

Figure 129: Representation (ii) of ∧

Any 2-ary function on the truth values can be described by such a table, and any such table describes a function. Since there are 4 different possible assignments of 0 and 1 to b and c, and each of these can be assigned a value, either 0 or 1, there are $2^4 = 16$ different 2-ary truth functions. Rather than present a separate table with stubs for each, we can use form (i) above with a single stub and show all 16 functions.

args		Function Number															
b	c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Value of Function given Arguments

Figure 130: The sixteen proposition logic functions of 2 arguments.

Notice that in providing this **numbering** of the functions, we are using one of our encoding principles mentioned earlier. That is, an encoding of the set of functions is implied in our presentation. Moreover, we also see that there is a correspondence

between the set of 2-ary functions and the set of subsets of a four element set. This will be useful later on when we deal with the need to simplify the presentation of functions, which will correspond to minimizing the hardware in a digital circuit. Finally note that the number we give for each function corresponds to the decoded binary represented by the corresponding column, when read most-significant bit at the top.

The following is a description of functions "in order of importance". For some of the functions, we give mnemonic rules for help in remembering when the function gives value 1.

Function	also written	Explanation	Mnemonics
f_0 (0000)	0	constant 0 function	
f_1 (0001)	$\wedge$, ·, &, &&, $\cap$, ',' (comma in Prolog), and implied by juxtaposition when no operator is shown: i.e. if p, q, and r are understood as proposition symbols, $p \vee qr$ means $p \vee (q \wedge r)$	"and" function	$p \wedge q == 1$ when both $p == 1$ and $q == 1$ $p \wedge q == 0$ when either $p == 0$ or $q == 0$
f_2 (0010)		negation of “implies”	
f_3 (0011)	π_1	projection of first argument	
f_4 (0100)		negation of “if”	
f_5 (0101)	π_2	projection of second argument	
f_6 (0110)	$\oplus$, $\not\equiv$	"exclusive-or" ("xor")	$p \oplus q == 1$ when p has the opposite value of q
f_7 (0111)	$\vee$, +, |, ||, and $\cup$	"or" function	$p \vee q == 1$ when either $p == 1$ or $q == 1$ $p \vee q == 0$ when both $p == 0$ and $q == 0$
f_8 (1000)	$\downarrow$	"nor" (not-or) Also called the “dagger” or joint-denial function.	nor(p, q) == 1 when $p == 0$ or $q == 0$
f_9 (1001)	$\equiv$, $\leftrightarrow$, $\Leftrightarrow$, and ==	"iff" ("if and only if")	result is 1 exactly when both arguments are equal
f_{10} (1010)	$\neg$	negation of second argument	
f_{11} (1011)	$\leftarrow$, $\Leftarrow$, $\subset$, and :-	“if” function	

	(the last in Prolog		
f_{12} (1100)	$\neg$	negation of first argument	
f_{13} (1101)	$\rightarrow, \Rightarrow, \supset$	"implies" function	$p \rightarrow q$ == 1 when p == 0 or q == 1 $p \rightarrow q$ == 0 when p == 1 and q == 0
f_{14} (1110)	|	"nand" (not-and) Clasically called the "Sheffer stroke" function or alternative-denial.	nand(p, q) == 1 when p == 0 and q == 0
f_{15} (1111)	1	constant 1 function	

Aside from the definition of the function in the truth table, there are some associations we should make with the commonly-used functions. Remember that the value of a function can only be 1 or 0. So it suffices to state exactly the cases in which the value is 1, the other cases being implied to be 0. We make these statements using "iff" to mean "if, and only if":

($b \wedge c$) == 1 iff b == 1 **and** c == 1
($b \vee c$) == 1 iff b == 1 **or** c == 1
($b \rightarrow c$) == 1 iff b == 0 **or** c == 1
($b \oplus c$) == 1 iff b is not equal to c
($\neg b$) == 1 iff b == 0

Stand-Alone Convention

Because 1 is equated to true, we sometimes omit the == 1 in a logical equation. In other words, we would read

$$b \wedge c$$

standing alone as

b and c

i.e. *b* is true and *c* is true. Likewise, since == can be regarded as an operator on bits, it behaves as "iff":

b iff c

is the same as

b == c

in the stand-alone convention, or

(b == c) == 1.

Finally, using the stand-alone convention

$$\neg b$$

the negation of b, would be the same as $(\neg b) == 1$, meaning that *b* is false (0).

Tautologies

> A **tautology** is a propositional expression that evaluates to 1 for every assignment of values to its proposition variables.

When we use the stand-alone convention for propositional expressions without any further qualifications on the meaning of variables, we are asserting that the expression is a tautology. The following are examples of tautologies:

$$1$$
$$\neg 0$$
$$p \vee \neg p$$
$$p \rightarrow p$$

However, there will be many cases where it is not so obvious that something is a tautology. Here are some examples:

$$(p \rightarrow q) \vee (q \rightarrow p)$$
$$p \rightarrow (q \rightarrow p)$$

Both of the above tautologies might look unintuitive at first. To prove that they are tautologies, one can try evaluating each assignment of 0 and 1 to the variables, i.e. construct the truth table for the expression, and verify that the result is 1 in each case.

Example Show that $(p \rightarrow q) \vee (q \rightarrow p)$ is a tautology.

For $p = 0, q = 0$: $(0 \rightarrow 0) \vee (0 \rightarrow 0) == 1 \vee 1 == 1$
For $p = 0, q = 1$: $(0 \rightarrow 1) \vee (1 \rightarrow 0) == 1 \vee 0 == 1$
For $p = 1, q = 0$: $(1 \rightarrow 0) \vee (0 \rightarrow 1) == 0 \vee 1 == 1$
For $p = 1, q = 1$: $(1 \rightarrow 1) \vee (1 \rightarrow 1) == 1 \vee 1 == 1$

Part of the reason that this formula might not appear to be a tautology concerns the way that we English speakers use words like “implies” in conversation. We often use “implies” to suggest a *causal* relationship between two propositions, such as:

“Not doing homework” implies “low grade in the course”.

In logic, however, we use what is called the *material* sense of implication. Two propositions might be quite unrelated causally, and still an implication holds:

“Disneyland is in Claremont” implies “It is raining in Claremont”

While there is obviously no relation between the location of Disneyland and whether it is raining in Claremont, the fact that the lefthand proposition has the value 0 (false) renders the above a true statement, since $0 \rightarrow p$ regardless of the truth value of p.

The other source of misdirection in the tautology $(p \rightarrow q) \vee (q \rightarrow p)$ is that we are not saying that one can choose any p and q whatsoever and it will either be the case that always $p \rightarrow q$ or always $q \rightarrow p$. Rather, we are saying that no matter what values we assign p and q, $(p \rightarrow q) \vee (q \rightarrow p)$ will always evaluate to 1. Thus

("It is sunny in Claremont" implies "It is raining in Claremont")
or ("It is raining in Claremont" implies "It is sunny in Claremont")

is true as a whole, even though the individual disjuncts are not always true.

Substitution Principle

The substitution principle is the following:

> **Substitution Principle**
>
> In a tautology, if we replace all occurrences of a given propositional variable with an arbitrary propositional expression, the result remains a tautology.

The reason this is correct is that, in a tautology, it matters not whether the original variable before substitution is true or false; the overall expression is still invariably true.

Example

In the tautology $p \vee \neg p$, replace p with $a \rightarrow b$. The result, $(a \rightarrow b) \vee \neg(a \rightarrow b)$ is also a tautology.

Logic Simplification Rules

These rules follow directly from the definitions of the logic functions $\wedge$, $\vee$, etc. In part they summarize previous discussion, but it is thought convenient to have them in one place.

For any propositions p, q, and r:

$\neg(\neg p) == p$	double negative is positive
$(p \wedge 0) == (0 \wedge p) == 0$	0 absorbs $\wedge$
$(p \wedge 1) == (1 \wedge p) == p$	$\wedge$ ignores 1
$(p \vee 1) == (1 \vee p) == 1$	1 absorbs $\vee$
$(p \vee 0) == (0 \vee p) == p$	$\vee$ ignores 0
$(p \vee \neg p) == 1$	the excluded middle
$(p \wedge \neg p) == 0$	the excluded miracle
$(p \rightarrow q) == (\neg p \vee q)$	$\rightarrow$ as an abbreviation
$(0 \rightarrow p) == 1$	false implies anything
$(0 \rightarrow p) == 1$	anything implies true
$(p \rightarrow 1) == 1$	$(1 \rightarrow p)$ forces p
$(p \rightarrow 0) == \neg p$	$(p \rightarrow 0)$ negates p
$\neg(p \wedge q) == (\neg p) \vee (\neg q)$	DeMorgan's laws
$\neg(p \vee q) == (\neg p) \wedge (\neg q)$	DeMorgan's laws
$p \wedge (q \vee r) == (p \wedge q) \vee (p \wedge r)$	$\wedge$ distributes over $\vee$
$p \vee (q \wedge r) == (p \vee q) \wedge (p \vee r)$	$\vee$ distributes over $\wedge$
$p \vee (\neg p \wedge q) == (p \vee q)$	complementary absorption rules
$\neg p \vee (p \wedge q) == (\neg p \vee q)$	complementary absorption rules
$p \wedge (p \vee q) == (p \wedge q)$	complementary absorption rules
$\neg p \wedge (p \vee q) == (\neg p \wedge q)$	complementary absorption rules

The first few of these rules can be used to justify the following convention, used in some programming languages, such as Java:

Short-Circuit Convention

Evaluation of Java logical expressions involving

&& for "and" ($\wedge$)

|| for "or" ($\vee$)

takes place left-to-right only far enough to determine the value of the overall expression.

For example, in Java evaluating

```
f() && g() && h()
```

we would evaluate f(), then g(), then h() in turn only so long as we get non-0 results. As soon as one gives 0, the entire result is 0 and evaluation stops. This is of most interest when the arguments to && and || are **expressions with side-effects**, since some side-effects will not occur if the evaluation of the logical expression is "short circuited". This is in contrast to Pascal, which always evaluates all of the expressions. Experienced programmers tend to prefer the short-circuit convention, so that redundant computation can be avoided.

Exercises

1 • Express the functions f_2 and f_4 from the table of sixteen functions of two variables using $\{\wedge, \vee, \neg\}$.

2 •• Does the exclusive-or function $\oplus$ have the property of commutativity? Of associativity?

3 •• Which of the following distributive properties are held by the exclusive-or function?

$p \wedge (q \oplus r) == (p \wedge q) \oplus (p \wedge r)$	$\wedge$ distributes over $\oplus$
$p \vee (q \oplus r) == (p \vee q) \oplus (p \vee r)$	$\vee$ distributes over $\oplus$
$p \oplus (q \wedge r) == (p \oplus q) \wedge (p \oplus r)$	$\oplus$ distributes over $\wedge$
$p \oplus (q \vee r) == (p \oplus q) \vee (p \oplus r)$	$\oplus$ distributes over $\vee$

9.5 Logic for Circuits

A general problem in computer design is that we need to implement functions on the bit domain out of a library of given functions. Such a library might include primitive circuits for implementing $\wedge$, $\vee$, $\neg$, etc. It would likely include some **multi-argument** variants of these. For example, we have both the **associative** and **commutative** properties:

$a \wedge b == b \wedge a$	commutative property of $\wedge$
$a \vee b == b \vee a$	commutative property of $\vee$
$a \wedge (b \wedge c) == (a \wedge b) \wedge c$	associative property of $\wedge$
$a \vee (b \vee c) == (a \vee b) \vee c$	associative property of $\vee$

When both the associative and commutative properties hold for a binary operator, we can derive a function that operates on a bag of values (i.e. repetitions are allowed and order is not important). For example,

$$\wedge (a, b, c, d) == a \wedge (b \wedge (c \wedge d))$$

We do not need to observe either ordering or grouping with such operators; so we could use the equivalent expressions

$a \wedge b \wedge c \wedge d$

$\wedge (d, c, b, a)$

among many other possibilities.

Now consider the following:

Universal Combinational Logic SynthesisQuestion

Given a function of the form $\{0, 1\}^N \rightarrow \{0, 1\}$ for some N, is it possible to express the function using functions from a given set of functions, such as $\{\wedge, \vee, \neg\}$, and if so, how?

As it turns out, for the particular set $\{\wedge, \vee, \neg\}$, we can express *any* function for any number N of variables whatsoever. We might say therefore that

$$\{\wedge, \vee, \neg\} \text{ is universal}$$

However, this set is not uniquely universal. There are other sets that would also work, including some with fewer elements.

Modulo 3 Adder Synthesis Example

Consider the functions f_1 and f_2 in our modulo-3 adder example, wherein we derived the following tables:

	f_1	wx 00	wx 01	wx 10
	00	0	0	1
uv	01	0	1	0
	10	1	0	0

Table for the first result bit of encoded modulo 3 addition.

	f_2	wx 00	wx 01	wx 10
	00	0	1	0
uv	01	1	0	0
	10	0	0	1

Table for the second result bit of encoded modulo 3 addition.

How can we express the functions f_1 and f_2 using only elements from $\{\wedge, \vee, \neg\}$? The reader can verify that the following are true:

$$
\begin{aligned}
f_1(u, v, w, x) == \; & (u \wedge \neg v \wedge \neg w \wedge \neg x) \\
& \vee (\neg u \wedge v \wedge \neg w \wedge x) \\
& \vee (\neg u \wedge \neg v \wedge w \wedge \neg x)
\end{aligned}
$$

$$
\begin{aligned}
f_2(u, v, w, x) == \; & (\neg u \wedge \neg v \wedge \neg w \wedge x) \\
& \vee (\neg u \wedge v \wedge \neg w \wedge \neg x) \\
& \vee (u \wedge \neg v \wedge w \wedge \neg x)
\end{aligned}
$$

How did we arrive at these expressions? We examined the tables for those combinations of argument values uvwx that rendered each function to have the result 1. For each such combination, we constructed an expression using only $\wedge$ and $\neg$ that would be 1 for this combination only. (Such expressions are called **minterms**. There are three of them for each of the two functions above.) We then combined those expressions using $\vee$.

We often use other symbols to make the propositional expressions more compact. Specifically,

> It is common to use either a postfix prime (') or an over-bar in place of $\neg$.
>
> It is common to use · in place of $\wedge$, or to omit $\wedge$ entirely and simply juxtapose the literals (where by a "literal" we mean a variable or the negation of a variable). A term constructed using $\wedge$ as the outer operator is called a **product** or a **conjunction**.
>
> We sometimes use + instead of $\vee$. An expression constructed using $\vee$ as the outer operator is called a **sum** or a **disjunction**.

Making some of these substitutions then, we could alternatively express f_1 and f_2 as

$$f_1(u, v, w, x) == u\, v'\, w'\, x' + u'\, v\, w'\, x + u'\, v'\, w\, x'$$

$$f_2(u, v, w, x) == u'\, v'\, w'\, x + u'\, v\, w'\, x' + u\, v'\, w\, x'$$

The following diagram indicates how we derive this expression for the first function.

wx

f_1	00	01	10
uv 00	0	0	(1)
01	0	(1)	0
10	(1)	0	0

u' v' w x'

u' v w' x

u v' w' x'

Figure 131: Showing the derivation of minterms for a function

Minterm Expansion Principle

We can apply the technique described in the preceding section to *any* function represented by a truth table. We call this the

> **Minterm Expansion Principle**
>
> To express a function as a sum of minterms, where a minterm is a product of literals that includes each of the arguments of the function:
>
> 1. Identify those combinations of variable values where the function has value 1.
>
> 2. Construct a product of literals corresponding to each combination. If a variable has value 1 in the combination, then the variable appears without negation in the product. If a variable has value 0 in the combination, then the variable appears with negation in the product.
>
> 3. Form the sum of the products constructed in step 2. This is the minterm expansion representation of the function.

The justification of this principle is straightforward. The function has value 1 for certain combinations and 0 for all others. For each combination where it has value 1, the corresponding minterm also has value 1. Since the minterm expansion is exactly the sum of those minterms, the function will have value 1 iff its minterm expansion has value 1.

The minterm expansion principle also shows us that the set $\{\wedge, \vee, \neg\}$ is universal, since the minterm expansion is made up of only these operators and variables. It tells us one way to implement a bit-function from primitive logic elements. Such an implementation is just a different representation of the minterm expansion, specifically a form of the

DAG representation for the syntax of the expression. For example, for the expression for f_1 above, the logic implementation would be shown as

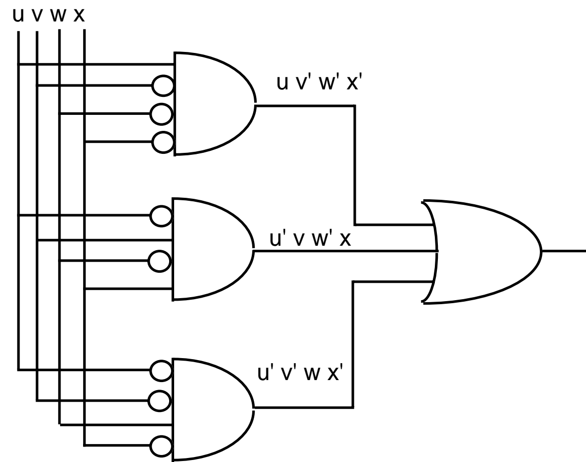


Figure 132: Implementation corresponding to minterm expansion of f_1(u, v, w, x) == u v' w' x' + u' v w' x + u' v' w x'

Here the small circles represent negation, the node with a curved left side is disjunction, and the nodes with straight left sides are conjunctions.

Later on, we will examine some ways to simplify such implementations, for example to reduce the amount of hardware that would be required. Meanwhile, it will be useful to have in our repertoire one other form of expansion that will help our understanding and analysis. This expansion will lead to an implementation sometimes different from the minterm expansion. However, the main uses of this principle will transcend those of minterm expansion.

Programmable Logic Arrays

A **programmable logic array** (**PLA**) is a unit that can be used to implement a variety of different functions, or several functions simultaneously. It is programmable in the sense that the functionality can be specified after the manufacture of the unit itself. This is done by blowing fuse links that are internal to the unit itself. This is allows a single integrated-circuit package to be used to implement fairly complex functions without requiring custom design.

PLAs are oriented toward a two-level gate combination, with the output being an OR-gate fed by several AND-gates of the overall inputs to the unit. Plain or inverted versions of each input signal are available. The structure of a PLA is depicted below. More than two levels can be obtained by connecting outputs of the PLA to inputs.

In the PLA, each AND and OR gate consists of many possible inputs. However, all of these possibilities are represented abstractly by a single wire. By putting a dot on that wire, we indicate a connection of the crossing line as an input. Thus functions that can be represented by two level sum-of-products (SOP) expressions can be coded in the PLA by reading directly from the expression itself.

Example Program a PLA to be a 3-bit binary incrementer modulo 8 (function that adds 1, modulo 8). The truth table for the incrementer is

input			output		
x_2	x_1	x_0	y_2	y_1	y_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

We wire the AND-gates to activate one column corresponding to each row of the truth table. We then wire the OR-gates to activate on any of the rows for which the corresponding output is 1. The result is shown below.

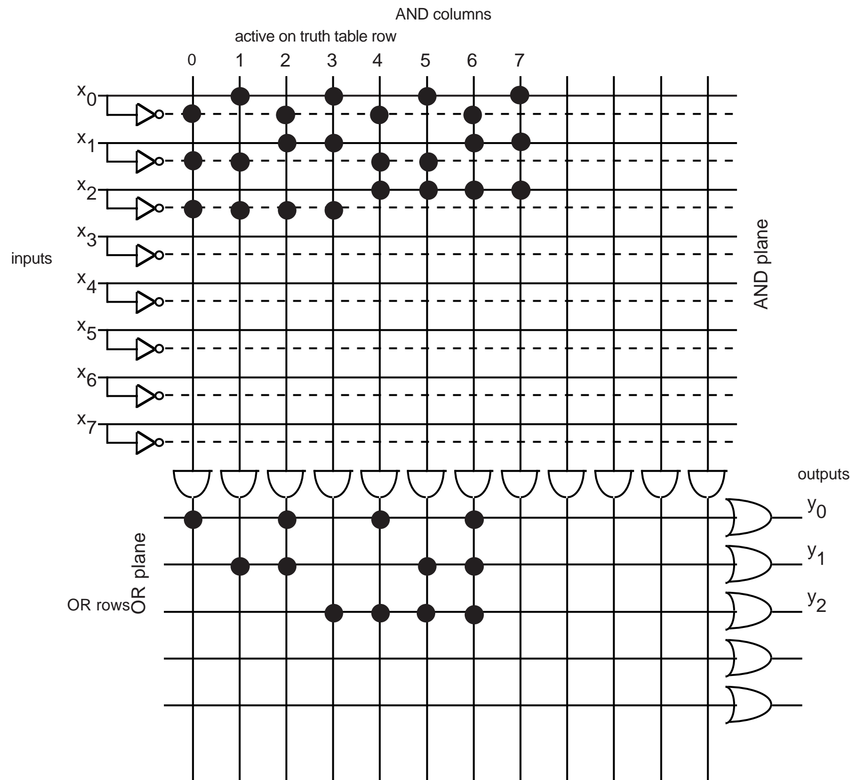


Figure 133: A PLA programmed to add 1 (modulo 8) to a 3-bit binary numeral

A less-cluttered, although not often seen, notation would be to eliminate the depiction of negation wires and indicate negation by open "bubbles" on the same wire. For the above logic functions, this scheme is shown below.

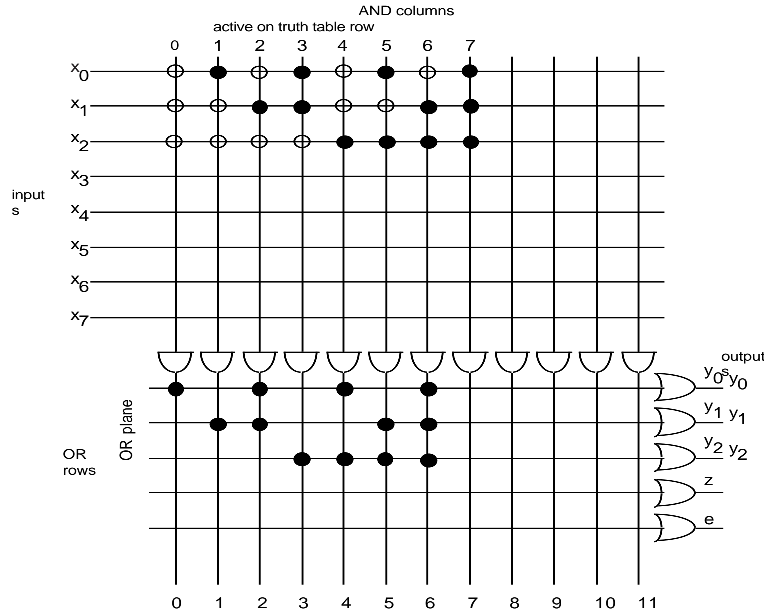


Figure 134: An alternate PLA notation for negation

Boole/Shannon Expansion Principle

Another way of establishing the universality of {$\wedge$, $\vee$, $\neg$}, as well as having other applications, is this important principle:

> **Boole/Shannon Expansion Principle**
> Let E be any proposition logic expression and p some proposition symbol in E. Let E_1 stand for E with all occurrences of p replaced with 1, and let E_0 similarly stand for E with all occurrences of p replaced with 0. Then we have the equivalence
>
> $$\mathbf{E == (p \wedge E_1) \vee (\neg p \wedge E_0)}$$

Proof: Variable p can only have two values, 0 or 1. We show that the equation holds with each choice of value. If p == 1, the lefthand side is equal to E_1 by definition of the latter. The righthand side simplifies to the same thing, since ($\neg 1 \wedge E_0$) simplifies to 0 and (p $\wedge$ E_1) simplifies to E_1. On the other hand, if p == 0, the lefthand side is equal to E_0. The righthand side again simplifies E_0 in a manner similar to the previous case.

There are several uses of this principle:

> Regrouping an expression by chosen variables (useful in logic circuit synthesis).
>
> Simplifying an expression by divide-and-conquer.
>
> Testing whether an expression is a *tautology* (whether it is equivalent to 1).

The Boole/Shannon Principle can be used to expand and analyze expressions recursively. Let us try it on the same expression for f_1 as discussed earlier. The righthand side for f_1 is

$$u\ v'\ w'\ x' + u'\ v\ w'\ x + u'\ v'\ w\ x'$$

If we take this to be E in the Boole/Shannon principle, we can choose any of the four variables as p. Let us just take the first variable in alphabetic order, u. The principle says that E is equivalent to

$$u\ E_1 + u'\ E_0$$

where E_1 is 1 v' w' x' + 1' v w' x + 1' v' w x', which immediately simplifies to v' w' x', since 1' is 0, which absorbs the other literals. Similarly, E_0 is 0 v' w' x' + 0' v w' x + 0' v' w x', which simplifies to v w' x + v' w x'. So we now have our original expression being recast as

$$u\ (v'\ w'\ x') + u'\ (v\ w'\ x + v'\ w\ x').$$

The implementation corresponding to the Boole/Shannon expansion could be shown as the following, where E_1 and E_0 can be further expanded.

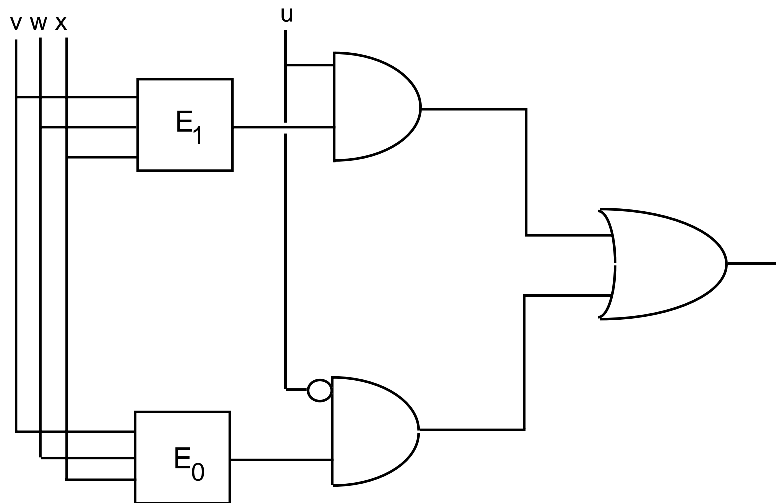


Figure 135: The Boole/Shannon principle applied to logic implementation

Incidentally, the structure below, which occurs in the Boole/Shannon principle, is known as a *multiplexor* or *selector*. It has a multitude of uses, as will be seen later. The reason for the name "selector" is that it can select between one of two logical inputs based upon the setting of the selection control line to 0 or 1. (Later we will call this an "address" line.)

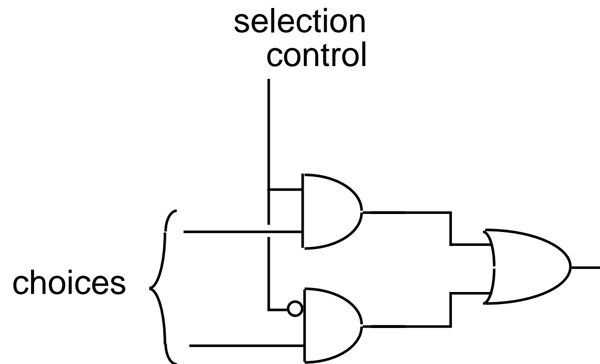


Figure 136: Multiplexor or selector structure

The multiplexor structure can be thought of as the hardware analog to the *if* statement in programming languages.

Tautology Checking by Boole/Shannon Expansion Example 1

Let's investigate whether $(p \to q) \vee (q \to p)$ is a tautology using the Boole/Shannon principle. Choose the variable p for expansion. Then

E_1 is $(1 \to q) \vee (q \to 1)$

E_0 is $(0 \to q) \vee (q \to 0)$

Since we know $(q \to 1) == 1$, E_1 simplifies to 1. We also know $(0 \to q) == 1$, so E_0 simplifies to 1. Thus E is equivalent to

$p \cdot 1 \vee p' \cdot 1$

which is a known tautology. Therefore the original is a tautology.

Observations In creating a Boole/Shannon expansion, the original expression is a tautology iff E_1 and E_0 both are tautologies. Since E_1 and E_0 have one fewer variable than the original expression (i.e. neither contains p, for which we have substituted) we have recursive procedure for determining whether an expression is a tautology: Recursively expand E to E_1 and E_0, E_1 to E_{11} and E_{10}, E_{11} to E_{110} and E_{110}, etc. until no variables are left. [We don’t actually have to use the numberings in a recursive procedure, since

only two expressions result in any given stage.] If any of the limiting expressions is 0, the original is not a tautology. If all expressions are 1, the original is a tautology.

The following diagram suggests the use of repeated expansions to determine whether an expression is a tautology:

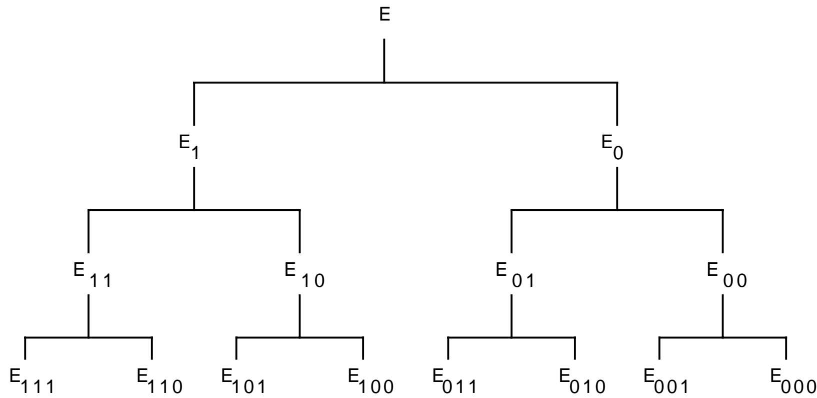


Figure 137: Tree showing the form of recursive use of Boole/Shannon expansion

Tautology Checking by Boole/Shannon Expansion Example 2

Let's determine whether or not $((a \rightarrow b) \wedge (b \rightarrow c)) \rightarrow (a \rightarrow c)$ is a tautology.

E is $((a \rightarrow b) \wedge (b \rightarrow c)) \rightarrow (a \rightarrow c)$.

Looking at E, we see that if we make c = 1, then the whole expression will simplify to 1. Thus c is a good choice for the first expansion variable.

Expanding *E* on *c*:

E_1 is $((a \rightarrow b) \wedge (b \rightarrow 1)) \rightarrow (a \rightarrow 1)$. Since $(a \rightarrow 1) == 1$ independent of *a*, this simplifies to $((a \rightarrow b) \wedge (b \rightarrow 1)) \rightarrow 1$, which further simplifies to 1 for the same reason. Thus we do not have to go on expanding E_1.

E_0 is $((a \rightarrow b) \wedge (b \rightarrow 0)) \rightarrow (a \rightarrow 0)$. Since for any p, $(p \rightarrow 0)$ is $\neg p$, E_0 simplifies to $((a \rightarrow b) \wedge \neg b) \rightarrow \neg a$.

Expanding the simplified E_0 on *a*:

E_{01} is $((1 \rightarrow b) \wedge \neg b) \rightarrow \neg 1$. This simplifies to $(b \wedge \neg b) \rightarrow 0$, which simplifies to $0 \rightarrow 0$, which simplifies to 1.

E_{00} is $((0 \rightarrow b) \wedge \neg b) \rightarrow \neg 0$, which simplifies to $(1 \wedge \neg b) \rightarrow 1$, which simplifies to 1.

Thus, by taking some care in choosing variables, we have shown the original E to be a tautology by expanding only as far as E_1, E_{01}, and E_{00}, rather than to the full set E_{111}, E_{110}, ... E_{000}.

Logic Circuit Simplification by Boole/Shannon Expansion Example

Occasionally when we expand on a particular variable using the Boole/Shannon expansion, the result can be simplified from what it would be with the full-blown multiplexor structure. Here once again is the equation for the Boole/Shannon expansion:

$$\mathbf{E == (p \wedge E_1) \vee (\neg p \wedge E_0)}$$

In the special case that E_1 simplifies to 0, the term $p \wedge E_1$ also simplifies to 0, so that E simplifies to $\neg p \wedge E_0$. Since several such simplifications are possible, let's make a table:

Case	**E simplifies to**
E_1 simplifies to 0	$\neg p \wedge E_0$
E_0 simplifies to 0	$p \wedge E_1$
E_1 simplifies to 1	$p \vee E_0$
E_0 simplifies to 1	$\neg p \vee E_1$
E_0 and E_1 simplify to 0	0
E_0 and E_1 simplify to 1	1
E_0 and E_1 simplify to the same thing	E_0
E_0 and E_1 simplify to opposites	$p \oplus E_0$

Table of some simplifications based on the Boole/Shannon expansion

For example, if E_0 simplifies to 0 then our original logic implementation based on the Boole/Shannon expansion could be replaced with the following much simpler one:

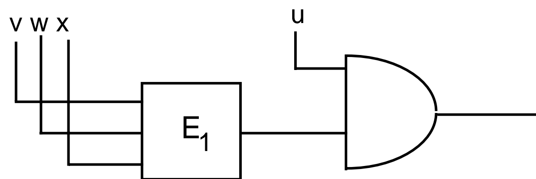


Figure 138: Simplified logic circuit as a result of Boole/Shannon expansion

Counterexamples to Propositions

When a logical expression is not a tautology, there must be some assignment of truth values to the variables under which the expression evaluates to 0 rather than to 1. Such an assignment is sometimes called a "counterexample". It is, of course, possible for multiple counterexamples to exist for a given expression.

The Boole/Shannon expansion tree can be used to produce counterexamples in the case that the starting expression is not a tautology. As discussed, a non-tautology must result in a node that simplifies to 0 somewhere in the tree. The path from the root to a given node corresponds to an assignment of truth values to some of the variables in the expression. Going to the left in the diagram corresponds to assigning the value 1 and to the right, the value 0. It is easy to see that the expression at a given node corresponds to a simplification of the expression under the set of choices made at each branch. Thus, if a node simplifying to 0 is encountered, the choices represented by the path from that node to the root for a counterexample.

Exercises

1 •• Show that the set {∨, ¬} is universal. [Hint: Show that ∧can be expressed using {∨, ¬}. Conclude that anything that could be expressed using only { ∧, ∨, ¬} could also be expressed using {∨, ¬}. Show that { ∧, ¬} is also universal.

2 •• Show that {*nand*} is universal. Show that {*nor*} is universal.

3 •• Show that the set of functions {→, ¬} is universal.

4 ••• Let **1** designate the constant 1 function. Is {**1**, ⊕} universal? Justify your answer.

5 •••• Show that the set of functions { ⊕, ¬} is not universal. [Hint: Find a property shared by all functions that can be constructed from this set. Observe that some functions don't have this property.]

6 ••• Is { ∧, ⊕} universal? Justify your answer.

7 ••••• Is it possible to devise a computer program to determine whether a set of functions, say each in the form of a truth table, is universal?

8 •• Show the implementation corresponding to the next phase of expansion using the Boole/Shannon principle to expand both E_1 and E_0 above.

9 •• Using the Boole/Shannon expansion principle, show each of the rules listed earlier in *Simplification Rules Worth Remembering*.

10 ••• Show that the "dual" form of the expansion principle, wherein $\wedge$ and $\vee$ are interchanged and 0 and 1 are interchanged.

11 ••• Verify that each of the simplifications stated in the *Table of some simplifications based on the Boole/Shannon expansion* is actually correct.

12 •• Think up some other useful simplification rules, such as ones involving $\oplus$ and $\equiv$.

13 •• Determine which of the following are tautologies using Boole/Shannon expansion:

$(p \wedge (p \rightarrow q)) \rightarrow q$
$\neg p \rightarrow p$
$\neg (\neg p \rightarrow p)$
$(\neg p \rightarrow p) \rightarrow p$
$\neg p \rightarrow (p \rightarrow q)$
$((p \rightarrow q) \rightarrow p) \rightarrow p$
$(p \rightarrow q) \vee (\neg p \rightarrow q)$
$(p \rightarrow q) \vee (p \rightarrow \neg q)$
$(p \rightarrow q) == (\neg q \rightarrow \neg p)$
$(p \vee q) \rightarrow (p \wedge q)$
$(p \wedge q) \rightarrow (p \vee q)$
$(p \rightarrow q) \wedge (q \rightarrow r) == (p \rightarrow r)$
$(p \rightarrow q) \wedge (q \rightarrow r) \wedge (r \rightarrow s) \rightarrow (p \rightarrow s)$

14 •• For those expressions in the last exercise above that turned out not to be tautologies, produce at least one counterexample.

15 ••• For the Logical Expression Simplifier exercise in the previous section, modify your program so that it gives a counter example for each non-tautology.

Karnaugh Maps

Karnaugh maps are a representation of truth tables for switching (proposition logic) functions that has uses in analysis and simplification. The idea can be traced to Venn diagrams used in visualizing sets. We assume the reader has prior exposure to the latter idea. To relate Venn diagrams to switching functions, consider a diagram with one region inside a universe. This region corresponds to a propositional variable, say x. Any 1-variable switching function corresponds to a shading of the area inside or outside the region. There are four distinct functions, which can be represented by the logical expressions x, x', 0, and 1. The corresponding shadings are shown below.

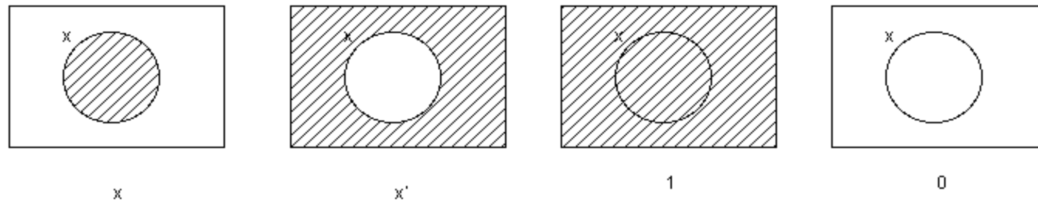


Figure 139: One-variable switching functions and their Venn diagrams.

Now consider two-variable functions. There are 16 of these and, for brevity, we do not show them all.

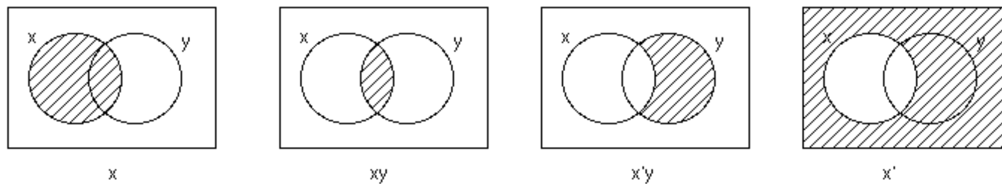


Figure 140: Some two-variable switching functions and their Venn diagrams.

The most important principle about Venn diagrams is that the sum of (the expressions representing) two or more functions can be depicted by forming the union of the shadings of the individual diagrams. This frequently leads to a view of the sum that is simpler than either summand.

Example

Show that x + x'y = x + y.

If we were asked to shade x + y, the result would be as shown below. On the other hand, the shadings for x and x'y are each shown in the previous figure. Note that combining the shadings of those figures results in the same shading as with x + y.

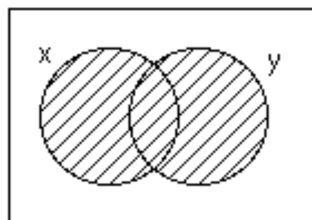


Figure 141: Venn diagram for x + y

Quite often, we would like to simplify a logical expression, but we don't know the answer in advance. To use Venn diagrams for this purpose, we would "plot" each of the

summands on the diagram, then "read off" a simplified expression. But it is not always obvious what the simplified result should be.

Example

Simplify xy'z' + x' + y.

The figure shows shadings for each of the summands, followed by the union of those shadings. The question is, what is the best way to represent the union? We can get a clue from the unshaded region, which is xy'z. Since this regions is unshaded, the shaded regions is the complement of this term, (xy'z)', which by DeMorgan's law is x' + y + z'.

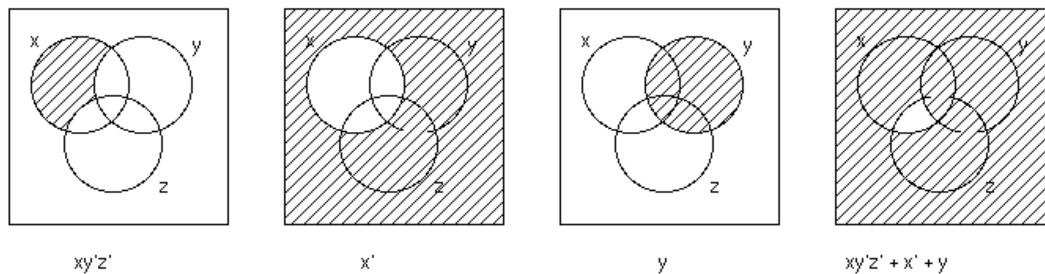


Figure 142: Venn diagrams for various expressions

Karnaugh maps are a stylized form of Venn diagram. They are most useful for simplifying functions of four variables or fewer. They can be used for five or six variables with more difficulty, and beyond six, they are not too helpful. However, a mechanizable method known as "iterated consensus" captures the essence of the technique in a form that can be programmed on a computer.

From Venn Diagrams to Karnaugh Maps

To see the relationship between a Karnaugh Map and a Venn diagram, let us assume three variable functions. The transformation from a Venn diagram to a Karnaugh map is shown below. Note that we are careful to preserve adjacencies between the primitive regions on the diagram (which correspond to minterm functions). The importance of this will emerge in the way that Karnaugh maps are used.

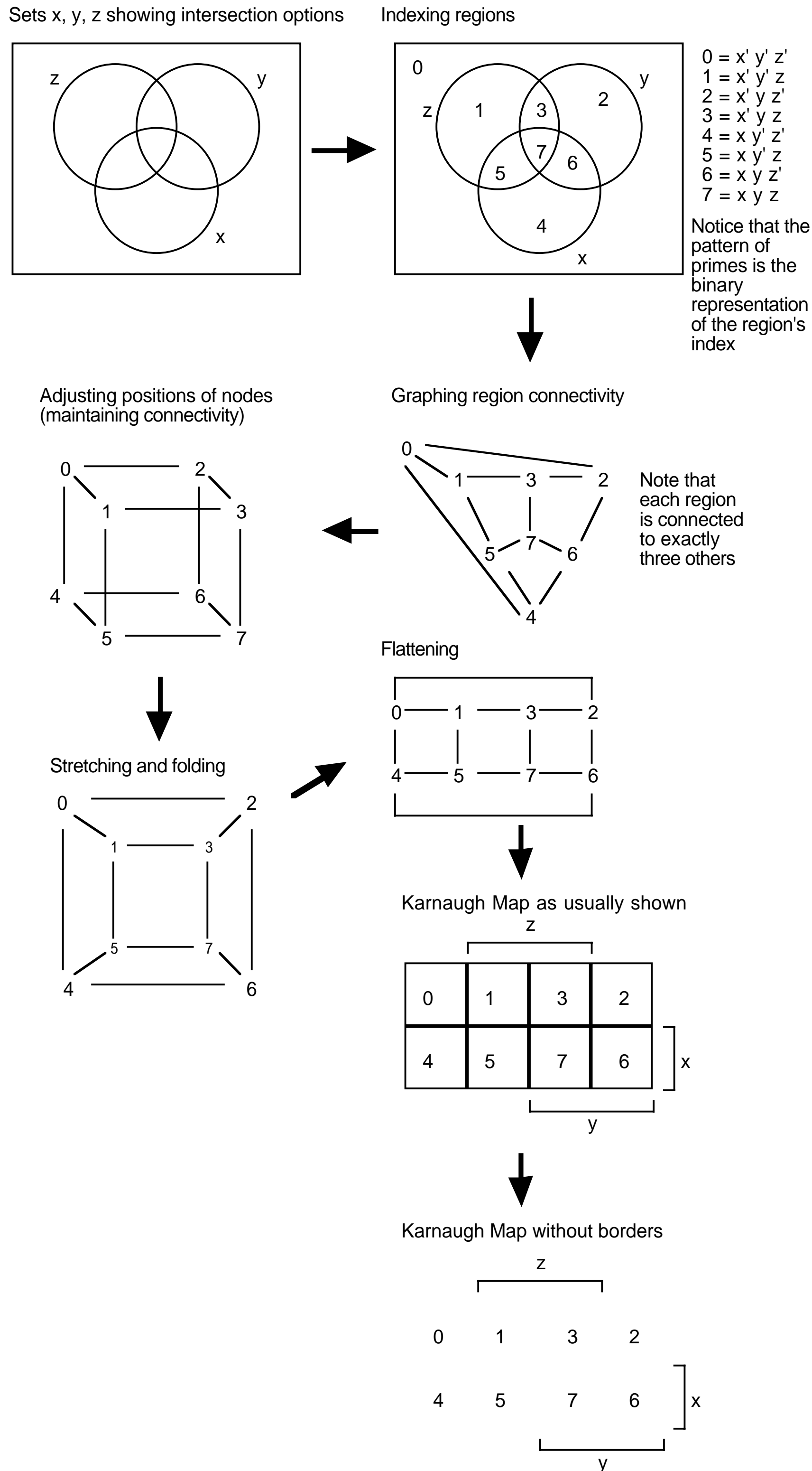


Figure 143: Transforming a Venn diagram into a Karnaugh map.

Hypercubes

Recall that a sum-of-product (SOP) form is a logical sum of products, where each product consists of either variables or their negations. For an n-variable function, each possible product term corresponds exactly to some sub-cube in the **n-dimensional hypercube H_n**, defined in an earlier chapter:

> H_0 is a single point.
>
> H_{n+1} is two copies of H_n, where each point of one copy is connected to the corresponding point in the other copy.

A **sub-cube** is a set of points in Hn that itself forms a hypercube. Above, the following sets of points are examples of sub-cubes: 0246, 1357, 0145, 2367, 04, 02, 0, and 01234567.

Conversely, each sub-cube corresponds to such a product term. Examples are shown in the various attachments. Therefore, any SOP form corresponds to a *set* of sub-cubes, and conversely. The truth table corresponding to such a function can be equated with the union of the points in those sets. These points are the rows of the table for which the function result is 1. Note that any given point might be present in more than one sub-cube. The important thing is that all points are "covered" (i.e. each point is included in at least one). Moreover, no sub-cube can contain points for which the function result is 0.

The means we have for making an SOP simpler are:

> Reduce the number of terms.
>
> Reduce the size of terms.

These two objectives translate into:

> Use **fewer** sub-cubes to cover the function (so there are fewer terms)
>
> Use **larger** sub-cubes (so the terms have fewer literals).

For example, given the choice between two sub-cubes of size 4 and one of size 8, we would always choose the latter.

One of the contributions of the Karnaugh map is to enable spotting the *maximal* sub-cubes, the ones that are not contained in other sub-cubes for the same function. These sub-cubes are usually called the *prime implicants* of the function. The word "prime" in this case carries the same meaning as "maximal". The word "implicant" means that each term in the SOP for a function *implies* the function itself, i.e. whenever the assignment to variables is such that the term is 1, the function itself must be 1. This is because the

function can be represented as the *sum* of such terms.

Karnaugh Map Example

Consider function f(x, y, z) = x'y' + x'yz + xy'z + xy

In this SOP, as in all, each term is an implicant. However, only x'y' and xy are prime. The other two terms correspond to sub-cubes of a larger implicant z, as can be seen from the following map.

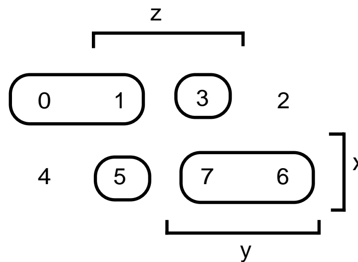


Figure 144: Karnaugh map for x'y' + x'yz + xy'z + xy

By examining the map, we can see that the sub-cube 1-3-5-7 corresponds to an implicant, in this case z. (It is a sub-cube by definition of "sub-cube", and it is an implicant because the function's value for all of its points are 1.) We can thus add this sub-cube to our SOP without changing the function's meaning:

f(x, y, z) = x'y' + x'yz + xy'z + xy + z.

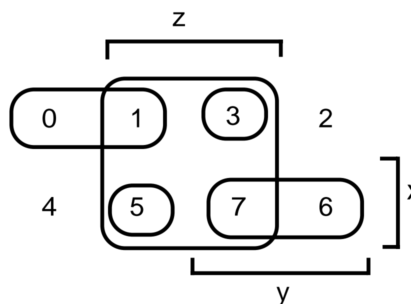


Figure 145: Adding a sub-cube to the map without changing the function

Then we notice that two other terms, xy'z corresponding to sub-cube 5, and x'yz corresponding to sub-cube 3, are both redundant. They are both *subsumed* by the new term z. (C is said to *subsume* D whenever D implies C.) Restricted to sub-cubes, C subsumes D when the points of C include all those of D.

As a consequence, we can eliminate the subsumed terms without changing the meaning of the function:

$$f(x, y, z) = x'y' + xy + z.$$

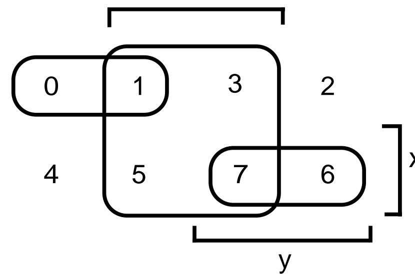


Figure 146: Eliminating subsumed terms from the map

Obviously we have achieved our goal of simplifying the function, by both reducing the number of terms, as well as the complexity of those terms. If we were implementing via NAND gates, we would have to use one NAND gate of 3 inputs and two of 2 inputs for this SOP, vs. one of 4 inputs, two of 3 inputs, and two of 2 inputs, for the original SOP, quite obviously a saving.

Notice that we cannot achieve further simplification by constructing still larger implicants from this point. Each implicant shown is prime.

What we have suggested so far is:

> The simplest SOP is constructed only of prime implicants.

We next observe that including *all* prime implicants is not necessarily the simplest. Since prime implicants can overlap, there might be redundancy if we include all. The following example shows this:

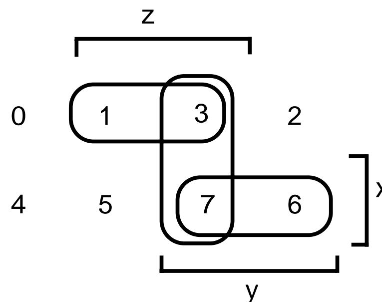


Figure 147: A Karnaugh map with redundancy among prime implicants

In the above example, the prime implicants are x'z, yz, and xy, corresponding to 13, 37, and 67 respectively. However, we do not need the second prime implicant to cover all points in the function. On the other hand, the first and third prime implicants will be required in any SOP for the function that consists only of prime implicants. Such prime implicants are called *essential*.

It is possible for a non-trivial function to have no essential prime implicants, as the following example shows:

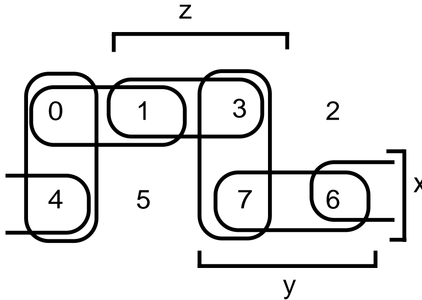


Figure 148: A Karnaugh map with no essential prime implicants

(Note that, because a map represents a hypercube, adjacency in the map extends to the “wrap-around” cases, such as xz’ shown above.)Here there are six prime implicants, yet none is essential. In each SOP that covers the function, we can leave out one of the prime implicants. Thus we have six different implementations of the same complexity, five 2-variable prime implicants each.

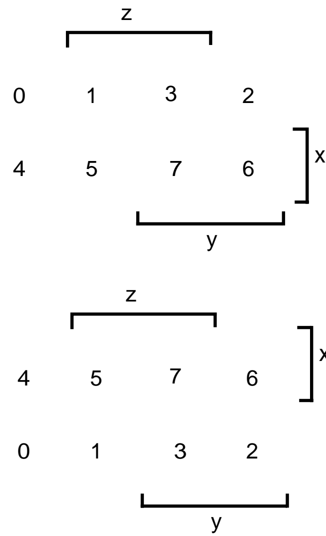
The observations about redundancy are part of the reason that simplification of switching functions is complex. We cannot use a simple, straightforward, algorithm to get the optimum implementation. Instead it appears that we must generally resort to a more complex "backtracking" process. We do not pursue this further in these notes.

Karnaugh Maps of Higher Dimension

Karnaugh maps work well for representing hypercubes of dimension up to four. After that, they become harder to use for visualization. However, the principle on which they are based, called “consensus”, can still be applied in a computer program, which is not limited to what can be visualized by human eyes. The figure below shows a 4-dimensional Karnaugh map obtained by juxtaposing two 3-dimension ones, one of which has been flipped over so that the cells with x = 1 are adjacent. This allows us to form the following sub-cubes:

- any 2 adjacent cells (horizontally or vertically, including wrap-around)
- any 4 adjacent cells in a 1x 4, 2 x 2, or 4 x 1 configuration, including wrap-around
- any 8 cells in a 2x 4 or 4 x 2 configuration, including wrap-around

Three variables: xyz



Four variables: wxyz

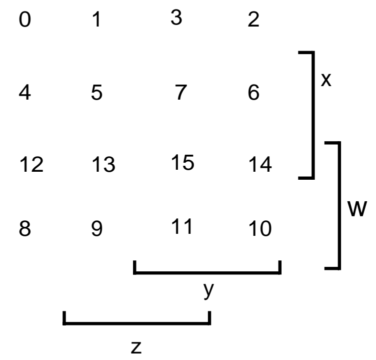


Figure 149: A 4-dimensional Karnaugh map

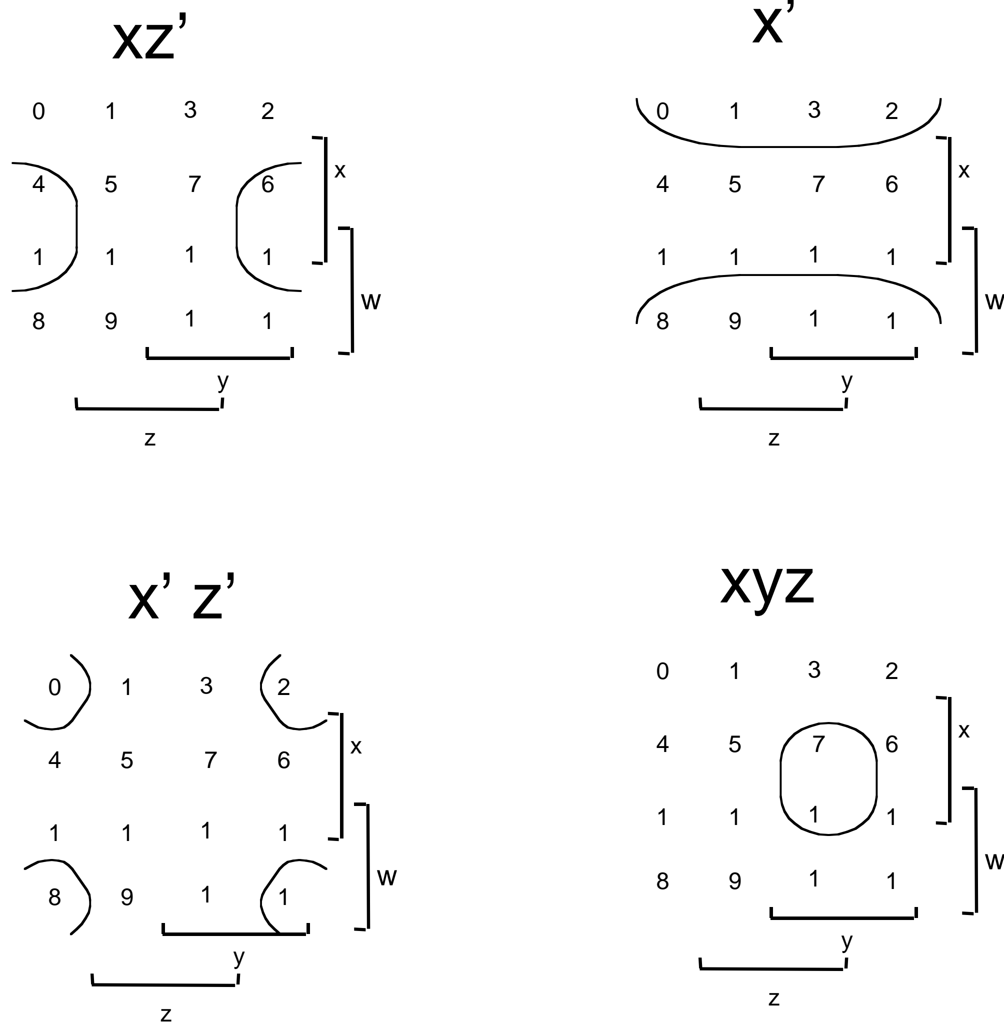


Figure 150: Sample sub-cubes on a 4-dimensional map

Functions and Karnaugh Maps with "Don't Cares"

In proceeding from a natural language problem statement to a switching function representation of the problem, the resulting function might not always be completely specified. That is, we will care about the function's results (0 or 1) for *some* combination of variables, but not care about its results for other combinations. One reason we might not care is that we know from the problem statement that these combinations cannot occur in practice.

Such "don't care" combinations often provide a bonus when it comes to finding

simplified SOP forms. Rather than stipulating an arbitrary choice of the function's value for these variables at the outset, we can wait until the simplification process begins. The technique is summarized as:

Choose the function value for don't care combinations to be 1 if it helps maximize the size of a covering sub-cube.

Example

Below we show a Karnaugh map for a function, where point 5, corresponding to term xy'z, is marked with a "d", indicating that we don't care about the function's output for that combination. In contract, the function's value is to be 0 for combinations x'yz' and xy'z', and 1 for all other combinations.

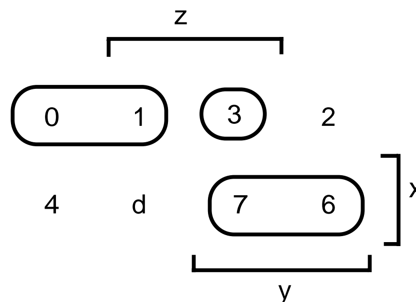


Figure 151: A Karnaugh map with "don't care" (marked d)

The choice of whether to cover any given cell marked "d" is up to us. Above, if we chose not to cover it (make the function have value 0 for xy'z), we would have the simplified implementation shown below, with SOP x'y' + x'z + yz + xy. Further simplification is possible in that one of the terms x'z or yz can be dropped without affecting the coverage: Either of x'y' + yz + xy or x'y' + x'z + xy both work.

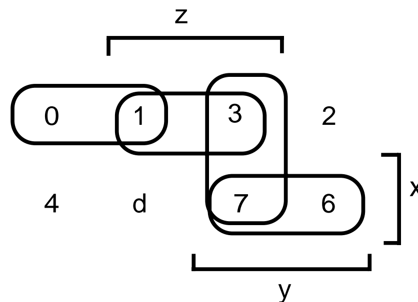


Figure 152: The prime implicants when d is assigned 0

If we choose to cover the cell marked "d" (make the function have value 1 for xy'z), we have the simplified implementation with SOP x'y' + xy + z, which is simpler than either of the simplest cases where we don't cover the d:

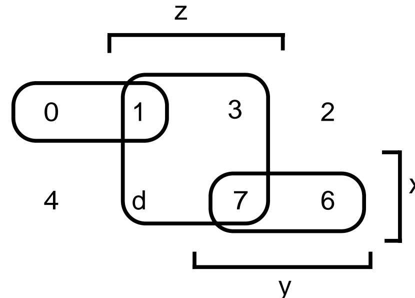


Figure 153: The prime implicants when d is assigned 1

In this case, it seems obvious that we should cover the d.

Iterated Consensus Principle for Finding Prime Implicants (Advanced)

Although what we are about to describe can be extended to include don't care cases, we choose not to do so for reasons of simplicity. When faced with functions with a large number of variables, we obviously would like to turn to the computer as a tool for simplification. Unfortunately, the technique presented so far for Karnaugh maps involves "eyeballing" the map to find the prime implicants. How can we express an equivalent technique in such a way that it can be represented in the computer? In short, what is the essence of the technique? This method is given the name "iterated consensus", and relies on two principles: consensus and subsumption.

The iterated consensus technique takes as input any set of product terms representing the function of interest. As output, it produces all of the prime implicants of the function. It is up to further analysis to use those prime implicants in constructing the simplest implementation.

The iterated consensus technique proceeds through a number of intervening states, where each state is a set of product terms. Initially this set is whatever is given as input. Finally, this set is the set of prime implicants. The consensus and subsumption principles are used to repeatedly modify the set until no further modifications are possible.

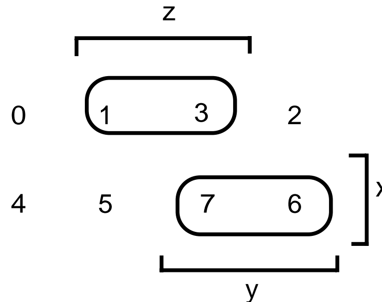
subsumption rule: If term U in the set is subsumed by a term V, then term U can be dropped from the set.

In terms of the Karnaugh map, this is equivalent to dropping a contained sub-cube.

consensus rule: If term U is a term not in the set, but which is the consensus of two

other terms V and W in the set, then term U can be added to the set. (However, terms that are subsumed by other terms in the set should not be added; they would just be removed by the subsumption rule anyway.) The exact definition of “consensus” will be given below; for now, we are discussing an informal example.

The consensus rule corresponds to introducing a new sub-cube on the map formed from points already covered by other sub-cubes. To see this, let us look at a typical map situation:



Clearly we can add the sub-cube 37 corresponding to the term yz. This term is the consensus of terms x'z and xy corresponding to the sub-cubes already covered. (We know that this sub-cube is not needed to cover the function, but the purpose of iterated consensus is to find prime implicants, and yz is certainly one.)

What we are saying by adding the consensus term is that

$$x'z + xy = x'z + xy + yz$$

To express the consensus in general, we note that the new term yz is found by the following considerations: For some variable, in this case x, one term has the variable complemented, the other uncomplemented. If we represent those terms as:

$$xF$$

and

$$x'G$$

then the consensus is just

$$FG$$

(in this case F is y, G is z, and therefore FG is yz).

Definition of Consensus: If U and V are two product terms, then:

If there is a single variable, say x, which appears uncomplemented in U and

complemented in V, then write U as xF and V as x'G (or the symmetric case, with U as x'F and V as xG), where both F and G are free of x. The consensus of U and V is defined to be FG. The operative identity in this case is:

xF + x'G = xF + x'G + FG

If the preceding case does not obtain, then the consensus is defined to be 0 (some authors would say it is "does not exist").

In terms of the Karnaugh map, the condition for the consensus to be non-zero is that the sub-cubes for F and G be "touching" on the map. The consensus term is the largest sub-cube that can be combined from sub-cubes of F and G, as suggested below.

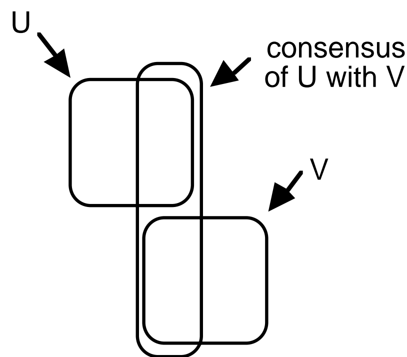


Figure 154: Showing consensus on a Karnaugh map

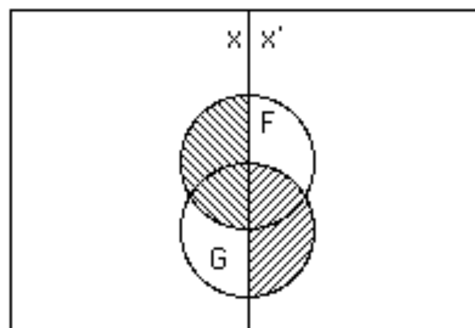


Figure 155: Showing consensus on a Venn diagram

Exercises

A good example of functions with don't cares can be found in various display encoders. For example, a seven-segment display consists of seven LEDs (light-emitting diodes), numbered s0 through s6, which display the digits from 0 through 9 as shown:

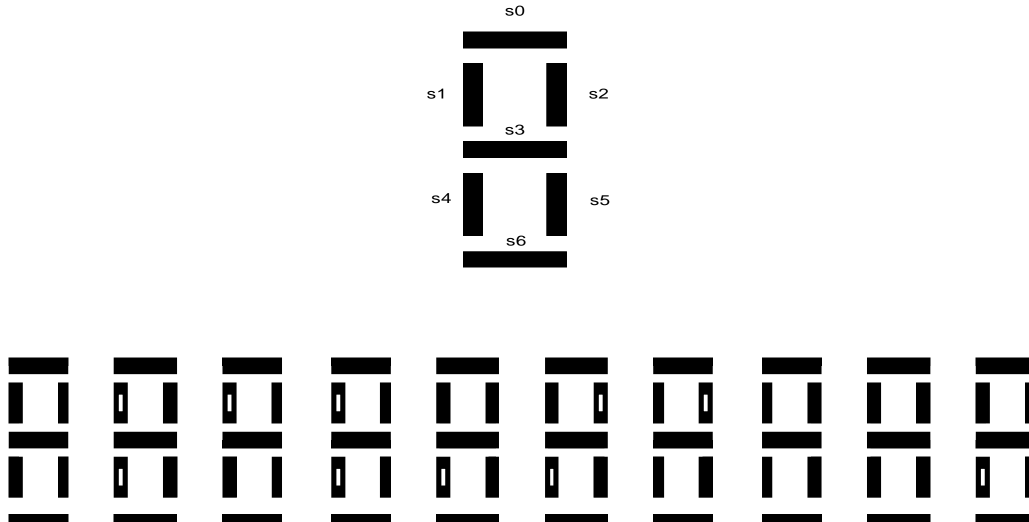


Figure 156: Display of 0 through 9 with a seven segment display

In the following, we assume that the digits are coded in BCD. This uses ten of sixteen possible combinations of four bits. The remaining combinations are don't cares. The seven segments correspond to seven switching functions.

1 • Give a Karnaugh map (with don't cares) for each of the switching functions.

2 •• Simplify each of the switching functions, using don't cares to advantage.

3 •• Construct gate realizations of the switching functions. Determine any possible sharing of product-terms among multiple functions.

4 •••• Develop a program that will input a logical expression and output the set of prime implicants for the corresponding function. Although different input formats are possible, a suggested internal format is to use a sequence of the symbols 0, 1, and x to represent a product term. For example, if we are dealing with a four-variable function, say f(u, v, w, x), then 01x0 represents u'vx'. (The x in the sequence represents that the absence of the corresponding letter, or equivalently, the union of two sub-cubes that are alike except for the values of that variable.) The reason for this suggestion is that the consensus of two such sequences is easy to compute. For example, the consensus of 01xx with 0x1x is 011x. This corresponds to u'v + u' w = u'vw.

9.6 Logic Modules

Although truth-tabular methods, maps, and the like are essential for understanding how computer logic works, they are not necessarily the best tools for building large systems, the problem being that the size of a truth table becomes overwhelming, even to a computer, when there are many inputs. The reason, of course, is that there are 2^N different input combinations for N input lines, and this number becomes large very fast. In order to handle this issue, designers structure systems using *modules* with understood behavior. At some level, truth-tabular methods are probably used to design aspects of these modules, but the modules themselves are understood using logic equations rather than tables.

Adder Module

A typical module found in a computer adds numbers represented by binary numerals. This module might be depicted as in the upper-left portion of the figure below. It could be realized by expanding it into the simpler FA ("full adder") modules shown in the main body of the figure. The term “full” is used for an adder module that adds three bits: two addend bits and a carry-in bit, to produce two bits: a sum and a carry-out bit. It contrasts with a “half adder”, which only adds two bits to produce a sum and carry-out. This type of adder structure is called a "ripple-carry" adder because the carry bits "ripple" through the FA gates. In the extreme case where all inputs, including the carry, are 1, the output carry production is delayed because it is a function of all of those input bits.

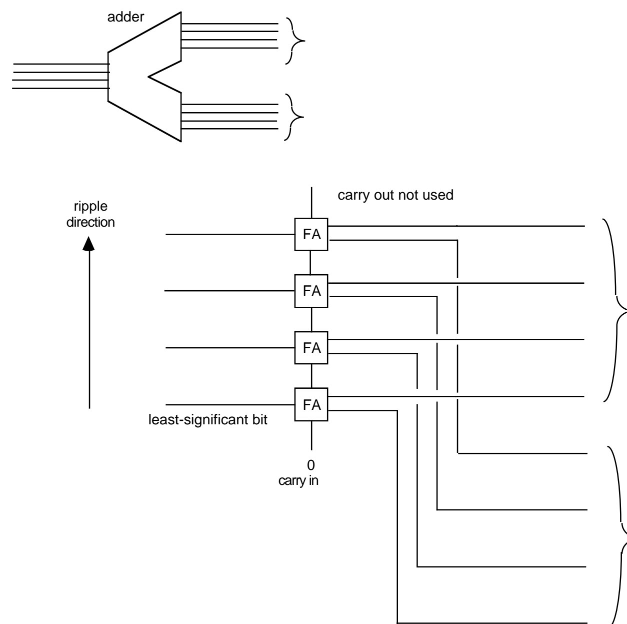


Figure 157: Expansion of the adder using ripple-carry; FA units are "full-adders"

The next figure shows a possible expansion of the FA modules using simpler logic functions. The M (majority) module has an output of 1 when 2 out of 3 inputs are 1. Therefore, its equation is:

carry-out = M(a, b, c) = ab +ac + bc

The $\oplus$ module is a 3-input exclusive-OR, i.e.

sum-out = a $\oplus$ b $\oplus$ c

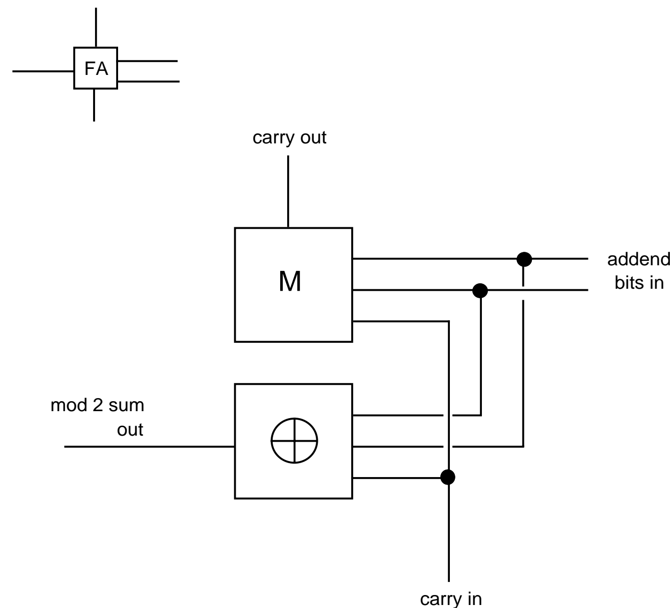


Figure 158: Expansion of the FA using M (majority) and 3-input exclusive-OR

Exercises

1 • Earlier we introduced the idea of a multiplexer, a module that has three inputs: two data inputs and an address input. The address input selects one or the other data input and reflects whatever is on that input to the output. Give a truth-table for the multiplexor. Although there are three input lines, we call this a "2-input" multiplexer, because selection is between two input liens.

2 •• Show the structure of a 4-input multiplexer. This unit will have inputs a, b, c, d and two address lines (assuming the address is encoded in binary). Such a device is shown below. (Hint: Use three 2-input multiplexers.)

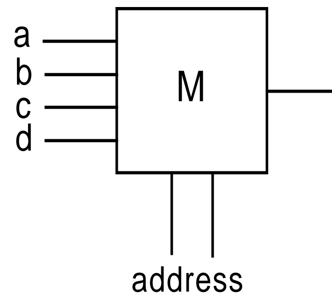


Figure 159: A 4-input multiplexer

3 ••• Show how to construct a 2n-input multiplexer from n-input multiplexers, for general n.

4 ••• Show how to construct recursively a 2^n-input multiplexer using only multiplexers with fewer inputs. If it is desired to build a multiplexer with 2^n inputs total, how many 2-input multiplexers would be required? (Construct a recurrence equation for the latter number and solve it.)

5 •• For the preceding problem, assuming that a singe 2-input multiplexer delays the input by 1 time unit, by how many units does your 2^n-input multiplexer delay its input.

6 ••• Show how a 4-input multiplexer can be used to implement an arbitrary combinational function of 2 logical variables. (Hint: Use the address lines as inputs and fix the *a*, *b*, *c*, *d* inputs.)

7 ••• Using the scheme of the previous problem, an 2^n-input multiplexer can be used to implement an arbitrary combinational function of how many logical variables?

8 ••• A *demultiplexer* (also called **DMUX**) reverses the function of a multiplexer, in that it has one input and several outputs. Based on the value of the address lines, the input is transmitted to the selected output line. The other output lines are held at 0. Show how to implement a 2-output and 4-output demultiplexer using simple logic gates.

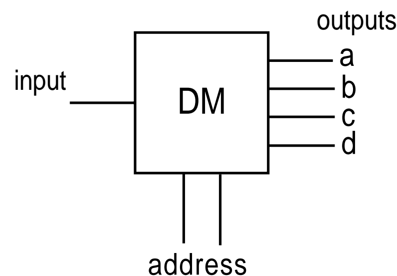


Figure 160: A 4-output demultiplexer

9 •• Show how to implement a 2^n-output demultiplexer for any n. How many simple gates are required in your construction?

10 •• A *decoder* is like a demultiplexer, except that the input is effectively held at a constant 1. Thus its outputs are a function of the address bits only. The decoder can be thought of as a converter from binary to a one-hot encoding. Suppose we have on hand an N-input decoder. What is the simplest way to build up an N-input multiplexer from this? What about building an N-output demultiplexer?

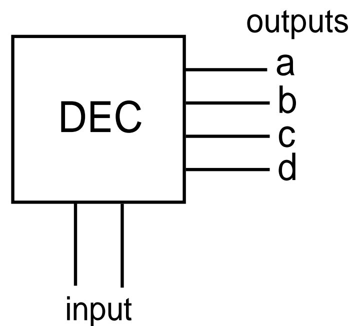


Figure 161: A 2-input, 4-output decoder, arranged to show the similarity to a demultiplexer.

11 •• Show that the outputs of a 2^n-output decoder are exactly the minterm functions on the address line variables.

12 •• An *encoder* reverses the role of outputs and inputs in a decoder. In other words, it converts a one-hot encoding to binary. Show how to build a 4-input encoder from simple logic gates.

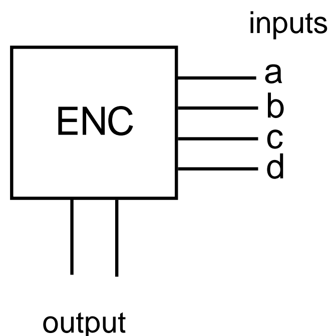


Figure 162: A 4-input, 2-output encoder.

13 ••• Show how to build a 3-input encoder from simple logic gates.

14 ••• Show how to build a 2^n-input encoder using a recursive construction.

15 •• Explore composing various combinations of encoder, decoder, multiplexer, demultiplexer together. What overall functions result?

16 ••• An "N-bean counter" counts the number of 1's among N input lines, presenting the output in binary. Design a logic circuit for a 3-bean counter (which will have 3 input and 2 output lines). Give a recursive construction for an N-bean counter, for arbitrary N.

9.7 Chapter Review

Define the following terms:

associative
binary code
binary-coded decimal
Boole/Shannon expansion
Cartesian encoding
combinational switching
commutative
conjunction
DeMorgan's laws
disjunction
don't-care condition
encoding
full adder
Gray code
half adder
hypercube
iff
implies
Karnaugh map
minterm expansion
one-hot code
parity
programmable logic array
proposition
subset code
substitution principle
tautology
universal set of switching functions
Venn diagram

9.8 Further Reading

George Boole, *An Investigation of the Laws of Thought*, Walton, London, 1854 (reprinted by Dover, New York, 1954). [Boole used 1 - *t* for the negation of *t*, working as if *t* were a number. The Boole/Shannon Expansion is stated: "If *t* be any symbol which is retained in the final result of the elimination of any other symbols from any system of equations, the result of such elimination may be expressed in the form

$$Et + E' (1-t)= 0$$

in which *E* is formed by making in the proposed system $t = 1$, and eliminating the same other symbols; and E' by making in the proposed system $t = 0$, and eliminating the same other symbols. Moderate.]

Frank Brown, *Boolean Reasoning*, Kluwer Academic Publishers, Boston, 1990. [Encyclopedic reference on Boolean algebra, with some applications to switching. Moderate to difficult.]

Augustus De Morgan, *On the Syllogism*, Peter Heath, ed., Yale University Press, New Haven, 1966. [DeMorgan's laws are stated: "(A, B) and AB have ab and (a, b) for contraries." Moderate.]

Martin Gardner, *Logic Machines and Diagrams*, University of Chicago Press, 1982. [Surveys diagrammatic notations for logic. Easy to moderate.]

Maurice Karnaugh, *The Map Method for Synthesis of Combinational Logic Circuits*, Transactions of the American Institute of Electrical Engineers, 72, 1, 593-599, November, 1953. [Introduction of the Karnaugh map.]

C.E. Shannon, *The synthesis of two-terminal switching circuits*, Trans. of the American Institute of Electrical Engineers, 28, 1, 59-98, 1949. [Gives a later version of the Boole/Shannon expansion.]

John Venn, *Symbolic Logic*, Chelsea, London, 1894. [Discourse on logic, introducing Venn diagrams, etc. Moderate]

Alfred North Whitehead and Bertrand Russell, *Principia Mathematica*, Cambridge University Press, London, 1910. [An original reference on logic. Moderate to difficult (notation).]