FROM BINARY TO QUANTUM LOGIC SYNTHESIS WITH DESIGN ALGORITHMS

Marek Perkowski

October 9, 2006

Contents

List of Tables

List of Figures

Chapter 1

Combinational Digital Circuits

1.1 Electronic Communication

Automation brought about the industrial revolution of the 18th and 19th centuries which led to better transportation that expedited the exchange of goods, services, and information in society. The greatest benefits to mankind, however, came with the use of electricity which not only provided for more comfort, enjoyment, and ease from toil through lighting, heating, work power, etc. but also brought about a communication age in which information could be exchanged faster than ever before. With the telegraph, a message could be transmitted across the United States a hundred times faster than by letter. The Morse code used in telegraphy is a system of communication in which electrons, when transmitted, are used to produce short and long patterns of sound, flashes of light, written dots and dashes, etc. which represent letters and numbers. The teletypewriter, on the other hand, is an electromechanical typewriter that either transmits or receives messages coded in electrical signals. Upon receiving a specific electrical signal, a mechanical function within the typewriter is performed that causes a particular letter, number, character, etc. to be typed on the paper. Since the computer, however, is expected to simulate human patterns of thought and behavior, further development was needed before its advent would be possible.

Humans have a memory in which information is stored. Their inputs are through taste, feeling, sight, hearing, and smell. They understand languages and can, therefore, communicate verbally as well as nonverbally. The input that a human receives is compared with memory, decoded to obtain meaning, a decision is made, and action can be taken if required. The problem to solve in constructing a computer was, therefore, to find a way to utilize a flow of electrons in such a manner that a machine would perform similarly.

Traditionally, we have used two different methods of communicating with computers. One way is for the machine to detect (feel, sense, read) whether current (or voltage) is present or not. This is said to be a **digital signal**. The other one is the **analog signal** in which measurements of the voltage can represent a process which can be described mathematically by differential equations in a continuous function of time. Therefore, the analog signal is a continuous wave signal in which voltage is always present and there are no interruptions. In order to facilitate understanding the analog signal, visualize the infinite number of points on a real number line. With real numbers like, for example, 0.01, 0.003, etc., there is always an intermediate number between any two given points. This means that, in the analog signal, one thing flows into the other in a smooth continuation with no breaks or levels.

Measurements of temperature with a thermometer or of pressure with a pressure gauge can be used as analogies. Mercury in a tube can be used to show the rise and fall of temperature. We graduate a scale to measure the mercury in the tube and to show the number of units of mercury which we then denote as degrees. The measurement of mercury, however, is not the temperature. It is only analog of the temperature. It rises and falls with the temperature and the height of the mercury in the tube can be measured at infinitely many points on the scale. An analog computer is comparable to this analogy because it performs a direct simulation of a physical system.

The analog signal finds good application in devices where processing is restricted to operations such as amplification or filtering, but high frequency of operation is a must. High fidelity recordings or reading information from a hard disk exemplify such a case. Sound impulses on the recording are converted to continuous electrical signals that are amplified and then reconverted to sound by the speakers. The continuous electrical signals are analog to the sound impulses. It is easy to perceive the voice transmissions in our present-day telephone system functioning in the same manner as described above. Dailing and switching, however, and even voice transmission, to some extent, is performed digitally. As time goes by, more and more systems are being converted to operate on digital signals and it is expected that, in the near future, most electrical systems will be utilizing this principle. It remains an open question what role, if any, analog circuits will play in computing systems of tomorrow. In this book however, we will concentrate on digital circuits only.

The current (or voltage) that the present-day electronic digital systems detect are digital signals in the form of individual **"bits"** (or discrete elements) of information: a **binary (two-state)** method of communication. The advantage of this method is due to the fact that the possibility of directing current to a specific point exists. These two discrete binary values simulate human logic much better for our purposes. It is either hot or cold, light or dark, with no grey areas in between, because it is relatively easy for a digital circuit to recognize a special level of pulses.

The word **"digital"** as in digital circuits, however, is not synonymous to the word **"binary"**. The term "digital computer" actually emerged from the fact that the early digital computers of XX Century were used mostly for numerical computations in which the discrete elements were **digits**. A **decimal system** could be used and the system would still be digital. A circuit with ten states and a discrete voltage value for each state, however, would be less reliable. Today, as a rule, when the term "digital circuit" is used it is usually understood that the **binary digital system** is meant by it.

The punched cards invented in XIX century (see history of computing in one of the next chapters) are an example of a digital system that is not **binary** (base-two), but **decimal** (base-ten). With such cards, a census taker or market research firm, for example, could make up a questionnaire and code the answers with zero through nine digits. Eighty coded answers would then fit on one card. When students take multiple-choice tests and record the answers on **scantron sheets**, they are read by an electronic, digital device. Each shaded area then stands for a specific answer. When the codes are entered into the digital computer, a code converter converts them into a binary code that the circuits are designed to accept.

The number "9" could be represented by 1001 or 100 0001 could stand for the letter "A", for example (codes will be discussed later). For now, it will suffice to say that the binary number 10101 in the digital computer means that voltage is "on" at the first, third, and fifth position while the voltage is "off" at the second and the fourth positions. In denoting these positions as such, we refer to what is called **"positive logic"** which will be used in all of the examples throughout this textbook.

1.2 Switching Circuits

Modern computers and microprocessors containing logic circuits actually perform both qualitative and quantitative functions and would, therefore, best be described as **"data" processing systems**. Data being defined as "bits" of information that is not necessarily numerical in nature. Nevertheless, the average person most likely thinks of a "computer" as an electronic device that performs mathematical, usually arithmetical, operations. We are usually not aware of all the other logic devices in our lives. Perhaps this is because people are used to interacting with computers when they need help. They use them as tools in accounting, science, compiling statistics, etc. Otherwise, logic devices go about their work without being noticed: in telephone systems, elevators, car alarms, heating and air conditioning systems, automatic bank tellers, automobile electrical systems, or perhaps even in quality control on the product assembly line, etc. These "tools" are logic, or **algorithmic**, devices since they perform a sequence of tasks in which the next step is based on the information on hand and depends on the preceding events: they operate algorithmically.

1.2.1 The Algorithm

As human beings, we usually act instinctively in a logical manner without mapping out every detail, but when we have a large number of tasks to perform we also make plans and list the problems to be solved. A messenger company with several deliveries throughout the city would surely plan them in sequence to avoid driving more than necessary. On the other hand, when baking a cake, the ingredients would first need to be mixed before being put into the oven, but if we were out of flour we would first have to buy some. We know such algorithms from computer programming and realize that we must first have a plan laid out which can be communicated to the computer before we can expect it to do our bidding.

Since digital circuits are designed in such a way that they will simulate human logic, these algorithms, which outline a path describing a sequence of events, could also be viewed as the circuit diagram itself. Algorithms are, therefore, an important part of anyone's life who deals with digital circuits in any manner. Logic, on the other hand, does not imply that a conclusion is either correct or incorrect, moral or immoral. The result is only as true as the premise that the reasoning is based on. If the premise is false, sound reasoning may produce a logically sound answer that is equally false. Therefore, whether designing, programming, or servicing logic circuits, the performance of the device will be only as good as our own. [1]

[1] it is true in logic that a true statement is implied by a false statement

An algorithm can be represented by a **flowchart** to help visualize the internal circuitry of a digital device. Suppose, we wanted to plan a day off from work. Our activities could depend on several parameters. What if it is raining or snowing? On the other hand, the sun may be shinning and it may be hot. We, therefore, provide for these possibilities as shown in the flowchart from Figure **??**.

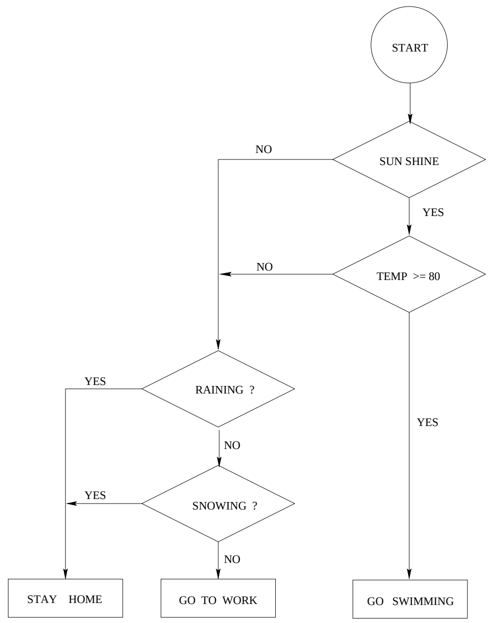


Figure 1.1: Flowchart example: Planning of a day

This shows that if **both conditions** are met, the **sun is shining** and the $temperature \geq 80\ degrees$, we go **swimming**. If either of these conditions fails, we test for **rain** and subsequently for **snow**. If either rain or snow is a given condition, we stay at **home**. Otherwise, we have no excuses and we go to work.

1.2.2 The Binary Logic

In digital circuits, logic describing our actions can best be shown by a switching circuit diagram instead of a flowchart. We can then show digital devices in a circuit which perform logical operations to accommodate statements such as the following.

```
If the sun is shinning AND the temperature is greater or equal
than 80 degrees, go swimming.

If the sun is NOT shinning OR the sun shines but the
temperature is NOT greater or equal 80 degrees AND
it is raining OR snowing, stay at home.
```

```
If it is  NOT raining AND NOT snowing, go to work.

If the sun is shinning  AND the temperature is larger or equal
than 80 degrees, do NOT stay at home.

If it is snowing  OR raining, do  NOT go swimming.

If it is snowing OR raining, but NOT both, go to work.
```

To meet the above requirements, we need circuits that represent the `AND`, `OR`, and `NOT` cases of logic.

`AND` simply means that both (or all) cases must be true before the function is performed. `OR` requires that `either` only one `or` the other (at least one out of all possibilities) be true. `NOT` is simply the opposite of its argument logic value.

Electrical signals such as voltages or currents exist throughout a digital system in either one of two recognizable values (except during transition). Voltage-operated circuits, for example, respond to two separate voltage levels which represent a binary variable equal to **logic-1** or **logic-0**. If we picture an active element such as an electromagnetic relay, diode, or transistor, etc. serving as on/off switch, we could think of electronic digital circuits as **switching circuits.** In this case, when the switch is closed, the device is conducting and it is assigned a value of logic-1, since voltage would be detected as being present. If the switch were open the value would be logic-0, because there would be no voltage present.

LOGIC-AND

Recall that before the logic-AND function can be performed, both (or all) of the requirements must be met. In an electrical switching circuit, this would be equivalent to having a series circuit with one switch behind the other. Then all of the switches would have to be closed before the function could be carried out. If only one of the switches were open, the requirement would not be met because the whole circuit would be open, see Figure **??**.

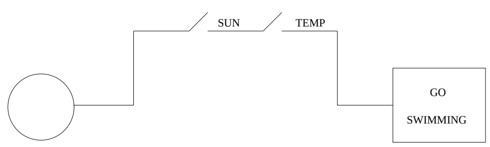


Figure 1.2: Logic AND function - serial connection of switches

LOGIC-OR

The logic-OR function is performed if at least only one of the requirements is met. This is equivalent to having a parallel switching circuit because only one of the switches needs to be closed, see Figure **??**.

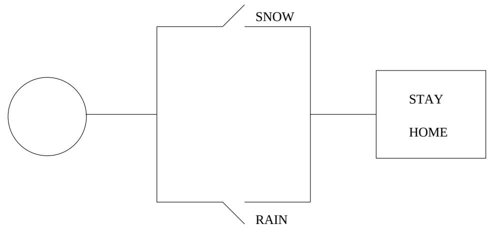


Figure 1.3: Logic OR function - parallel connection of switches

An entire logical statement can be constructed and illustrated by a combinational, series/parallel circuit. The flowchart given in section **??** is shown again below as a switching circuit's diagram. In this case, there are three possible outputs, or conclusions. If the sun is shinning AND the temperature $\geq$ 80 degrees, the output is: "Go swimming". However, if it is raining OR it is snowing, the output is: "Stay at home", see Figure **??**. When comparing the above switching circuit to the flowchart example, think of `"NO"` as a logic-0 when the switch is open and `"YES"` as a logic-1 when the switch is closed.

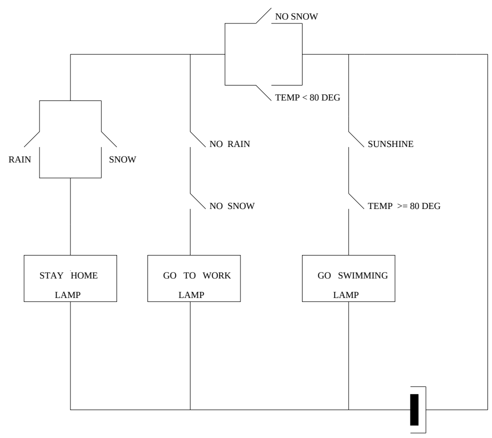


Figure 1.4: Three output switching circuit

We can also make up a logic circuit from a **menu**.

Statement:

```
The meal consists of:
 Soup or salad,
  entree,
 and dessert.
For dessert, there is a choice of fruit, cake, or ice cream.
```

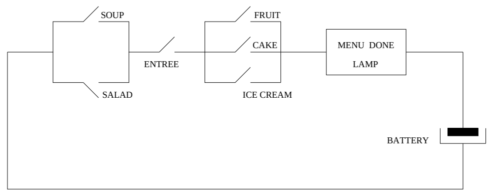


Figure 1.5: Circuit gives ON on lamp menu when you select soup or salad, the entree and one of possible dessert choices

(Note that in English it is understood that the comma after `"salad"` above stands for `"and"`.) The circuit is shown on Figure ??. Whenever you do a choice with the input switches, the meal lamp will go `on`. Otherwise it will stay `off`, meaning that your choice of lunch remains incomplete.

Visualize the ordering from the above menu by closing an appropriate switch to indicate your choice. Notice that the conjunction `"and"` will always mean that the nouns connected by it are in series. The conjunction `"or"` implies to branch because there is more than one possibility.

Example 1.1

The following situation demonstrates a more realistic practical application of switching logic:

> There are two different transmission lines, line # 1 and line # 2, that need to be supplied with current at all times, see Figure ??. As shown below, source # 1 usually supplies line # 1 while source # 2 supplies line # 2. However, should one of the sources fail, its line could be fed by the other source. If both sources fail, an auxiliary source can supply both lines. Needless to say, a line should not be connected to two sources at the same time. There are five switches (Sw_1, Sw_2, Sw_3, Sw_4, Sw_5) that allow the current to be directed as desired. What combinations meet requirements 1, 2, and 3?

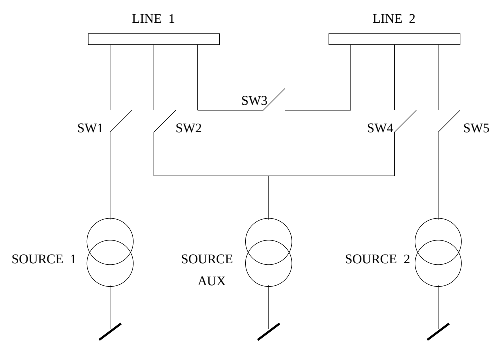


Figure 1.6: Circuit with three sources for two lines

• As a rule, line # 1 would receive current from source # 1 by closing switch # 1.
• Therefore, Sw_1 = line # 1.
• Likewise, line # 2 receives current from source # 2 when switch # 5 is closed.
• Therefore, Sw_5 = line # 2.
• If source # 1 failed, line # 1 could be fed by source # 2 by closing switch # 3 between the lines.
• Therefore, Sw_5 **AND** Sw_3 = Line # 1 and line # 2.
• The reverse would be true if line # 2 needed to be supplied by source # 1.
• Sw_1 **AND** Sw_3 = Line # 1 and line # 2.
• If the auxiliary source were needed, switches Sw_2 and Sw_4 would be closed to supply both lines.
• Sw_2 **AND** Sw_4 = Line # 1 and line # 2.
• If both lines were supplied by only one source, there would be three possibilities:

$$(Sw_3 \textbf{ AND } Sw_5) \textbf{ OR } (Sw_1 \textbf{ AND } Sw_3) \textbf{ OR } (Sw_2 \textbf{ AND } Sw_4). \tag{1.1}$$

In digital logic notations, however, the algebraic sign for multiplication of A and B (a dot between symbols of arguments; or simply, AB) is used to denote the logic-AND. Multiplication is not implied, even though the logic-AND is also referred to as a product. Likewise, the plus sign (+) is used to denote the logic-OR case, but addition is not meant with it even if the logic-OR is referred to as a sum.

Therefore, the following equation should be written from Equation **??**.

$$(Sw_3 \cdot Sw_5) \ + \ (Sw_1 \cdot Sw_3) \ + \ (Sw_2 \cdot Sw4) \tag{1.2}$$

Either combination of switches provides a solution to supply both lines with current. Let us observe that this expression assumes that just one source feeds both lines.

LOGICAL-NOT

As we have shown, variables can also be used, as in mathematics, to abbreviate logic expressions. Each variable, however, also has a **complement**. The complement of the variable A, would be its opposite and would be denoted by $\overline{A}$. We refer to both the variable and its complement as **literals.** Terms such as AB or $A+B$ also have complements: $\overline{AB}$ or $\overline{A+B}$. This is the logic-NOT: `NOT-A, NOT-B, NOT-(AB), NOT-(A+B).`

To illustrate the use of the logic-NOT, let us refer back to the "menu-circuit."
The variables can be declared as follows:
so = soup
sa = salad
e = entree
f = fruit
c = cake
i = ice cream
M = meal to complete the function.
Of course, when ordering, it would normally suffice to indicate only what is desired. It is then implied that the other items are not wanted. In circuit logic, however, it is necessary

$$\begin{array}{rclcccc} voltage \cdot voltage & = & voltage: & 1 \cdot 1 = 1 & & (all \cdot all = all) \\ no\ voltage \cdot no\ voltage & = & no\ voltage: & 0 \cdot 0 & = & 0 & (nothing \cdot nothing = nothing) \\ no\ voltage \cdot voltage & = & no\ voltage: & 0 \cdot 1 & = & 0 & (nothing \cdot all = nothing). \end{array}$$

to be specific. Therefore, we need to include all the switches and indicate which of them are required and which are NOT required.

Observe that in the previous example some always hungry person could order both soup and salad, plus all three desserts, and the light menu would be lightened.

If we define a proper menu that you can eat either soup or salad but not both, only one of: fruit, cake, or ice cream but not more, the circuit would be as in Figure ??.

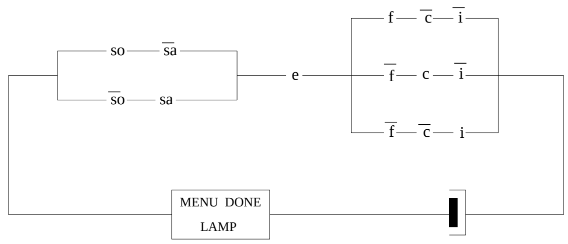


Figure 1.7: A new menu restricts the number of dishes you can eat

The function would be written:

$$M \;=\; (so \cdot \overline{sa} + \overline{so} \cdot sa)(e)(f \cdot \overline{c} \cdot \overline{i} \;+\; \overline{f} \cdot c \cdot \overline{i} \;+\; \overline{f} \cdot \overline{c} \cdot i)$$

1.3 Definition of Binary Logic

Recall that there are two different states possible in a digital device. Either there is a voltage signal present or it is NOT present. "Voltage present" can also be thought of as being a `YES`, `TRUE` or `HIGH` state and is denoted by the binary digit "1". "Voltage NOT present", conversely, is a `NO`, `FALSE` or `LOW` state denoted by the binary digit "0".

A digital device detects whether voltage is at the input or not. For the output of the logic-AND to be equal 1, all of the inputs must have voltage on them. With only two states (voltage present or voltage NOT present), it is not possible to ever have more than just voltage or to have less than no voltage. The two-state system offers only the alternative of "ALL" or "NOTHING". Therefore, voltage on an input = 1, no voltage = 0. The AND-function with two inputs produces the following output.

1.3.1 Truth Tables

A so called **truth table** can be constructed to determine the output given the various input values. Since there are only two states in the binary system, the number 2 raised to the n power provides us with the number of combinations possible. Assume that n equals the number of inputs. With two inputs, A and B, there would be two squared = 4 possibilities. Given the variables A, B, C, two raised to the third power = 8 combinations, etc.

In a truth table for the two inputs A and B, there are four possibilities, two zeros and two ones for each variable as shown below. Note the output (result) is only 1 when $1 \cdot 1 = 1$ because $0 \cdot 0 = 0$, $0 \cdot 1 = 0$, $1 \cdot 0 = 0$. See Table **??**.

Inputs		Output
A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

Table 1.1: Logic-AND truth table for two inputs

The logic that corresponds to the AND-truth table for two variables can be seen in the statement: **"If the sun is shinning AND the temperature $\geq$ 80 degrees, go swimming"**, mentioned at the beginning of section **??** on switching circuits. When both situations, A and B, are given, the condition is true, and a "1" appears in that position in the truth table. Note the following logic-AND truth table for three inputs, Table **??**.

Inputs			Output
A	B	C	ABC
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Table 1.2: Logic-AND truth table for three inputs

A truth table for two or more inputs can also be constructed for the logic-OR in the same manner as for the logic-AND.

The binary logic for the logic-OR is as follows.

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 1 \end{aligned}$$

The logic statement given at the beginning of section **??** corresponding to the OR logic was: **"If it is raining OR snowing, stay at home"**. In this case, if one of the situations is given, A OR B, the condition is true and a "1" appears in those places in the truth table (if it's both raining and snowing, you also stay home). Note that the logic-OR requires that at least one input equals 1 for the output to be equal 1 as shown in the truth table for three inputs, Table ??.

Inputs			Output
A	B	C	ABC
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Table 1.3: Logic-OR truth table for three inputs

1.4 The Design Procedure

To become familiar with the procedure of designing circuits, we suggest that you follow the steps given below.

1. State the problem verbally.
2. Sketch a circuit diagram to help visualize the problem (it is advisable to use a different color for each path)
3. Denote variables (inputs and output) with letters.
4. Find the combinations that make the statement true (a truth table is helpful)
5. Express the function algebraically.

The following example illustrates the use of NOT, AND, and OR logic combined and shows how the design procedure can be used.

Example 1.2

A long hallway has a door at each end. The light needs to be turned on when a person enters either door and switched back off upon exiting either door. Find the appropriate solution.

Step # 1

If both switches are in the `"off"` position before entering the front door and we throw the switch at that door to turn the light on, the front door switch would be `"on"` while the back door switch would be `"off"`. Conversely, however, if we entered the back door and did the same thing to turn the light on, the back door switch would be `"on"` while the front door switch was still `"off"`.

There are only two possibilities to turn the light `"on"` depending on which door is used.

```
Either the front switch = 1 and the back switch = 0
or the back switch = 1 and the front switch = 0.
```

Step # 2
Draw a circuit diagram of the switches showing the two possibilities; see Figure **??**:

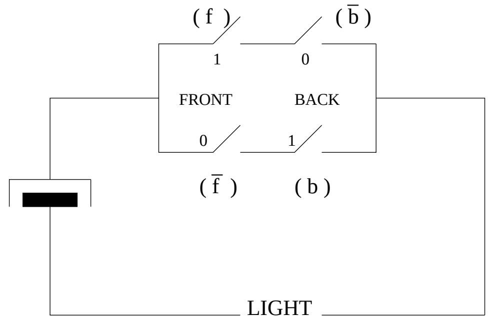


Figure 1.8: A circuit diagram of the switches showing the two possibilities

Step # 3
Variables:
f = front switch on, $\overline{f}$ = front switch off,
b = back switch on, $\overline{b}$ = back switch off,
L = light on.
Step # 4
A truth table can be constructed to help find the combinations that make the statement true. With only the two possibilities (true/false, on/off, 0/1), the base is 2 and with only 2 switches (or variables) two squared = 4 combinations, Tables **??**, **??**.

Inputs		Output						
f	b	fb	$\overline{f}\,\overline{b}$	$\overline{f}b$	$f\overline{b}$	$\overline{f}+b$	$f+\overline{b}$	$fb + \overline{f}\,\overline{b}$
0	0	0	1	0	0	1	1	1
0	1	0	0	1	0	1	0	0
1	0	0	0	0	1	0	1	0
1	1	1	0	0	0	1	1	1

Table 1.4: Verification of the Switch Truth Table

Inputs		Output		
f	b	$(f+b)(\overline{f}+\overline{b})$	$(\overline{f}+b)(f+\overline{b})$	$(\overline{f}b)+(f\overline{b})$
0	0	0	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	1	0

Table 1.5: Verification of the Switch Truth Table, continued

Step # 5
Express the function algebraically:

`(front=1 and back=0) or (back=1 and front=0)`

can be expressed as:

$$(f\overline{b}) \; + \; (b\overline{f}) \quad = \quad L$$

Beginning with step # 1 "perfect induction" was applied which was then illustrated very clearly by the truth table. The algebraic method, however, offered the quickest possible solution. The exercises will provide the opportunity to practise the design procedure until the reader becomes proficient at it. In the following sections, we will add an additional step to it: logic devices. Subsequently, we will deal with algebraic methods and other possibilities to simplify designing. Switching circuits should only serve as an introduction into digital circuits. With time and experience, it may not be necessary to follow all the steps outlined but the reader may find them helpful, especially in the beginning.

1.5 The Logic Gates

According to the mathematical definition, a **function** has a **domain,** a **rule,** and a **range**. Often teachers use an analogy of a machine to explain functions. In the example, the machine takes members of the domain, applies the rule, and produces unique members of the range. This applies to digital functions as well. To perform the logic-AND function, a rule must be applied that will insure that the output is only then equal to logic-1 when all the inputs equal logic-1. The rule for the logic-OR function must be that the output will equal logic-1 when any input equals logic-1. The machine that is used to perform these operations on digital logic functions is called a **LOGIC GATE.** The gate takes the input, applies the rule, and produces the appropriate output accordingly.

In the preceding section, we introduced logic functions, truth tables, and design procedures using switches as examples. In reality, however, in digital circuits, these switching functions are performed by logic gates.

AND-GATE
The example:
"If the sun is shining AND the temperature $\geq$ 80 degrees" is an AND function and requires an AND-gate as shown in Figure **??**.

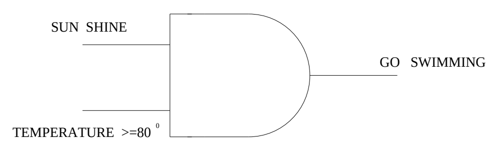


Figure 1.9: Logical AND gate

OR-GATE
The example:
"If it is raining OR snowing" is an OR function and requires an OR-gate as shown in Figure **??**.

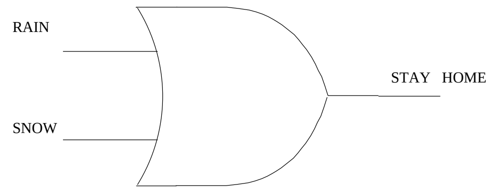


Figure 1.10: Logical OR gate

Logic gates are actually circuits themselves, blocks of hardware that produce a logic-1 or logic-0 output signal if the input requirements are satisfied. One single AND gate could be constructed using two diodes to implement the function $AB = F$ as shown in Figure **??**.

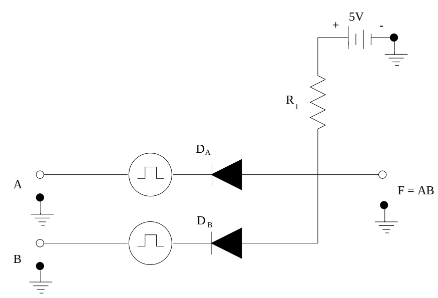


Figure 1.11: Diodes used for the AND logic function $F = A \cdot B$

The anode shown is connected to +5V through $R1$. The diodes conduct forward as long as there is no voltage at the inputs A and B. In this state, the output F is pulled down to a low level "0". When a high level "1" in the inputs A and B makes the cathode positive, the diode is cut off and the output F is high, "1". Even with one input high and the other input low, the low input will cause the output to be pulled down to a low, "0" state.

As for circuit designers, today, a gate's internal circuitry is of less interest, because the gates are prefabricated and supplied to the designer in hardware packages. At the advent of digital circuitry, however, design was much more sophisticated and complicated. The ABC and other one-of-a-kind computers such as the UNIVAC I and its successors built in the early and mid-1950s had vacuum tubes as their basic components. The construction of only one gate could have taken up as much as four square inches of space and everything had to be hand wired. In 1947, scientists at Bell labs invented a tiny, solid-state device called the **transistor** (short for **transfer resistor**). Made of semiconducting materials, the transistor could act as a switch controlling the ebb and flow of electrons. In the late 1950s, transistors replaced vacuum tubes as the major components. Transistors could be mounted closely together, saving space, on circuit boards. New technologies minimized and refined components even further. With time, 100,000 and more transistors, diodes, resistors, and capacitors - the equivalent of more than 10,000 complete circuits- could be placed on a single silicon chip less than one eighth of an inch square. Circuits made in this manner are called **integrated circuits**, or **ICs**. Today, the following packages of gates are, therefore, available to be used by circuit designers.

Type	Number of gates
Small-Scale Integration (SSI)	10 - 12
Medium-Scale Integration (MSI)	up to 100
Large-Scale Integration (LSI)	up to 1,000
Very large-Scale Integration (VLSI)	up to 50,000

In the logic gate, various components are interconnected to form an electronic circuit. It is then mounted on a metal or plastic package, and connections are welded to external pins to form the IC **??**. Integrated circuits differ from other electronic circuits composed of detachable components in that individual components in the IC cannot be separated or disconnected. The circuit inside the package is accessible only through the external pins. Therefore, it is this package of logic gates that the digital circuit designer works with when designing circuits.

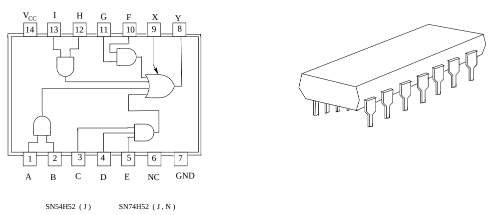


Figure 1.12: SN5432, SN5408, TTL IC with four gates

It is of utmost importance to memorize the symbols used to represent the various gates. These are, however, only symbols used for design purposes. Gates are so small that it is impossible to detect them with the naked eye. The chances are that they do not resemble these symbols.

1.6 Combinational Networks

To illustrate the combinational circuit, we revert to the "menu-circuit" example. This time, however, we have appropriately constructed the circuit using AND and OR gates. An AND gate is placed at the output "Meal" because three inputs will be true to make the output true. Entree is a direct input into the AND gate but soup and salad are inputs to an OR gate because a choice of one or the other is given. The dessert function is also performed by an OR gate because the choice of one out of three items is possible.

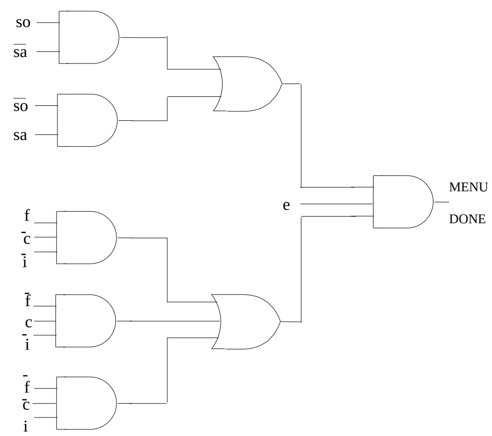


Figure 1.13: Combinational circuit using AND and OR gates

In the logic gate, various components are interconnected to form an electronic circuit. In the design procedure, we found, by perfect induction and using the truth table, the function to solve the problem of designing a double-throw switch. It was expressed algebraically as:

$$(f\overline{b}) + (b\overline{f}) = L$$

Once the function is known, the 6th design procedure step is then to draw the logic diagram using gate symbols. When, as in this case, the inputs and output are known, but the circuit needs to be shown, we say that we "synthesize a circuit" or "realize the function." To do this, we usually begin at the output and work backwards to the inputs. The circuit for the double-throw switch is shown in Figure **??**.

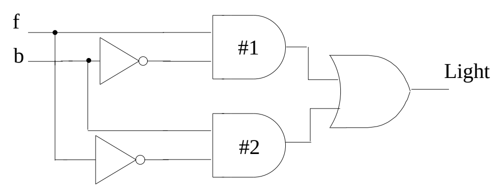


Figure 1.14: Double-throw switch circuit using AND and OR gates as well as inverters

Note that there is an OR connecting two ANDs in the function $(f\overline{b}) + (b\overline{f}) = L$. This indicates that there is an OR gate at the output. The OR gate at the output has two inputs, each of which is the output of an AND gate. Therefore, AND gate 1 and AND gate 2 are placed behind the output OR gate. The inputs into AND gate 1 are f and $\overline{b}$ and the inputs into AND gate 2 are b and $\overline{f}$.

NOT-gate
INVERTER
The complement of f, $\overline{f}$, and of b, $\overline{b}$, are used here for the first time and allow us to introduce the gate that performs this function, the **NOT gate** or**inverter**. Note that there is an inverter between b and the input of AND gate 1 as well as between f and the input of AND gate 2. The inverter is used in order to change the values of f and b to their opposite values.

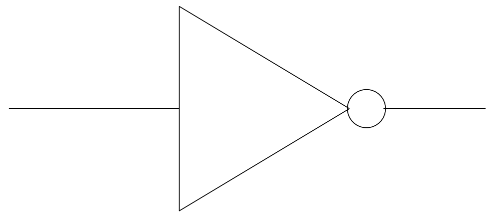


Figure 1.15: Not-gate or inverter

The logic statement in section **??** corresponding to the NOT logic was: **"If it is NOT snowing, go to work"**.

Note the **"bubble"** at the output of the inverter that denotes the "NOT" function. A line is also drawn over the variables f and b to show that their values have been complemented.

NAND gate
NOR gate
It is, however, not always necessary to use a NOT gate in the design of digital circuits when we want to complement the value of the output. This feature is also conveniently incorporated in the AND and OR gates. By placing the NOT feature at the output of these gates, we complement the value. Whatever the value would have been, is now the opposite. In this case, complemented AND gates are called **NAND gates** and complemented OR gates are called **NOR** gates.

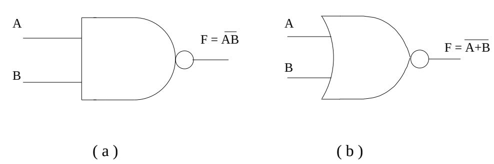


Figure 1.16: (a) A two-input NAND gate, (b) a two-input NOR gate

For comparison, the AND, OR, NAND, and NOR gates are listed in Figure **??** with input variables A and B and output F. AND, NAND, OR, and NOR gates, of course, may have more than two inputs. The multiple input AND gate responds with a logic-1 output if all of the inputs are logic-1. The output produces a logic-0 if any input is logic-0. The NAND gate, conversely, responds with a logic-0 output if all of the inputs are logic-1 and produces a logic-1 if any input is logic-0. The multiple input OR gate responds with a logic-1 output if any of the inputs is logic-1. The output produces a logic 0 only if all the inputs are logic-0. The NOR gate responds in the opposite way, with a logic-0 if any of the inputs are logic-1 and produces a logic 1 only if all of the inputs are logic-0.

A logic NAND statement would be, for example, `"If it is NOT snowing or it is NOT raining, go swimming"`. If A or B is a NOT situation, the condition is true and a "1" appears in the truth table at that location. This gate can be also described as follows: **"If it is not true that it is both snowing and raining, go swimming"**.

A logic NOR statement would be, "If it is NOT snowing and it is NOT raining, go swimming". In this case, both A and B must be a NOT situation to be true and for a "1" to appear at that place in the truth table.

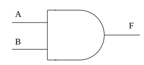
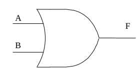
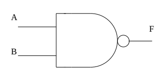
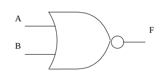
NAME of GATE	GRAPHIC SYMBOL	ALGEBRAIC FUNCTION	TRUTH TABLE
AND		$F = AB$	A B F 0 0 0 0 1 0 1 0 0 1 1 1
OR		$F = A + B$	A B F 0 0 0 0 1 1 1 0 1 1 1 1
NAND		$F = \overline{AB}$	A B F 0 0 1 0 1 1 1 0 1 1 1 0
NOR		$F = \overline{A + B}$	A B F 0 0 1 0 1 0 1 0 0 1 1 0

Figure 1.17: Digital logic gates

The input signals A and B in the two-input gates shown above may exist in one of four possible states: 00, 10, 11, 01. These input signals are shown below in a so called **timing diagram** together with the output signals for the inverter, AND, and OR gates. The timing diagram reflects the same information obtained from the truth table for two inputs showing the response of each circuit to each of the four possible input binary combinations. The high portion of the line is representative of "1" in the output while the low portion stands for "0" being produced.

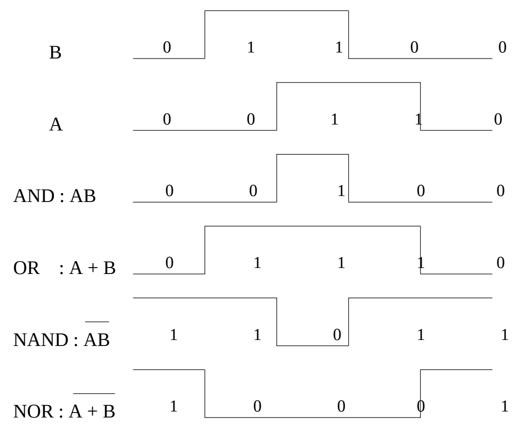


Figure 1.18: Timing diagram. Input-output signals for gates

1.7 Analysis of a Circuit

Given a circuit diagram in which the inputs are known, it can be **traced** and **analyzed** to ascertain the function. The procedure is to divide the circuit into levels starting with level one at the output gate. The gates providing the output gate with input would then constitute the second level, their input gates would then be the third level, etc. The circuit in Figure **??** illustrates this procedure.

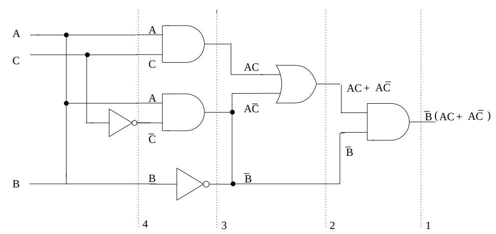


Figure 1.19: Logic circuit gate levels

Begin at the input of the first path, note the output after the gate and repeat the same for each path in that level. Proceed to the next level, note the outputs, etc. Continue until only one combined result is obtained at the output.

Problem

Analyze the circuit from Figure **??**.

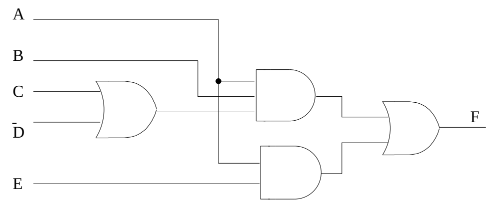


Figure 1.20: Logic circuit showing gate levels

Problem
Analyze the circuit from Figure ??. Observe that some gates are on both even and odd levels, so they play different functions in the circuit, so that they must be copied first to allow the circuit's analysis.

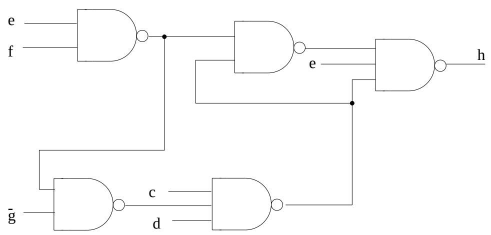


Figure 1.21: Logic circuit that must be transformed by copying gates before analysis of levels

1.8 Exclusive-Or (EXOR) Gates

EXOR gate
The OR gate that we have used up to this point is actually comparable with what we would call "inclusive or logic". This is because it produces a TRUE at the output in case one or the other, any or all, of the inputs are TRUE. The following example illustrates this:

`If you are thirsty, you can drink coffee, tea, water, any combination thereof, or all of them.`

There are situations, however, when a choice is given but only one can be chosen, not both or all. For example:

`"Jane can choose to marry Tom or Dick but she cannot marry both."`

This function would be: (Tom, NOT-Dick) or (NOT-Tom, Dick)

$$(T\overline{D}) + (\overline{T}D)$$

The gate used for this function is the EXOR gate or exclusive OR. It is denoted by the ring $\oplus$ sign.

$$(T \oplus D)$$

Compare the above statement with the EXOR truth table shown in Figure **??**.

A	B	$A+B$	$A \oplus B$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

Table 1.6: OR and EXOR truth tables for two variables, please compare

For the statement to be true and for a "1" to appear in that position in the truth table, one of the following conditions must prevail.

- `If NOT Tom and if NOT Dick, Jane cannot marry.`
- `If NOT Tom but Dick, Jane can marry.`
- `If Tom but NOT Dick, Jane can marry.`
- `If both Tom and Dick, Jane cannot marry.`

Recall the double-throw switch problem. We used an OR gate, two AND gates, and two inverters for the circuit but this was also a $(\overline{A}B) + (A\overline{B})$ situation. If the switch at the front door was on, the switch at the back door had to be off, or vise versa. An EXOR gate should be used in such a situation. By doing so, hardware can be saved, which is a top priority in circuit design. The truth table from **??** compares the OR and the EXOR gate for two variables. The symbol of EXOR gate is shown in Figure **??**.

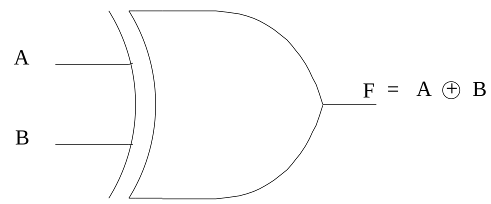


Figure 1.22: EXOR gate with two inputs

This can be extended to multiple variables. The truth table for the EXOR function with three variables shows that when the inputs are all the same (all zeros or all ones), the output = 0. More precisely stated: An **EVEN** number of ones in the input produces an output = 0. An **ODD** number of ones in the input produces an output = 1.

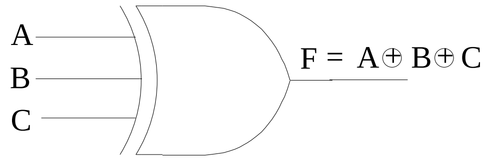


Figure 1.23: EXOR truth table for three variables: (a) symbol

Inputs			Output
A	B	C	$A \oplus B \oplus C$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Table 1.7: Logic-EXOR truth table for three inputs

1.9 Operational Devices

In the introductory chapters, we will be concerned with the design of what we call **Combinational Circuits** which serve as an introduction into the studies of digital circuits and are followed by the study of **Sequential Circuits**. The main difference between these two areas of study is that, in combinational circuits, the inputs alone determine the output of the function while, in **sequential circuits**, the present state together with the inputs determine the output. Sequential circuits provide us with memory and storage possibilities and are able to perform such functions as finding the maximum element or sorting. Finding minimum element in a vector will be shown in the following example.

for i:= 2 to last do
$iflist[i] < list[i-1]then$
$MIN := list[i]$

For the time being, however, we direct our attention towards combinational circuits. Interestingly, some calculations specified by flowcharts can be realized both by combinational and sequential circuits. The minimum circuit specified above belong to them. It is called an **iterative circuit**. Such circuits are very important, they constitute the heart of every arithmetic computer, and as you will see, they are also at the base of logic machines.

Designing digital circuits also requires that we consider how to solve problems involving addition, subtraction, comparison, code conversion, etc. Devices are needed for comparison operations such as:

$X = 3.0$
$X + Y = Z$

$A < B$
$D > C$,
etc.
Decimal adders, code converters, etc. are also needed since we communicate with computers through the use of decimal digits and alphabetical letters which must be converted to binary numbers.

Consider the loop and flow chart in Figure ?? and try to visualize what kinds of devices would be needed to perform these functions.

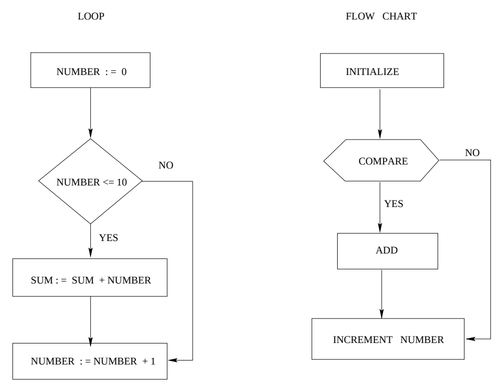


Figure 1.24: Comparison of a program loop, a flowchart, and an iterative circuit

One of the devices needed is a **magnitude comparator**. One binary number of four bits can then be compared to another binary number of four bits and provide the results:
$(A = B)$ or
$(A < B)$ or
$(A > B)$ or
$(A \geq B)$ or
$(A \leq B)$.
Signals that are a result of comparison operation are examples of predicate signals. Such signals play important role in every processor composed from sequential circuits (co-called **Finite State Machines**) and data path circuits.

Before beginning with the practical application of designing a magnitude comparator, let us first gain some insight into the **binary (base-two) number system**.

1.10 Binary Number System

The **radix** or **base** for the **binary number system** is two because two different digits (zero and one) are used. The **base two** number 101 (one, zero, one to base two) is such a number and is equal to the decimal number 5 (five to base ten). The **base ten** system uses the ten digits 0,1,2,3,4,5...9. In the **decimal system**, the **least significant digit** (starting from the right) is in the zero position. The second digit to the left is in position one, the third digit to the left is in position two, etc. To obtain the value of a decimal number, ten is raised to the power of the position number and then multiplied by the digit in that position. The various positions are then added together.

Example 1.3

The number 2,426 (to base ten) is arrived at as follows.
6 is in the least significant position: the position zero.
2 is in the first position to the left: position one.
4 is in the second position to the left: position two.
2 is in the third position to the left: position three.
(10 raised to the position $0 = 1) \cdot 6 = 6$
(10 raised to the position $1 = 10) \cdot 2 = 20$
(10 raised to the position $2 = 100) \cdot 4 = 400$
(10 raised to the position $3 = 1000) \cdot 2 = 2{,}000$

$2{,}000 + 400 + 20 + 6 = 2{,}426$ (to base ten).
Similarly, to calculate the decimal value of the binary number 1011, two (instead of ten) is raised to the power of the position number and multiplied by the digit in that position. The various positions are then added together.

Example 1.4

1011(to base two)
1 is in the least significant position: position zero
1 is in the first position to the left: position one
0 is in the second position to the left: position two
1 is in the third position to the left: position three
(2 raised to the position $0 = 1) \cdot 1 = 1$
(2 raised to the position $1 = 2) \cdot 1 = 2$
(2 raised to the position $2 = 4) \cdot 0 = 0$
(2 raised to the position $3 = 8) \cdot 1 = 8$

$1 + 2 + 0 + 8 = 11$ (to base 10).
See Figure ?? for summary of these operations.

Figure 1.25: Express the binary number 0010 1010 as a decimal number

```
position:  7     6     5     4      3     2     1     0

         -----------------------  -----------------------
          128    64    32    16     8     4     2     1
           *0    *0    *1    *0    *1    *0    *1    *0

          ---   ---   ---   ---   ---   ---   ---   ---
            0 +   0 +  32 +   0 +   8 +   0 +   2 +   0 = 42
  binary    0     0     1     0     1     0     1     0
```

With time, you will recognize that the values of the binary positions (from right to left) are: 1, 2, 4, 8, 16, 32, 64, etc. (i.e. the number two raised to the power of 0, 1, 2, 3, 4, 5, 6,

decimal	binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Table 1.8: Binary numbers from 0000 to 1111. Please analyse pattern of changes of each bit.

etc.) There is no limit to the number of **bits** (short for **binary digit**) possible. As you will see, the next value is always double the previous value. If a one is in position 5, the value of 16 is added because $2^4 = 16$. If that position contains a zero, no value is added. Since one position represents one binary digit, or bit, it is one bit of data, or information.

It is then easy to see the decimal equivalents of the following binary numbers, Table **??**.

Another conversion example is shown in Figure **??**.

Figure 1.26: Express the decimal number 69 as a binary number

```
position:  7     6     5     4      3     2     1    0

         ------------------------  ---------------------
          128    64    32    16     8     4     2     1
           *0    *1    *0    *0    *0    *1    *0    *1

          ---   ---   ---   ---    ---   ---   ---   ---
            0 +  64 +   0 +   0 +   0 +   4 +   0 +   1 = 69
 binary     0     1     0     0     0     1     0     1
binary 0100 0101 = 69(to base ten).
```

The above method of converting decimal to binary is, however, impractical; especially when dealing with larger numbers. The number 69(to base ten) is more easily converted to binary by

- dividing the number by "2" and
- taking the remainder to be the "LSD" of the binary number.
- the result is then also divided by "2" and

- its remainder is the next binary digit, etc.

Each result is divided by "2" and the remainder equals the next binary digit until " 0/2 " as shown in Figure ??.

```
divide by "2"      result        remainder
69/2          =      34     +        1    (LSD)
34/2          =      17     +        0
17/2          =       8     +        1
 8/2          =       4     +        0
 4/2          =       2     +        0
2/2           =       1     +        0
1/2           =       0     +        1
0/2           =       0     +        0    (MSD)
```

Figure 1.27: Convert 69(to base ten) to binary:

Binary equivalent of 69(to base ten): 0100 0101

It may be of interest to note that in Pascal programming the "div" instruction performs the above division by "2" while the "mod" instruction provides the binary digits.

Convertion of 2,426(to base ten) to binary is shown in Figure ??.

```
2,426/2  =  1213  +  0
1213/2   =   606  +  1  {binary 1=     2}
 606/2   =   303  +  0
 303/2   =   151  +  1  {binary 1=     8}
 151/2   =    75  +  1  {binary 1=    16}
  75/2   =    37  +  1  {binary 1=    32}
  37/2   =    18  +  1  {binary 1 =   64}
  18/2   =     9  +  0
   9/2   =     4  +  1  {binary 1 =  256}
   4/2   =     2  +  0
   2/2   =     1  +  0
   1/2   =     0  +  1  {binary 1 = 2048}
   0/2   =     0  +  0
```

Figure 1.28: Convertion to binary: Binary 0000 1001 0111 1010 = 2,426(to base ten).

Note that the binary number is written in **"nibbles"** with four bits to one nibble and that each position contains one bit which is equivalent to one input of a logic gate. The nibble to the far left in the above number "2,426" was written "0000" because, in this case, the last operation started a new nibble but was a zero. Therefore, all the other digits in the nibble are also zeros. These, however, need not be written since they contribute nothing to the value of the number. It is sufficient to write 2,426(to base ten) = 1001 0111 1010

A sequence of eight bits is called a **"byte"**.

1.11 Designing the (=) Function of the Comparator

As we have seen in the loop example before, comparison of variables is often necessary. For instance, the variable A (10 or binary 1010) will be compared with the variable "number" (B) as it is added up with each pass, and until $A = B$, which means until 10 passes have been made and B (number) also equals 10 (or binary 1010). Since A is a four bit binary number (1010), four bits (or positions) for the four digits are needed in the comparator. Each bit in A must be compared with each bit in B on each pass.

Therefore, in descending order, the Table **??** can be created.

```
Position number:          3   2   1   0
                          --------------
                   A =   A3  A2  A1  A0
                   B =   B3  B2  B1  B0
```

Figure 1.29: Table

Starting at B (number) $= 0$, 10 passes will need to be made. The comparisons are then as in Figure **??**.

```
Position number:          3   2   1   0
                          --------------
START             A   =   1   0   1   0
                  B   =   0   0   0   0

Pass:   #1        B   =   0   0   0   1
        #2        B   =   0   0   1   0
        #3        B   =   0   0   1   1
        #4        B   =   0   1   0   0
        #5        B   =   0   1   0   1
        #6        B   =   0   1   1   0
        #7        B   =   0   1   1   1
        #8        B   =   1   0   0   0
        #9        B   =   1   0   0   1
       #10        B   =   1   0   1   0
A = B
```

Figure 1.30: Positions

Constructing the "equal" portion of the comparator

In the comparator, each bit in A must be compared to each bit in B. Therefore, for a four bit number, four double-throw switches would be needed. One double-throw switch circuit for each bit comparison is needed. Visualize A as representing the front door switch and B as representing the back door switch. An inverter is placed after the output of the OR gate, however, so that when there is a match (both A and $B = 1$ or 0), there will be a mismatch at the inputs of both AND gates. This results in an output at the AND gates of "0". Two "0" inputs at the OR gate produce a "0" in its output, but the inverter converts this "0" to "1" for a match and ($A = B$).

When the inputs A and B do not match (for example, $A = 1$, $B = 0$), the inputs at one AND gate will be 00 and 11 at the other, which produces an input into the OR gate of 01. The OR gate output is then "1", but the inverter converts it to a "0" for a mismatch and A is not equal to B, Figure **??**.

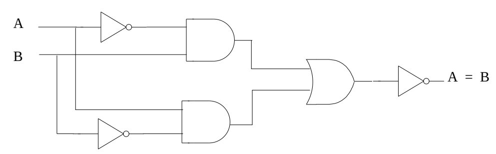


Figure 1.31: EXOR implementation for the double-throw switch circuit

EXOR implementation for the double-throw switch circuit

Recall, however, that it is not necessary to use a double-throw switch circuit for such a design. The EXOR gate could be used with the inverter, as shown below, for the same purpose of saving hardware, Figure **??**.

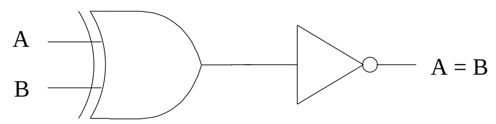


Figure 1.32: EXNOR gate created from EXOR and inverter

1.12 The EXNOR Gate

On the other hand, the EXOR with inverter is also not needed. The EXOR function also has a complemented form: the EXNOR, also called equivalence, equality or coincidence, Figure **??** and Table **??**.

A	B	$\overline{A \oplus B} = A \odot B$
0	0	1
0	1	0
1	0	0
1	1	1

Table 1.9: EXNOR truth table

The combination of EXOR gate and inverter can be then replaced by the EXNOR gate.

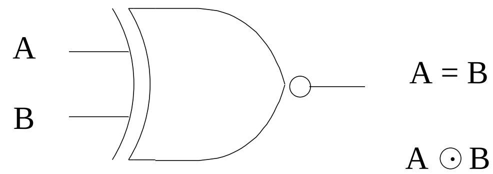


Figure 1.33: EXNOR gate symbol

Therefore, in order to compare all four bits, four EXNOR gates could be used for the (=) function. Their outputs would be connected to the input of an AND gate and when all the AND gate inputs were "1", its output would = 1 for a match and $A = B$.

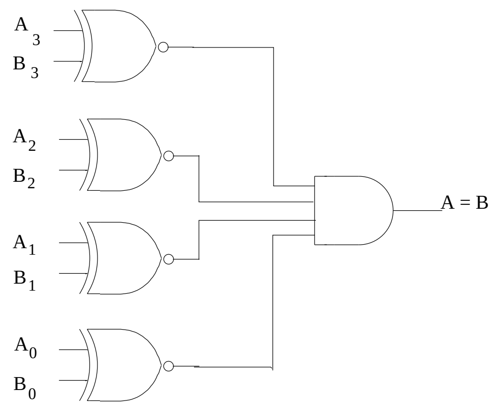


Figure 1.34: Four EXNOR gates with one AND gate serve as a comparator of equality

When we use the term "EXNOR" in this text, the function is denoted by simply complementing the EXOR. When the EXNOR, however, is referred to as an **equality**, a **coincidence**, or an **equivalence** it is usually denoted by a ring/dot sign.

1.13 The Inhibition and Implication Functions

After having introduced the EXOR and EXNOR function, we are able to acquaint you with two further functions, the inhibition and the implication functions. The truth table for two binary functions is shown in Table **??** below with the EXOR, EXNOR, inhibition and implication functions for comparison. Note that (inhibition + inhibition) = EXOR, $a\overline{b} + \overline{a}b$, while (implication)(implication) = EXNOR, $(a + \overline{b})(\overline{a} + b)$.

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

Table 1.10: Truth tables of inhibition, EXOR, implication, and EXNOR functions

The inhibition and implication functions are realized with AND-gates, respectively OR-gates.

1.14 The Complete Magnitude Comparator

The (=) comparison function alone, however, is not enough to constitute a complete **magnitude comparator**, $A < B$ and $A > B$ are also needed. EXOR gates and EXNOR gates are, therefore, not used for this purpose. The double-throw switch circuit actually is used as the basis for this device. Observe the SN54L85, 4-bit, magnitude comparator supplied by Texas Instruments, as shown in Figure **??**.

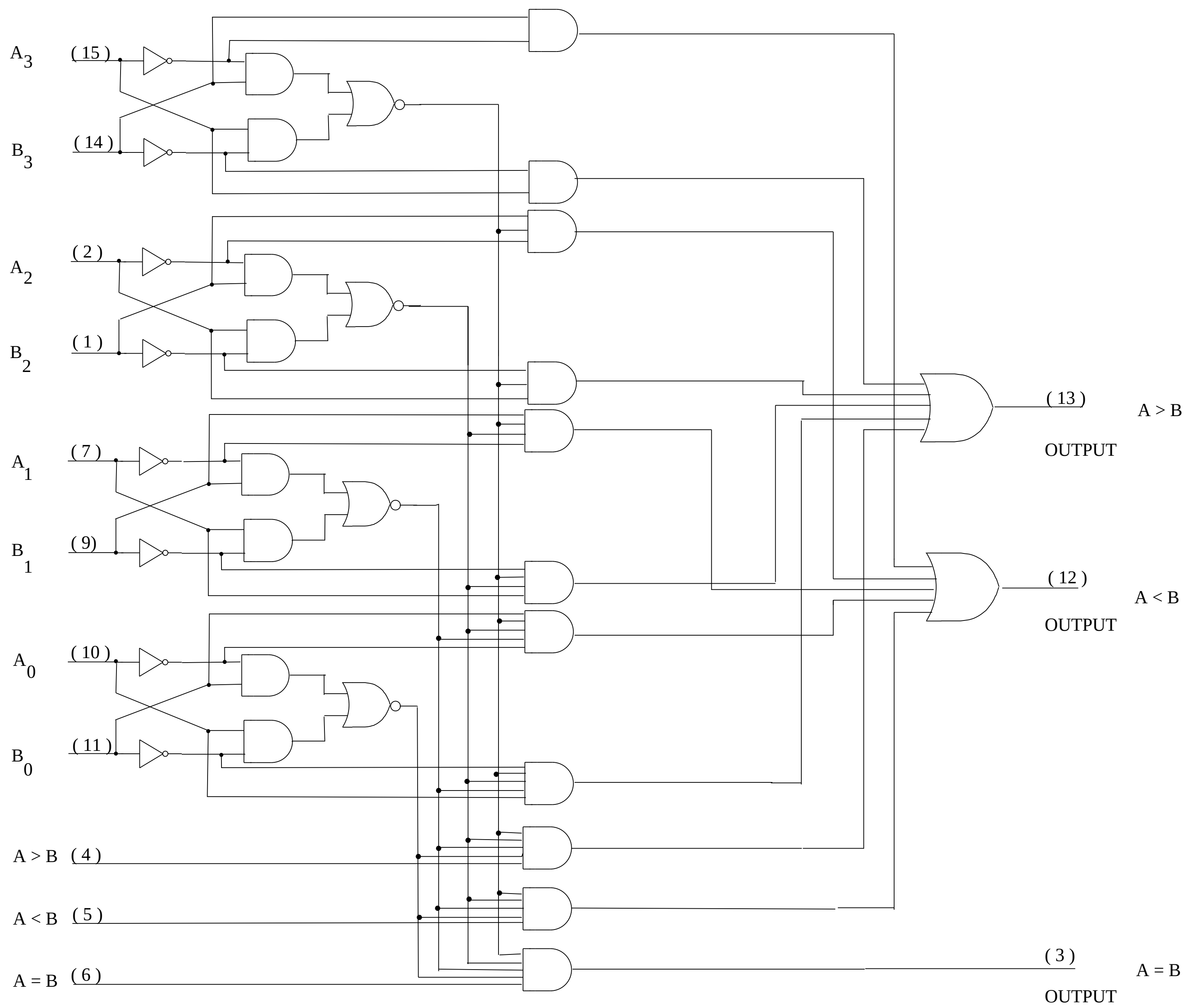


Figure 1.35: Integrated Circuit SN54L85

As you can see, in order to perform the functions $A < B$ and $A > B$, the double-throw switch circuit is used, but it is tapped into before the AND gates to provide input to two other AND gates. The output of one of these AND gates then leads to the $A < B$ function while the output of the other one connects to the $A > B$ function. Let us analyze this device to see how these functions are accomplished. Note the circuit A_0, B_0, for example. Both AND gates tap into the function at $\overline{A}$ and B as well as at $A\overline{B}$. They are also connected to the outputs of the other three circuits. Therefore, when the other three bits match, the three AND inputs = 1. If $B_0 = 0$ and $A_0 = 1$, all AND inputs = 1, the AND gate output then

= 1, the connecting OR gate output = 1, and $A > B$. If $B_0 = 1$ and $A_0 = 0$, all the AND inputs = 1, the connecting OR gate output = 1, and $A < B$.

The Texas Instruments 4 bit magnitude comparator, SN54LS85 is also shown in Figure **??** to illustrate a second possibility of achieving this function.

Analyze this circuit

Observe the top view of SN54L85. As you can see, $A = B$ output can be wired from pin 3 while the $A < B$ and $A > B$ outputs are taken from pins 12 and 13 respectively.

1.15 The Integrated Circuit Logic Families

There are various logic families of digital ICs available commercially but it is not in the scope of this text to discuss their properties in detail. It would, however, be beneficial for the serious beginning student of digital circuits to have, at least, a TTL (transistor-Transistor logic) data book for reference. Then, with the acquisition of a few basic ICs the practical application of such devices could be experimented with. In such a laboratory setting, a power supply and voltage regulator would be needed so that the required voltage could be applied to the circuit. By probing the outputs with a logic probe, the circuit can be tested for the proper values.

The following packages should enable you to build most simple circuits.

Packages containing:

4 AND and 4 NAND gates with 2 inputs each
2 AND and 2 NAND gates with 4 inputs each
4 OR and 4 NOR gates with 2 inputs each
2 OR and 2 NOR gates with 4 inputs each
6 inverters
4 EXOR gates with 2 inputs each

Three such packages are shown in Figure **??** below. Consider how they can be wired to build the previously described comparator.

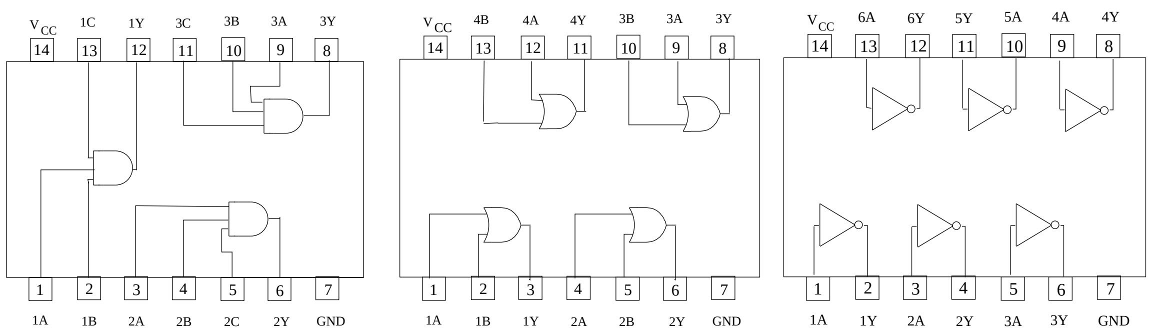


Figure 1.36: Integrated Circuits: SN5486, SN5416, SN541S21

For comparison, the AND, OR, NAND, NOR, NOT, EXOR, and EXNOR gates are listed below with their graphic symbol, algebraic function, and truth table. The EXOR truth table shows that an ODD number of ones in the input produces an output = 1, but an EVEN number of ones in the input produces an output = 0. The truth table for the EXNOR is the opposite. An ODD number of ones in the input produces an output = 0. An EVEN number of ones in the input produces an output = 1 in the EXNOR gate.

1.16 Boolean Algebra

1.16.1 Introduction

Throughout the ages, there have been various number systems. We are used to the decimal system with 100 cents to a dollar or 100 cm to 1 meter, but the old Egyptian base 12 number system, for example, is also still prevalent in our time: 12 to a dozen, 12 inches to a foot, etc. Up until a few years ago, the British still had a money system that originated from the base 12 number system. There were 12 pence to a shilling and 20 shilling to a pound. When the decimal system was introduced in the money system in Great Britain, some people had a lot of difficulty adapting. Even new adding machines were needed. In the old machines, when one pence was added to eleven pence, one shilling appeared. Fifteen shilling and seven shilling did not add up to 22 shilling but to 1 pound and 2 shilling. This illustrates, to some extent, that there are different systems with different rules, even if they have some similarities. Up till now in this text, we have dealt with the decimal and the binary number systems. Later on, we will also look into the octal (base 8) and the hexadecimal (base 16) number systems but for the present we want to concentrate on the base 2 system as well as the algebra that is associated with it.

The development of the binary number system is contributed to **Gottfried Leibniz** (1646 - 1716). It was little noted in his age, however, because there was little practical use for it. Then in 1854, **George Boole** introduced a systematic treatment of logic which also used two symbols. His system did not gain attention early, it remained practically unnoticed until **E.V. Huntington** in 1904 formulated postulates for it. **Claude E. Shannon** - an American mathematician - then made a major application of Boolean Algebra to switching circuits in 1938. (By the way, it was a Master Thesis. I wish your M.S. theses will be also useful !!) The links between number systems, Cartesian Products and logic (binary and multivalued) will be discussed in more detail in next chapters.

Shannon demonstrated that the properties of bistable electrical circuits can be represented by this algebra. In the historical development, in chapter **??**, we will present that the use of this system along with the use of vacuum tubes for the first time in the ABC and ENIAC computers ushered in the electronic digital computer age. The Boolean Algebra applicable to two-state systems is the algebra used in this text with reference to such switching circuits. Next, we will other useful algebras.

Many of the same rules apply to Boolean Algebra as they do in ordinary algebra. All of the postulates and theorems that apply are tabulated after the proofs that will follow. If you are a beginner, caution is recommended not to confuse the rules of ordinary algebra with those of Boolean Algebra, applying them inappropriately.

The "dot" multiplication (*) as well as the plus (+) sign are only used here to help those already familiar with ordinary algebra to be able to relate to the Boolean Algebra manipulations better. Actually, the conjunctive form denoted by the plus (+) sign and called "sum" or "join" represents the OR-ing of terms. The normal form can also be expressed as `(c OR e)` or `(c + e)` while the complemented form is expressed as `(c NOR e)` or $\overline{(c + e)}$. Conversely, the "product" or "meet" is the disjunctive form denoted by the "dot" multiplication sign, which represents the AND-ing of terms. The normal form can be expressed as `(c AND e)`, or `(c * e)`, or `(ce)` while expressing the complemented form as `(c NAND e)`, `NAND(c, e)` or $\overline{ce}$.

1.17 Binary Boolean Algebra

Binary Boolean Algebra is based on the two binary digits "1" and "0". Previously, we showed that in the OR function voltage present ("1") at any input would produce voltage ("1") at the output. In the AND function, voltage "1" must be present at all inputs before voltage "1" will be produced at the output. Given a truth table for the variables A and B, the following, Table **??**, applies:

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

Therefore,

$$A + 1 = 1 \qquad (if\ A = 0, 0 + 1 = 1)(if\ A = 1, 1 + 1 = 1) \tag{1.3}$$

$$A + 0 = A \qquad (if\ A = 0, 0 + 0 = 0)(if\ A = 1, 1 + 0 = 1) \tag{1.4}$$

$$A + A = A \qquad (if\ A = 0, 0 + 0 = 0)(if\ A = 1, 1 + 1 = 1) \tag{1.5}$$

$$A + \overline{A} = 1 \quad (if\ A = 0, \overline{A} = 1, and\ 0 + 1 = 1)(if\ A = 1, \overline{A} = 0, and\ 1 + 0 = 1) \tag{1.6}$$

$$A * 0 = 0 \qquad (if\ A = 0, 0 * 0 = 0)(if\ A = 1, 1 * 0 = 0) \tag{1.7}$$

$$A * 1 = A \qquad (if\ A = 0, 0 * 1 = 0)(if\ A = 1, 1 * 1 = 1) \tag{1.8}$$

$$A * A = A \qquad (if\ A = 0, 0 * 0 = 0)(if\ A = 1, 1 * 1 = 1) \tag{1.9}$$

$$A * \overline{A} = 0 \quad (if\ A = 0, \overline{A} = 1, and\ 0 * 1 = 0)(if\ A = 1, \overline{A} = 0, and\ 1 * 0 = 0) \tag{1.10}$$

1.18 Duality

If we closely examine the rule that applies to the AND-ing and the rule applying to the OR-ing of terms, we see that a relationship exists. An AND operation requires that ALL of the arguments must equal 1 before the result is 1. The OR operation requires that only one of the arguments be true for the conclusion to be true.

$$0 * 0 = 0, \qquad 1 * 1 = 1, \qquad 0 * 1 = 0 \tag{1.11}$$

$$1 + 1 = 1, \qquad 0 + 0 = 0, \qquad 1 + 0 = 1 \tag{1.12}$$

Interchanging the operators, replacing ones by zeros, and zeros by ones does not effect the validity of the function. A **DUAL** exists for any Boolean statement. A dual function is, therefore, one in which the operators (AND and OR) have been interchanged and (if there are any in the function) zeros have been replaced by ones and ones have been replaced by zeros. Note the following expressions:

$$A + B,\ A(B+C),\ A(BC),$$

Their duals are, respectively:

$$AB,\quad A + (BC),\quad A + (B + C) = A + B + C,$$

Duality principle

`Every Boolean expression has a dual expression. If the dual of an expression is desired, simply interchange OR and AND operators, and replace ones by zeros and replace zeros by ones. If there are no zeros or ones we simply interchange the OR and AND operators to obtain the dual expression.`

1.19 Basic Theorems and Properties

Boolean distributive laws:
By factorization: $A(B + C) = AB + AC$
By duality, we obtain for the left side: $A(B + C) \Rightarrow A + (BC)$
and for the right side, by duality, we obtain: $AB + AC \Rightarrow (A + B)(A + C)$
Using the above identities, we obtain: $A + BC = (A + B)(A + C)$
This can be verified using absorption as follows:
$(A + B)(A + C) = AA + AC + AB + BC$
$= A + A(C + B) + BC$
$= A + BC$

Boolean absorption laws:

$$A + AB \ = \ A \tag{1.13}$$

Proof:
$A + AB = A(1) + (AB) = A(1 + B)$ (by identity 1 + B = 1)
$= A(1)$
$= A$
By duality: `A + AB => A(A+B)`
Let us check:

```
A (A+B) = AA + AB
        = A  + AB
        = A(1) + (AB)
        = A (1 + B) (by identity 1 + B = 1)
        = A(1)
        = A
```

Thus

$$A(A + B) \ = \ A \tag{1.14}$$

Boolean commutative laws:

$$AB \ = \ BA \tag{1.15}$$

By duality:

$$A + B \;=\; B + A \tag{1.16}$$

The first can be seen through perfect induction and the second one results from the duality.

Boolean associative laws:

$$A(BC) = (AB)C \tag{1.17}$$

By duality:

$$A + (B + C) = (A + B) + C \tag{1.18}$$

It is easy to see that the Boolean commutative and associative laws are the same as the commutative and associative laws in ordinary algebra.

The postulates and theorems of Boolean algebra thus far covered are summarized as in Table **??**.

Table 1.11: Summary of postulates and theorems of Boolean algebra

	Postulates and Theorems	Dual
	$X + 0 = X$	$(X)(1) = X$
	$X + \overline{X} = 1$	$(X)(\overline{X}) = 0$
	$X + X = X$	$(X)(X) = X$
	$X + 1 = 1$	$(X)(0) = 0$
Commutative:	$X + Y = Y + X$	$XY = YX$
Associative:	$X + (Y + Z) = (X + Y) + Z$	$X(YZ) = (XY)Z$
Distributive:	$X(Y + Z) = (XY) + (XZ)$	$X + (YZ) = (X + Y)(X + Z)$
Absorption:	$X + (XY) = X$	$X(X + Y) = X$
Invocation:	$\overline{\overline{X}} = X$	

The OR operation is often called **logical sum** or simply sum, join, or union and may be denoted by $(X + Y)$, or $(X \vee Y)$, or $(X \cup Y)$, or `(X OR Y)`. The AND operation is referred to as **logical multiplication** (product, intersection or meet) and may be denoted by (XY), or $(X \wedge Y)$, or $(X * Y)$, or $(X \cdot Y)$, or $(X \cap Y)$, or `(X AND Y)`.

In prefix notation: `(AND X Y)`.
In Polish notation (parentheses free notation), we will have `AND X Y`.
In Reverse Polish Notation (RPN): `X Y AND`.

The **prime** or **complement** $\overline{X}$ is often denoted by `X'`, $\neg X$, or $!X$. You have to be familiar with various notations that are used in various books and design automation tools that we will use.

Boolean properties that are very useful and worth memorizing are:

1. Commutative and associative properties identical to ordinary algebra.
2. Special distributive and absorption properties of Boolean algebra.
3. Duality
4. The identities:
5. $X + \overline{X} = 1 \quad (X)(X) = X$
6. $X + X = X \quad (X)(\overline{X}) = 0$
7. $(X)(1) = X$

As well as the De Morgan's laws that will follow.

1.20 De Morgan's Laws

De Morgan's laws, which we owe to the English mathematician **Augustus De Morgan** (1806 - 1871), founder with George Boole of symbolic logic, formally states that the complement of a product $(\overline{AB})$ is the sum of the complements $(\overline{A} + \overline{B})$, and that the complement of a sum $(\overline{A+B})$ is the product of the complements $(\overline{A}\ \overline{B})$. In other words, if we complement AB, we change the operator to (+) and we complement the literals. If we complement $(A + B)$, we change the operator (+) to (*) and we complement the literals, Figure **??**, Figure **??**.

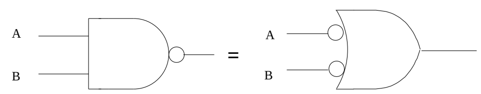


Figure 1.37: Complementing AB results in $\overline{A} + \overline{B}$

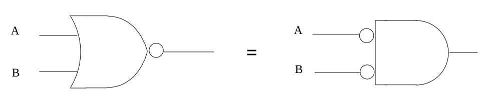


Figure 1.38: Complementing $A + B$ results in $\overline{A} \cdot \overline{B}$

Recall the design procedure example in which the double-throw light switch was designed. Referring once again to the truth table, we see that there are actually four solutions to this problem.

Table 1.12: Truth table for double-throw switch

A	B	$(\overline{A}B + A\overline{B})$	$(\overline{A} + B)(A + \overline{B})$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

As shown in the circuit designs below, the first circuit is synthesized with an OR gate at the output followed by two AND gates. The input A is inverted before one AND gate while the input B is inverted before the other. If we apply De Morgan's Laws to the first function, we obtain the second circuit, which has an AND gate at the output followed by two OR gates. While the first circuit is an EXOR operation, placing an inverter after its output changes it to an EXNOR operation equivalent to the second function, Figure **??**

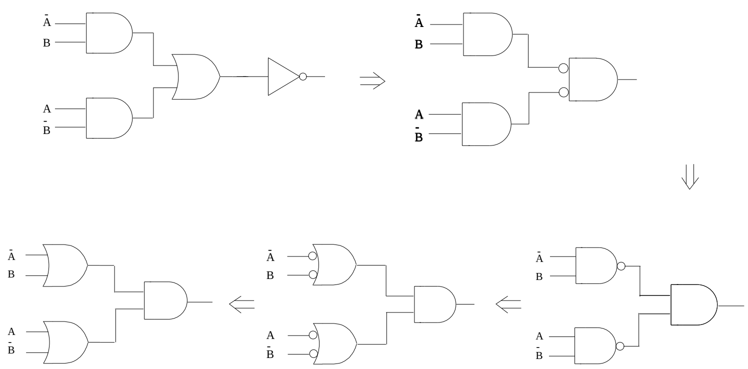


Figure 1.39: Illustration of De Morgan theorem

Illustration of DeMorgan theorem.
The same thing applies to the other two functions.

In the above example, however, the inverters are both before only one of the OR gates (in the first function) or before only one of the AND gates in the last function.

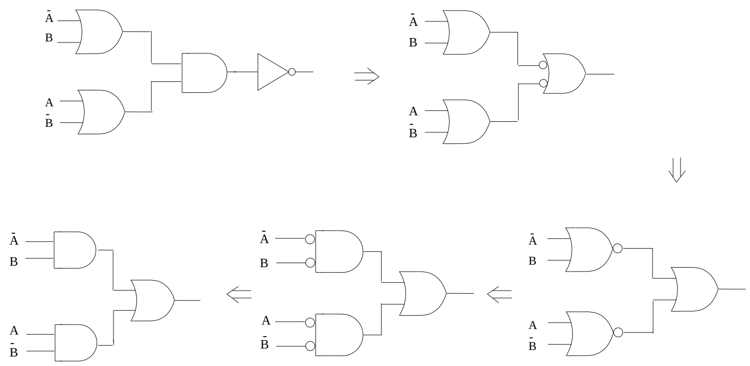


Figure 1.40: Illustration of De Morgan theorem

$(\overline{A}\ B)\ +\ (A\overline{B})$ is De Morgan's complement of $(\overline{A}\ +\ B)(A\ +\ \overline{B})$ and
$(AB)\ +\ (\overline{AB})$ is De Morgan's complement of $(A+B)(\overline{A}\ +\ \overline{B})$.
If we placed an inverter after the AND gate of the function $(A+B)(\overline{A}\ +\ \overline{B})$ its complement $(AB)+(\overline{A}\ \overline{B})$ would be the result.

This seems reasonable because $(A+B)(\overline{A}\ +\ \overline{B})$ is an EXOR operation. Placing an inverter after its output would make it an EXNOR operation, which $(AB)\ +\ (\overline{A}\ \overline{B})$ is.

Table 1.13: Truth table for double-throw switch

A	B	$(A+B)(\overline{A}\ +\ \overline{B})$	$(AB)\ +\ (\overline{A}\ \overline{B})$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

This demonstrates De Morgan's theorem which states that the complement of a function can be obtained by interchanging **AND** and **OR** operators and by complementing **each literal**.

Note that according to the principle of duality only the operators are interchanged (ones and zero are also assigned their opposite value if there are any). De Morgan's theorem calls for interchanging the operators as well as complementing all the literals. $\overline{A}$ becomes A and A becomes $\overline{A}$, etc.

De Morgan's Theorem and the principle of duality, however, have two distinct purposes. The principle of duality is implemented when we want to factor a Boolean function, manipulate it algebraically, and minimize it. In such a case, changing operators does not change the validity of the function. De Morgan's Theorem cannot be used in the same manner. Picture what would happen if we intentionally placed an inverter somewhere in a circuit. The function would be changed to the opposite of what it should be. If we, however, placed two inverters (one behind the other) in the circuit, the second one would cancel out the bad effect of the first one. We, therefore, want to use De Morgan's Laws to minimize the number of gates as well as the number of inputs in a circuit in order to save on hardware. At the end of this chapter, we will show more practical applications of De Morgan's Laws and discuss conversions to NAND and NOR logic. You will find them very useful in all your projects.

1.21 The Venn Diagram

The relationships among the variables of a Boolean expression can be illustrated using the "Venn diagram" shown in Figure ??. Each circle is labeled as variable A respectively variable B. The overlapping portion $= AB$, and both A and $B = 1$. Inside the circle A, $A = 1$, but $B = 0$, in the portion that does not overlap. Therefore, inside this area of circle A, we have $A\overline{B}$. However, inside circle B, the relationship in the pertinent area is $\overline{A}\ B$ since here $B = 1$ and $A = 0$. Outside both circles, the relationship is $\overline{A}\ \overline{B}$ since both A and $B = 0$.

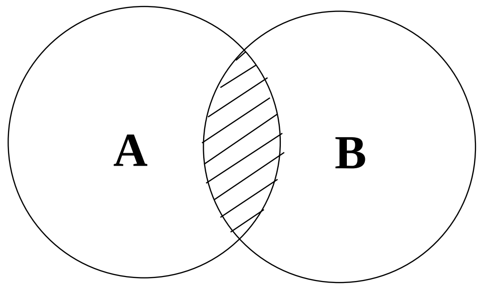


Figure 1.41: Venn Diagram for two variables

The Boolean absorption theorem is shown in the following Venn diagram, ??. The overlapping area of AB is actually included in the circle A and, therefore, $AB + A = A$.

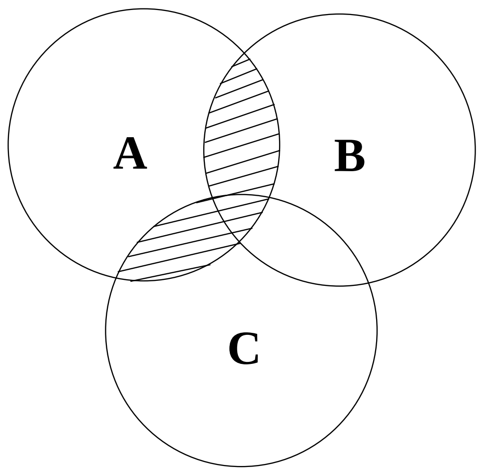


Figure 1.42: Venn Diagram for three variables

The distributive law for $AB + AC = A(B + C)$ is illustrated by the Venn diagram shown in Figure ??. The inner most area where the three circles overlap is shared equally by all three variables. Therefore, the area intersecting the circle A with the area enclosing B or C is the same area shared by AB or AC.

It is interesting that the concept of Venn Diagram has much to do with the "computer" of Lullus, the Jevon's machine, and next, the Svoboda's machine. It is a sense a predecessor of the CCM. We will discuss about this in much detail in the next chapters.

1.22 Canonical and Standard Forms

Minterms and Maxterms

The complementation and duality principles also allow us to introduce the terms **"sum of products" (SOP)** and **"product of sums" (POS)**. SOP is the logical sum of two or more logical products as in $(AC + BD)$. Its dual is the logical product of a logical sum expressed in POS terms: $(A + C)(B + D)$.

As we have demonstrated, a binary variable has a normal form (x) as well as a complemented form $(\overline{x})$. This means that for two variables such as a and b there are four different combination possibilities: ab, $\overline{a}b$, $a\overline{b}$, and $\overline{a}\,\overline{b}$. Each of these combination possibilities (AND terms) is a **standard product.**

Similarly, the normal form of the variable a as well as its complemented form $\overline{a}$ with the normal form of the variable b and its complemented form $\overline{b}$ expressed as $a+b$, $\overline{a}+b$, $a+\overline{b}$, and $\overline{a}+\overline{b}$ provide four different combination possibilities. Each of these OR terms is a **standard sum.**

In Boolean algebra, the OR-ing of terms refers to expressing a Boolean function as a **sum of minterms**. The expression is the standard sum of products (SOP) form and the terms are the **minterms**. Conversely, the AND-ing of terms means that the Boolean function is expressed as a **product of maxterms**. The expression is the standard product of sums (POS) and the POS terms are the **maxterms**.

The truth table Table **??** shows the maxterms and the minterms for three variables.

Table 1.14: Minterms and maxterms for three variables

a	b	c	Minterms	Maxterms	output
0	0	0	$\overline{a}\,\overline{b}\,\overline{c}$	$a+b+c$	0
0	0	1	$\overline{a}\,\overline{b}\,c$	$a+b+\overline{c}$	0
0	1	0	$\overline{a}\,b\,\overline{c}$	$a+\overline{b}+c$	0
0	1	1	$\overline{a}\,b\,c$	$a+\overline{b}+\overline{c}$	1
1	0	0	$a\,\overline{b}\,\overline{c}$	$\overline{a}+b+c$	0
1	0	1	$a\,\overline{b}\,c$	$\overline{a}+b+\overline{c}$	1
1	1	0	$a\,b\,\overline{c}$	$\overline{a}+\overline{b}+c$	1
1	1	1	$a\,b\,c$	$\overline{a}+\overline{b}+\overline{c}$	1

Minterms are formed by complementing the literal if its value is ”0”. Ones in the minterm are not complemented. By doing so, all of the terms are ”1”. Conversely, all of the maxterms are complemented so that they = ”0”. The ones in each maxterm are complemented so that they = ”0” but the zeros are not complemented.

The output shown in the table above is reflective of a function which we have arbitrarily chosen to illustrate the use of minterms and maxterms.

$$(\overline{a}\ b\ c)\ +\ (a\ \overline{b}\ c)\ +\ (a\ b\ \overline{c})\ +\ (a\ b\ c)$$

are the minterms that make up the SOP form of the function. For the function to be valid, all of the minterms that correspond to $f = 1$ must be written in a SOP form. Conversely, all of the maxterms that correspond to $f = 0$ are written in a POS form.

The corresponding POS expression is made up of maxterms,

$$(a+b+c)(a+b+\overline{c})(a+\overline{b}+c)(\overline{a}+b+c)$$

Note that, applying De Morgan's Laws, if this function is complemented, it yields:

$$\overline{f} = \overline{a}\ \overline{b}\ \overline{c} + \overline{a}\ \overline{b}\ c\ + \overline{a}\ b\overline{c} + a\ \overline{b}\ \overline{c}$$

which is the corresponding expression of the function in SOP form containing the minterms.

Both the SOP and the POS expression are equally valid solutions to the function.

1.23 Precedence of Operators

As in computer programming, the precedence of operators in digital logic is also:

1st parentheses
2nd NOT (complement)
3rd AND
4th OR

Correspondingly, for expression $\overline{(A+B)}$, the expression inside the parentheses is first evaluated, and then complemented. In expression $\overline{A}\ \overline{B}$, the complement of A and then the complement of B are evaluated, and then their result expressions are AND-ed.

1.24 Circuit Minimization by Algebraic Manipulation

As we mentioned earlier, it is of utmost importance to save on hardware cost and increase the speed of the circuit. Often, after having designed a circuit, it can be simplified and the number of gates and inputs can be reduced. One way of accomplishing this is to utilize the Boolean postulates and theorems elaborated on above.
The two functions shown below are duals of each other:

the functions $AB + A\overline{B}$ **and** $(A+B)(A+\overline{B})$**.**
They can both be simplified by factoring and using the identities.

Example 1.5

$AB + A\overline{B}$ becomes $A(B+\overline{B})$ after factoring out A.
This equals $A(1)$ since $X + \overline{X} = 1$
$(A+B)(A+\overline{B}) = A$ is equivalent after applying the principle of duality.
The solution to both functions is, therefore, simply A. These rules are very useful.

Minimize the following functions:
$ABC + \overline{A}B + AB\overline{C}$
The B is common in all of the terms, we, therefore, factor it out.
$= B(AC + \overline{A} + A\overline{C})$
Factor out A in the terms where it occurs.
$= B(A(C+\overline{C}) + \overline{A})$
Since $X + \overline{X} = 1$
$= B(A(1) + \overline{A})$
Since $X + \overline{X} = 1$
$= B(1)$
$= B$

Example 1.6

$AB\ +\ A\overline{B}C$
$= A(B + \overline{B}C)$, **factoring out** A.

$$= A(B+C) \tag{1.19}$$

The dual of this function is $(A+B)(A+\overline{B}+C)$
Let us multiply:
$= A + A(\overline{B}+C) + BA + B(\overline{B}+C) = A + BC.$
which is dual to the result of Formula **??**, as may be expected.

In the last example, terms can be added that do not affect the value of the function. They can be used, however, to factor out other terms and simplify the function.

Example 1.7

Minimize the function $AB\ +\ BC\ + C\overline{A}$.
$= AB + BC(1) + C\overline{A}$
$= AB + BC(A+\overline{A}) + C\overline{A}$ $\qquad (since\ A+\overline{A}=1)$
$= AB + ABC + \overline{A}BC + C\overline{A}$
$= AB + \overline{A}C$

1.25 NAND and NOR Implementation

Basic gate packages (ICs) supplied by manufacturers - mainly due to the savings involved - are usually made up of NAND or NOR gates, with the inverter feature already incorporated in the gates. It is, therefore, important to learn to convert the designed circuit to one using NAND and/or NOR gates.

For the conversion to NAND and NOR logic, we will use the invert-OR gate and the invert-AND gate. These are simply OR gates or AND gates with the inputs complemented. Recall that a NAND gate was an AND gate with its output complemented while a NOR gate was an OR gate with its output complemented.

By complementing the inputs of the OR gate, we obtain the equivalent, by De Morgan's Laws, of the NAND gate. Conversely, by complementing the inputs of the AND gate, it can be used in place of a NOR gate (Figure **??**).

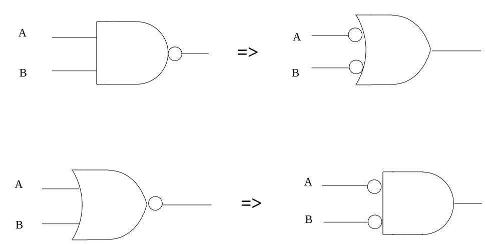


Figure 1.43: De Morgan Theorem for gates: NAND and Invert-OR gate, NOR and Invert-AND gate

De Morgan Theorem for NAND and NOR gates

When we use the small circle at the input terminals of the AND-invert or OR-invert gates, this graphic symbol should be understood to denote a "negative-logic" polarity for the inputs, but the output of the gates would have a "positive-logic" assignment. We have placed the negative and positive logic polarity in the truth table below to show the relationships between the AND, NAND, OR-invert, OR, NOR, and AND-invert gates.

A	B	AND	NAND	OR-invert	OR	NOR	AND-invert
0	0	0	1	1	0	1	1
0	1	0	1	1	1	0	0
1	0	0	1	1	1	0	0
1	1	1	0	0	1	0	0

Throughout this text, positive-logic is used exclusively in the examples.

The circuit below, with the four inputs a, b, c, d, shows the function $f = c + (\overline{a}\,\overline{b}) + (a\,\overline{d})$. Figure **??**.

Figure 1.44: Conversion to NAND gates

Conversion to NAND gates.

To change this circuit to a circuit of only NAND gates, we proceed as follows:

1. Starting at the output, complement the inputs of the OR gate (small circles).

2. Complement all of the gates (or literals) behind the output OR gate in order to correct the change that occurred by implementing #1 above (a double negative = affirmative). Otherwise, the validity of the function would have been changed.

   In case of partial conversion do:

   - replace the output gate by a NAND gate since it is equivalent to the invert OR gate.
   - both NAND gates should be replaced by invert OR gates.
   - cancel the inverters as well as the complements on the inputs of the invert OR gates. An inverter, however, must be added before the input of the lower OR gate. The complement + complement cancelled each other out. See Figure **??**.

Figure 1.45: Partial Conversion to NAND gates

Partial conversion to NAND gates.

At the output of the circuit, the function emerges as $\overline{\overline{c}(a+b)(\overline{a}+d)}$. Applying De Morgan's Laws, however, we can change the operators and complement the literals to obtain

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

which is the function that we began with. The final circuit is, therefore, equivalent to the original circuit. In the original circuit, however, there were six gates. In the final circuit, there are only five gates. By using NAND gates, we saved one gate. This procedure is recommended because often the savings can be even greater.

Similarly conversions and partial conversions to NOR gates can be done. Further, one can use extension of this method to create circuits from a mixture of AND, OR, NAND, and NOR gates.

1.26 Minimization of Two-Level Boolean Functions

1.26.1 Basic Notions

Boolean Hypercubes

Before synthesizing a circuit design, the Boolean function f should always be simplified as concisely as possible. To this end, the truth table representation of a function is unique but often the function is too complex for this method to be the most efficient approach. In previous sections we discussed the algebraic manipulation possibilities of minimizing a Boolean function. These methods, however, lack specific rules that would direct us to a successful solution using a clear sequence of steps.

As a function, however, the Boolean algebraic expression can also be plotted in a Cartesian coordinate system, Figure **??**. We can view the x axis as being representative of the Boolean variable x and let the y axis denote the variable y of a function. Each have a most significant binary value of 1 and a least significant value of 0. Each line, x and y, stands for one variable but, of course, a one variable function would not need a logic gate to implement it.

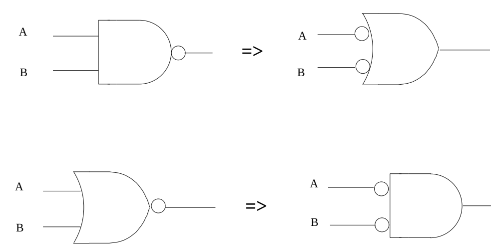


Figure 1.46: Cartesian Coordinate System for Boolean Functions

The input signals x and y in a two-input gate, however, can exist in one of four (2 squared = 4) possible states as the truth table for x and y shows, Table **??**.

X	Y	States possible
0	0	00
0	1	01
1	0	10
1	1	11

Actually Boolean algebra is simply a lattice with a few specific properties. Observe the Cartesian coordinate system shown in Figure **??**. Three mutually perpendicular lines (x axis, y axis, z axis) pass through a point 0, called the **origin**. Only the positive portions of the axes are shown. The positive y axis is pointing away from the viewer, the positive z axis is pointing to the right, the positive x axis is pointing upward. P = any point in space and the three planes through P perpendicular to the three axes intersect the x axis, the y axis, and the z axis at the points corresponding to the numbers x, y, and z, respectively. Therefore, P is associated with the **"ordered triple"** of numbers (x, y, z) which are called the **rectangular** or **Cartesian coordinates** of P.

5in 5in

Figure 1.47: Functions of three variables represented in a 3-Dimensional Cartesian System

Using only the yz plane, the points $P(0,0)$, $P(0,1)$, $P(1,0)$, and $P(1,1)$ emerge, which are the four possible states for two variables in Boolean two-state algebra. Connecting the four points at which the sides of the angle formed intersect (the vertices), a rectangle is constructed. This represents the basis on which we will build. The lattice plays an important role in characterizing various computation models and it is highly recommended that the serious student of digital logic become familiar with it.

To extend the above example to three variables, we utilize all the numbers (x, y, z) in the ordered triple forming two rectangles as shown in Figure **??**.

5in 5in

Figure 1.48: Kmap 3

Thus by connecting each vertex to its adjacent vertex, a cube emerges, which represents the eight possible states (2 raised to the 3rd power) for 3 variables in Boolean two-state algebra seen in the truth table Table **??**.

X	Y	Z	Possible states
0	0	0	000
0	0	1	001
0	1	0	010
0	1	1	011
1	0	0	100
1	0	1	101
1	1	0	110
1	1	1	111

Note that two vertices are called "adjacent" when they differ only in one coordinate value: For example:

000, 001, 010, and 100
001, 011, and 101
011, 010, and 111
010, 011, 000, and 110, etc.

For brevity, vertices will also be referred to as 0-cubes.
The two variable (y, z) example produces ordered pairs which represents "two-dimensional space" since each point in space has two coordinates. The three variable (x, y, z) example, however, yields ordered triples and therefore represents "three-dimensional space" since each point in space has three coordinates. These points can also be represented in a map coordinate system called the **Karnaugh map** which is made up of rectangles that stand for one minterm or for one maxterm each.

1.27 Karnaugh Maps

The Karnaugh map procedure is a very powerful tool to find the simplest, minimal expression of the function in the most direct manner. From it, we can obtain all possible ways that a function can be expressed in a standard form, enabling us to select the best solution: any one in a sum of products (SOP) or product of sums (POS) with a minimum number of variables. The above two-dimensional space for two variables is transferred to the Karnaugh map as shown in Table **??**. As shown by arrows, rows correspond to variable x, and columns correspond to variable y.

$\downarrow x, \rightarrow y$	0	1
0	$\overline{x}\,\overline{y}$	$\overline{x}\,y$
1	$x\,\overline{y}$	$x\,y$

=

$\downarrow x, \rightarrow y$	0	1
0	00	01
1	10	11

Since the two variables (x, y) have two values (0 and 1) each, x, y, $\overline{x}$, and $\overline{y}$ are all minterms. There are four rectangles, one for each minterm. The 0s and 1s, marked for each row and for each column, designate the values of x and y, respectively.

The map for the three variables x, y, and z would appear as shown in Table **??**. Rows correspond to combinations of variables x and y, and columns correspond to variable z.

$\downarrow xy, \rightarrow z$	0	1
00	$\overline{x}\,\overline{y}\,\overline{z}$	$\overline{x}\,\overline{y}\,z$
01	$\overline{x}\,y\,\overline{z}$	$\overline{x}\,y\,z$
11	$x\,y\,\overline{z}$	$x\,y\,z$
10	$x\,\overline{y}\,\overline{z}$	$x\,\overline{y}\,z$

=

$\downarrow xy, \rightarrow z$	0	1
00	000	001
01	010	011
11	110	111
10	100	101

Notice that the map for three variables is twice the size of the map for two variables. The two variable map had four (2 squared= 4) rectangles while the map for three variables has eight (2 raised to the 3rd power = 8) rectangles. Whenever a variable is added, the size doubles. A map for four variables would have (2 raised to the 4th power= 16) sixteen rectangles, etc. x and y are designations of the rows 00, 01, 11, 10 where $x = 0, 0, 1, 1$, and $y = 0, 1, 1, 0$ as in the truth table. z denotes the columns for $z = 0, 1$.

In the above (generalized) Cartesian coordinate system, two variables formed a rectangle. When three variables were used, a rectangle was added (one top and one bottom rectangle) so that by connecting the adjacent vertices a cube was formed. The consequence is that four variables would be represented by two cubes as shown below and which is referred to as a four-dimensional hypercube. Figure **??**.

5in 5in

Figure 1.49: Four-dimensional hypercube, four-variable Karnaugh map

Note that when going from a two variable map to a three variable map, the map was extended in the length. When the next variable was added for a four variable map, it was doubled in the width.

Five variables would be represented by four cubes or (2 raised to the 5th power) 32 rectangles in the map. Six variables are represented by eight cubes or 64 rectangles in the map, etc. Figure **??**.

5in 5in

Figure 1.50: Five variable Karnaugh map

The five variable map is, therefore, double in size to the right, which is equivalent to having two cubes placed side by side. Figure **??**.

5in 5in

Figure 1.51: Six variable Karnaugh map

The six variable map is doubled in length.

This procedure can then be extended for additional variables.

Note that there are various ways in which a function can be placed into and read from a map. The student that pursues studies in digital logic will encounter them all at one time or the other. These various methods will not be shown here, however, so that confusion can be kept to a minimum and the learning of good design techniques is facilitated. The coordinate system used in the map in this text is in the sequence 0000, 0001, 0011, 0010, etc. which equals the decimal numbers 0, 1, 3, 2, etc. This is the so called "Gray" or reflected code. In this sequence, one number differs by no more than one digit to the next number. There are, however, also other coordinate designations that can be used.

1.28 Realizing a Function in a Map

SOP Form

To illustrate the procedure of realizing a function by the map method, recall the two forms of the function used to introduce the canonical and standard forms in the preceding chapter. The SOP form satisfying the requirements contained the **"minterms"**

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

Conversely, the POS form contained the **"maxterms"**

$$(a+b+c)(a+b+\overline{c})(a+\overline{b}+c)(\overline{a}+b+c)$$

The literals a, b, or c have the value of 1 in their normal form but have the value 0 in their complemented form. The rectangle coordinates for the SOP form

$$(\overline{a}bc)+(a\overline{b}c)+(ab\overline{c})+(abc)$$

are, therefore,

$$(011) + (101) + (110) + (111)$$

(1st) Place a 1 in each of these rectangles in the map as shown below.
(2nd) The adjacent terms (terms that differ no more than by one coordinate value) need to be grouped. The number of minterms grouped into a "cell", as it is called, must be a number raised to the power of two. This means that the "cell" can be composed of either only one retangle or it can be made up of 2, 4, 8, 16, etc. minterms. See Figure **??**.

5in 5in

Figure 1.52: map

Note that the rectangle coordinate (111), in this case, is shared three times. This is done in order to minimize the number of variables and to insure maximum coverage.
(3rd) The original four minterms have now been combined into three cells. Each cell consists of two coordinates each. Both of these coordinates need to be combined to yield only one for the cell. This is done as follows:

```
011   110   101
111   111   111
---   ---   ---
-11   11-   1-1
```

yields $bc + ab + ac$.

Note that the coordinates are not added together but only compared. A 1 with a 1 yields a 1. Also, a 0 with a 0 yields a 0. A 1 with a 0 or a 0 with a 1, however, produces a redundant literal which is equal to neither one nor zero. The redundant literal is denoted by a dash (-) above to show that it is left out of the term when the cell coordinate is written. Then the coordinate (-11), for example is simply bc, 11- equals ab, and 1-1 is ac. In the first cell above, the a is the redundant literal. Even when multiple minterms are compared, if one literal differs from the others, a redundant literal is the result.

When converting the "cell coordinate" into an algebraic expression the variable a is to the far left, b is in the middle, and c is to the far right. If the digit in the corresponding position is a 1, the variable remains in its normal form. If the digit is a 0, the variable is in its complemented form. If a redundant literal is in that position, the variable is not listed. Since there are ones in these "cells", the expression is written in its SOP form.

The minimized expression for

$$(1-1)(11-)(-11)$$

is therefore,

$$(ac) + (ab) + (bc)$$

Note, however, that there is an a that can be factored out of the expression that will simplify it even further to

$$a(b + c) + (bc)$$

These three circuits (the original, the minimized, and the factored function) are synthesized and shown in Figure **??**.

5in 5in

Figure 1.53: Three circuits

Comparing these three possibilities, demonstrates the need for minimization. The number of gates and inputs compares as in Table **??**.

	gates	inputs
original	8	16
minimized	4	9
factored	4	8

Converting to NAND logic

The savings of one input in the above example, however, does not justify synthesizing the circuit with the factored version of the function. Since NAND gates will most likely be more available, it would be wiser to convert the minimized version to the NAND version of the circuit as shown below.

As outlined at the end of the previous chapter, the NAND version is implemented by first changing the output OR gate of the minimized circuit to an "invert-OR gate" and exchanging all of the following AND gates for NAND gates. Then the output invert-OR gate can be exchanged for its equivalent: a NAND gate. The output produced is

$$\overline{\overline{a\ b}\ \ \overline{a\ c}\ \ \overline{b\ c}}$$

which by De Morgan's Laws is equivalent to

$$(ab) + (ac) + (bc)$$

POS form

Direct your attention now to the coordinates that contain zeros in the above Karnaugh map. These are

$$(000)(001)(010)(100)$$

equivalent to

$$(\overline{a}\ \overline{b}\ \overline{c})(\overline{a}\ \overline{b}\ c)\ (\overline{a}\ b\ \overline{c})(a\ \overline{b}\ c)$$

Recall that in our introduction of the canonical forms these were the terms of the function that equaled zero.

When De Morgan's Laws are applied, we see that they are equal to

$$(a + b + c)(a + b + \overline{c})(a + \overline{b} + c)(\overline{a} + b + c)$$

which are the maxterms in the POS form.

Placing a POS form of a function in a map

The POS form of the function is placed in the map in a similar manner as when the minterms were placed in the map.

The coordinates for

$$(a + b + c)(a + b + \overline{c})(a + \overline{b} + c)(\overline{a} + b + c)$$

are, however,

$$(000)(001)(010)(100)$$

Everything is opposite from the SOP form: zeros are ones and ones are zeros in the map. It is best to first take the complement of the function in POS form to obtain

$$(\overline{a}\ \overline{b}\ \overline{c})\ + (\overline{a}\ \overline{b}\ c) + (\overline{a}\ b\ \overline{c}) + (a\ \overline{b}\ \overline{c})$$

and the coordinates can be read as listed above. Otherwise, think of the variable in its normal form as a 0 and in its complemented form as a 1.

- In the case of the POS form, however, a 0 is placed in each of the rectangles that these coordinates designate. The cells are formed in the same manner as before.
- However, in the POS form, a 1 stands for the complemented form and the maxterms are OR-ed.

Therefore, the "cell" coordinates are

```
000  000  000
100  001  010
---  ---  ---
-00  00-  0-0
```

which equals the complement of the function.

$$\overline{f} \;=\; (\overline{b}\,\overline{c}) + (\overline{a}\,\overline{b}) + (\overline{a}\,\overline{c})$$

However, applying De Morgan's Laws

$$f \;=\; (b+c)(a+b)(a+c)$$

is the POS form. Its circuit is shown in Figure **??**.

Figure 1.54: The circuit in POS form

Converting to NOR logic

As in the SOP form example, this circuit can also be implemented using NOR logic. By changing the output AND gate to an `"invert-AND gate"` and changing the following OR gates to NOR gates, the validity of the function is not effected. The output invert-AND gate, however, can now be exchanged for a NOR gate, which leaves us with only NOR gates in this POS function. The output of the circuit is now

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

which according to De Morgan's Laws is equivalent to

$$(a+b)(b+c)(a+c)$$

5in 5in

Figure 1.55: NOR circuit

Note that the SOP form was the NAND circuit while the POS form was the NOR implementation.

1.29 Proof of Boolean Properties by the Map Method

In section ??, we showed how a Boolean function could be minimized by algebraic manipulation. Now that we have shown how a literal becomes redundant by map minimization, we can relate this to the Boolean properties.

$$A(A+B) \text{ by duality } = A + AB \text{ which is equal to } A.$$

Placing $A + AB$ in the map below shows that when the minterms are grouped into a cell, (1-) emerges which equals A since B became redundant.

Given a function such as $ABC + A$, for example, we can see that this calls for a three variable map (ABC). In the second term of the function, however, only A appears. It is understood, in this case, that B and C are redundant and the term is equivalent to $(A - -)$. It, therefore, actually represents the minterms $AB\overline{C}$ and $A\overline{B}C$ but since 110 and 101 have made B and C redundant, the term is given only as A. When a literal is missing in a given function, you need to look for the minterms in the map that made them redundant.

$$A(\overline{A}+B) \text{ by duality } = A + \overline{A}B \text{ which equals } AB.$$

By the map, 10 and 11 are grouped to yield 1- making B redundant and 01 is grouped with 11 for -1 in which A is redundant. Therefore, $A + \overline{A}B = AB$.

$$(A+B)(A+\overline{B}) \text{ by duality } = AB + A\overline{B} \text{ which equals } A.$$

The map shows the grouping of 10 with 11 to yield 1- which = A.

When a function is common in all terms, it will cover all of them such as in

$$A\overline{B} + A\overline{B}C + A\overline{B}\,\overline{C} + A\overline{B}D \;=\; A\overline{B}$$

As we see from the map below, the minterms 1000, 1001, 1011, and 1010 yield 10–, which is $A\overline{B}$ since both C and D became redundant.

Similarly, the function

$$ABC + A\overline{B}C + AB\overline{C} + A\overline{B}\,\overline{C} \;=\; A$$

in the map from Figure ?? because the group of minterms 110, 111, 100, 101 eliminated B and C leaving only A.

5in 5in

Figure 1.56: (maps)

1.30 Grouping Adjacent Terms

Note that when we say to group adjacent terms, we do not mean only terms that are "side by side" in the map. Vertices or 0-cubes that differ only in one coordinate value are meant. As in the example of

$$f = (\overline{a}\ \overline{b}\ \overline{c}) + (\overline{a}\ \overline{b}\ c) + (\overline{a}\ b\ \overline{c}) + (a\ \overline{b}\ \overline{c})$$

above, the coordinate 000 at the top and the coordinate 100 at the bottom of the map were grouped together into one cell. Corner, top, bottom and side retangles are grouped together. Also recall that the points P in the n-dimensional space of the Cartesian coordinate system are represented by vertices or 0-cubes which are identical with the "minterms" or "maxterms" placed in the rectangles of the map, and are, therefore, the map coordinates. Refer to the cube and you will see that the vertices 000 and 100 are "side by side" (in the cube) even though these coordinates are found at the top and the bottom of the map.

A cell, however, is a group of minterms or maxterms and constitutes what we refer to as a **"prime implicant"** in the SOP form, or as a **"prime implicate"** in the POS form.

1.31 Minimization by the Map Method as Opposed to Algebraic Minimization

The above grouping procedure as well as the ease of minimization using the map method, is best demonstrated when we insert the function

$$f = (a\ b\ c) + (b\ c\ d) + (a\overline{b}\ \overline{d}) + (\overline{a}\ b\ c\overline{d}) + (\overline{a}\ \overline{b}\ \overline{d})$$

into the map. As shown in Figure **??**, the variable A is "absorded" completely. By forming one prime implicant from the corners and grouping the remaining four minterms into a prime implicant, the function is simplified to $f = bc + \overline{bd}$. Note for later reference when we introduce "hazards" that we might want to add the prime implicant $(c\overline{d})$ to the realization of this function to avoid a possible hazard.

5in 5in

Figure 1.57: BR-E-map

The circuit for $f = bc + \overline{b}\ \overline{d}$ is shown in Figure **??**. The second diagram shows the same circuit in its NAND gate implementation.

5in 5in

Figure 1.58: The NAND circuit without hazard

To further demonstrate the usefulness of the map method, let us first minimize the function

$$f \ = \ (\overline{a}+\overline{b}+\overline{c})(a+b+c)\ (\overline{a}+\overline{b}+c)(a+b+\overline{c})$$

by the algebraic method and then place it in a Karnaugh map.

First, we convert the function to its dual so that it can be factored.

$$(\overline{a}\ \overline{b}\ \overline{c}) \ + \ (a\ b\ c)+(\overline{a}\ \overline{b}\ c) \ + \ (a\ b\ \overline{c})$$

We split the function into two halfs and factor $\overline{a}$ out of the first half and factor a out of the second half.

$$\overline{a}[(\overline{b}\ \overline{c}) \ + \ \overline{b}\ c] \ + \ a(b\ c \ + \ b\ \overline{c})$$

Factor out $\overline{b}$ and b.

$$\overline{a}[\overline{b}\ (\overline{c} \ + \ c)] \ + \ a[b(c \ + \ \overline{c})]$$

since $(c \ + \ \overline{c}) = 1$

$$\overline{a}[\overline{b}(1)]+a[b(1)] = (\overline{a}\ \overline{b}) \ + \ (a\ b)$$

As you will recall, this is the EXNOR function (a negation of EXOR).

The SOP form and the POS form appear in the maps from Figure **??**.

5in 5in

Figure 1.59: SOP and POS maps

For practise, sketch the original circuit. You will see that there is a considerable savings when, in the end, only one EXOR is needed to implement the circuit. Consider how you could have put this function in a truth table to obtain a solution.

1.32 Summary on Functions

A number of functions have already been introduced which could be recognized from their patterns in the truth table. We will once again list these functions for two variables below in the truth table, Table **??** with the appropriate name of the operation. Subsequently, we show the corresponding map for the function for your information.

1	2	3	4	5	6	7	8	9	10
A	B	AB	$\overline{(AB)}$	A+B	$\overline{(A+B)}$	$A\overline{B}$	$\overline{A}\ B$	$A\ \overline{B}\ +\ \overline{A}B$	$\overline{A}\ \overline{B}\ +\ AB$
0	0	0	1	0	1	0	0	0	1
0	1	0	1	1	0	0	1	1	0
1	0	0	1	1	0	1	0	1	0
1	1	1	0	1	0	0	0	0	1

1	2	11	12	13	14	15			
A	B	AB	$\overline{(AB)}$	A+B	$\overline{(A+B)}$	$A\overline{B}$	$\overline{A}\ B$	$A\ \overline{B}\ +\ \overline{A}B$	$\overline{A}\ \overline{B}\ +\ AB$
0	0	0	1	0	1	0	0	0	1
0	1	0	1	1	0	0	1	1	0
1	0	0	1	1	0	1	0	1	0
1	1	1	0	1	0	0	0	0	1

BLAD POWYZEJ

1.33 Binary Specification of a Function

A Boolean function can also be specified in its SOP or POS form by its "minterm" or "maxterm" binary equivalent as shown below. In the SOP form, the function
$(\overline{a}\,\overline{b}\,\overline{c}\,\overline{d}) + (\overline{a}\,\overline{b}\,\overline{c}\,d) + (\overline{a}\,b\overline{c}\,d) + (\overline{a}\,b\,c\overline{d}) + (a\,b\overline{c}\,\overline{d}) + (a\,b\overline{c}\,d) + (a\,b\,c\overline{d}) + (a\,\overline{b}\,\overline{c}\,\overline{d}) + (a\,\overline{b}\,c\,\overline{d})$
can be expressed by listing the map coordinates of the minterms of the function as follows

$$F^1 = \begin{bmatrix} 0000 \\ 0001 \\ 0101 \\ 0110 \\ 1100 \\ 1101 \\ 1110 \\ 1000 \\ 1010 \end{bmatrix}$$

Note the 1 that is placed at the top of F to indicate that ones should be placed in these coordinates.

Since there are four digits in each term, a four variable map is needed to minimize the function. Therefore, a map with 2 raised to the power of four = 16 rectangles is constructed and ones are entered at the above given coordinates. Figure **??**.

5in 5in

Figure 1.60: Map for SOP function

The "prime implicants" are created by circling and grouping the minterms into cells of 1, 2, 4, etc. minterms each. Therefore, the following minterms that differ no more than by one coordinate can be grouped together.

```
0000  0000  0001  0101  1100  1100  0110  1110  1000
1000  0001  0101  1101  1101  1000  1110  1010  1010
----  ----  ----  ----  ----  ----  ----  ----  ----
-000  000-  0-01  -101  110-  1-00  -110  1-10  10-0
```

It soon becomes apparent that the SOP approach is not a very minimized solution. Therefore, in the same manner, the POS form is as follows.

$$F^0 = \begin{bmatrix} 0011 \\ 0010 \\ 0100 \\ 0111 \\ 1111 \\ 1001 \\ 1011 \end{bmatrix}$$

Subscript 0 of F^0 indicates that zeros are to be placed at these coordinates since it is the POS form.

The zeros in the above map are, therefore, circled to form the "prime implicates" and the groups can be listed as shown below.

```
0100  1001  1011  0011  0011
      1011  0011  0010  0111
                        1111
                        1011
----  ----  ----  ----  ----
0100  10-1  -011  001-  --11
```

Note that 0100 is a lone coordinate but the top coordinate 0011 and the bottom coordinate 1011 have been grouped twice.

The resulting POS form is, therefore,

$$\overline{f} = (\overline{a}\, b\, \overline{c}\, \overline{d}) + (a\, \overline{b}\, d) + (\overline{b}\, c\, d) + (\overline{a}\, \overline{b}\, c) + (c\, d)$$

which equals

$$f = (a + \overline{b} + c + d)(\overline{a} + b + \overline{d})(b + \overline{c} + \overline{d})(a + b + \overline{c})(\overline{c} + \overline{d})$$

when De Morgan's Laws are applied.

Both of these circuits are shown below with NAND and NOR logic equivalent circuit, Figure **??**.

5in 5in

Figure 1.61: Equivalent circuits

1.34 Decimal Specification of a Function

A function can also be expressed in such a way that the coordinates in the map are related to the decimal equivalent. For example

$$f(a,b,c,d) \;=\; \sum(2,3,4,7)$$

$\sum$ stands for "the sum of" and means that these are minterms in the SOP form.

When the function is expressed as

$$f(a,b,c,d) \;=\; \prod\;(2,3,4,7),$$

$\prod$ stands for the POS form and means that these are maxterms.

The decimal numbers refer to the rectangle in the map that corresponds to the binary equivalent.

In the following example, the row marked binary (01) and the column marked binary (1) form the rectangle binary (011) or ($\overline{a}\ b\ c$), which is equivalent to the decimal number 3. Figure **??**.

5in 5in

Figure 1.62: Three variable Karnaugh map

The function

$$f(a,b,c,d) \;=\; \sum\;(0,1,5,6,8,10,12,13,14)$$

indicates that there are four variables and therefore a map with 16 rectangles is needed.

Given a SOP function such as

$$f(a,b,c,d) \;=\; \sum\;(0,1,5,6,8,10,12,13,14)$$

ones would be placed in the rectangles bearing these numbers.

Given a POS function such as

$$f(a,b,c,d) \;=\; \prod\;(2,3,4,7,9,11,15)$$

zeros would be entered in these rectangles.

These are the same coordinates as in the previous example except that they are the decimal equivalent of the coordinate for the rectangle.

1.35 The Don't Cares

If the value of a mintern is unknown or if its value is of no significance, we denote it by X and call it a "don't care". When "don't-cares" (unspecified values) occur in a function, they can be grouped with the ones in the SOP form or with the zeros in the POS form to simplify the function. If they are not adjacent and not needed, they can, however, be ignored. The coordinates of the unspecified values can be given in binary form or they can be specified by their literals ($a\;\overline{b}\;c\;\overline{d}\;e$). In the function given below, the coordinates of the terms inside the "inside" parenthesis contain "don't-cares".

$$f(a,b,c,d,e) \;=\; \sum[0,10,11,12,13,18,19,22,25,26,(2,4,6,9,23,29)]$$

This is a five variable function and therefore a map with 4 rows and 8 columns (32 rectangles) is needed. After inserting the ones and the "don't-cares" at the indicated coordinates, the minterms can be grouped to ascertain the prime implicants. These are as follows, Figure **??**.

5in 5in

Figure 1.63: 5 variable map

```
00100  01101  01001  00110  10011  00010  01101
01100  01100  01011  00010  10010  01010  11101
                     00000  10110  11010  01001
                     00100  10111  10010  11001
-----  -----  -----  -----  -----  -----  -----
0-100  0110-  010-1  00--0  10-1-  --010  -1-01
```

$$f \;=\; (\overline{a}\;c\overline{d}\;\overline{e}) + (\overline{a}\;b\;c\overline{d}) + (\overline{a}\;b\;\overline{c}\;e) + (\overline{a}\;\overline{b}\;\overline{e}) \;+\; (a\;\overline{b}\;c) \;+\; (\overline{c}\;d\;\overline{e}) \;+\; (b\;\overline{d}\;e)$$

Implement this function using NAND logic.

1.36 The Tabulation Method

1.36.1 Quine-McCluskey Method of Minimization

This method of tabulation can be used to simplify a function without factoring or mapping it. Recall that $AA = A$ and that $A + A = A$. Therefore, one A can be eliminated and minimum cover is still insured. There is no need to have two functions that accomplish the same goal. The Quine-McCluskey method is a way to cross-check and eliminate unnecessary double-cover.

As we have seen, a function can be given in one of the following ways.

$$f = (\overline{a}\, b\, \overline{c}\, \overline{d}) + (\overline{a}\, b\, c\, d) + (a\, \overline{b}\, \overline{c}\, d) + (a\, \overline{b}\, c\, \overline{d}) + (a\, b\, \overline{c}\, \overline{d}) + (a\, b\, \overline{c}\, d) + (a\, b\, c\, \overline{d}) + (a\, b\, c\, d)$$

$$F^1 = \begin{bmatrix} 0100 \\ 0111 \\ 1001 \\ 1010 \\ 1100 \\ 1101 \\ 1110 \\ 1111 \end{bmatrix}$$

$$f(a, b, c, d) = \sum (4, 7, 9, 10, 12, 13, 14, 15)$$

First set up a table to determine the prime implicants as shown in Table **??**. Enter the minterms as columns across the top of the table. Leave the left column empty so that the prime implicants can later be entered in it as the rows. Enter the minterms in their binary form but place the decimal equivalents in parenthesis under them to facilitate keeping track of where the prime implicants were derived from.

Table 1.15: Prime Implicant Table

	minterms							
Prime Implicants	0100 (4)	1001 (9)	1010 (10)	1100 (12)	0111 (7)	1101 (13)	1110 (14)	1111 (15)

Next, we need to compare all of the minterms to determine which of them differ by only one coordinate. First, we create a table and list all of the minterms in a column and sort them according to the number of ones that they contain. Since the function in question is made up of four variables, it is possible for a minterm to contain 0 ones, 1 one, 2 ones, 3 ones, or 4 ones. Five groups would then be needed. In this case, however, none of the minterms contain 0 ones. Therefore, a 0-level is not needed and we can begin with level #1 and have 4 levels as shown in Table ??. Recall that minterms are the vertices, or 0-cubes as they have been labeled. Level #1 is made up of the 0-cubes that contain only 1 one. Level #2 is made up of those that contain 2 ones, etc.

The 0-cubes from the higher level are then compared with those in the level below it. If by comparing two 0-cubes, they differ by only one coordinate the resulting prime implicant is listed in a second column called "1-cube" as shown in Table ??. When a 0-cube has been used for comparison, mark it in some way (we have used (*) to do so) this will serve as a future reminder when it is compared again. For example, note the first row of the 1-cube column in the Table ??. The term (-100) is listed with (4,12) beside it because it was derived from these terms. In the 0-cube column, the 4 and 12 are marked by (*) as a reminder that they have already been covered by a prime implicant. On the other hand, the term (110-) in the 1-cube column is also marked (*) because when it was formed from 12 and 13 they had been used already to form the prime implicants (-100) and (1-01). The same thing applies for all the other terms in this column marked (*).

	0-cubes	1-cubes	2-cubes
level #1	0100 (4)*	-100 (4,12)	
level #2	1001 (9)*	1-01 (9,13)	
	1010 (10)*	1-10 (10,14)	
	1100 (12)*	110- (12,13)*	
		11-0 (12,14)*	11–
level #3	0111 (7)*	-111 (7,15)	(12,13,14,15)
	1101 (13)*	11-1 (13,15)*	
	1110 (14)*	111- (14,15)*	
level #4	1111 (15)*		

Table 1.16: Quine Table

The 1-cubes from level one are then compared with those on level two and the 1-cubes on level two are compared with those one level three, etc. If during the comparison, the two 1-cubes differ by only one coordinate they are marked by (*) as above and the resulting prime implicant is listed in a third column called "2-cube". This procedure is continued for n-columns until none of the terms differ.

The "end products" are those terms that were not used to make up a new term. This means that the prime implicants are those terms that have not been marked (*). In the above example these are (-100), (1-01), (1-10), (-111), and (11–), which we enter in the prime implicant table as rows. As you enter these prime implicants place a mark (x) in the column of the minterms from which this prime implicant was derived.

Table ??

It is not necessary to list (110-), (11-0), (11-1), and (111-) because these are marked to show that they are covered in the above prime implicants.

Table 1.17: Prime Implicant Table

	Minterms							
Prime Implicants	0100 (4)	1001 (9)	1010 (10)	1100 (12)	0111 (7)	1101 (13)	1110 (14)	1111 (15)
-100 (4,12)	(x)			(x)				
1-01 (9,13)		(x)			(x)			
1-10 (10,14)			(x)			(x)		
-111 (7,15)				(x)				(x)
1 11– (12,13,14,15)				x		x	x	x

Finally, locate the minterm columns that contain only one (x). Circle the (x) in that column because if this minterm is covered only once, this prime implicant is absolutely essential. This makes it an "essential prime implicant" or simply an "essential." Also circle (or mark in some way) any other minterm that the essential also covers. In the first row above, (-100) covers both 4 and 12. All of the essentials, in this case, are (-100), (1-01), (1-10), and (-111) but they cover the remaining minterms as well. For the moment, we will then say that (11–) is not needed to implement the function, but we will refer back to this later on.

The simplified function is, therefore,

$$f(a,b,c,d) \;=\; \sum (4,7,9,10,12,13,14,15)$$
$$f \;=\; (b\;\overline{c}\;\overline{d}) \;+\; (a\;\overline{c}\;d) \;+\; (a\;c\overline{d}) \;+\; (b\;c\;d)$$

as confirmed by the Karnaugh map from Fig. **??**

5in 5in

Figure 1.64: The Karnaugh map

Of course, in the above prime implicant table, the minterms and the prime implicants can be denoted strictly by their decimal equivalent, leaving out the binary numbers to simplify the table.

1.37 Hazards and Propagation Delays

The signals that travel from the inputs of a digital circuit to its outputs pass through a series of gates and take a certain amount of time to propagate from the inputs to the output. This is defined as the "propagation delay" of the gate and the sum of these propagation delays through all of the gates is the total propagation delay of the circuit. Gates that receive at least one input from an output of a first-logic-level gate are considered to be the second logic level, etc. The reduction of the propagation delay in a circuit may be more important than the reduction in the total number of gates and can be accomplished by reducing the number of logic levels in the circuit. This aspect, however, will be covered during our discussion of `"multi-level"` circuits. For the time being, we want to investigate "hazards" as they occur due to propagation delay and consider circuit designs that can eliminate them.

Direct your attention again to the above Karnaugh map. Its circuit is synthesized in NAND logic from Fig. **??**.

5in 5in

Figure 1.65: NAND circuit

NAND logic.

As you will recall, we said that (11–) was not needed to realize the function since according to the "Quine-McCluskey" method all of the minterms were sufficiently covered. It is highly recommended, however, to include this prime implicant in the function even if it means that an additional NAND gate will be needed. By doing so, a possible "hazard" caused by propagation delay can be removed.

With inputs of, for example, $a = 0$, $b = 1$, $c = 1$ in a circuit and if c was directed through various logic levels and changed in some of the levels to 0, the outputs should be $c = 1$ and $\overline{c} = 0$. Due to the difference in time, however, that c and $\overline{c}$ have to travel through the various logic levels in the circuit, a situation could arise in which, at the output, $c = 1$ and $\overline{c} = 1$ or vise versa; which is not logical.

We avoid this phenomenon by changing the minimal realization of the function and adding this additional NAND gate, $\overline{(ab)}$, to the circuit. The consequence is that in the Karnaugh map additional functions are created by forming prime implicates from adjacent minterms even if they are already included in another prime implicate. In the above example (11–) connects all of the ones in the row, which connects all four of the other prime implicates as well.

The map and circuit in Fig. **??** show the function $(\overline{a}\ c) + (b\ \overline{c})$ which was designed with the hazard.

5in 5in

Figure 1.66: Circuit with hazard

The following map and circuit, Figure **??** have included the additional function $(\overline{a}\ b)$ to make the realization $(\overline{a}\ c)\ +\ (b\ \overline{c})\ +\ (\overline{a}\ b)$.

5in 5in

Figure 1.67: Circuit with no hazard

Even through it called for an additional AND gate, the above circuit insures that the input of the OR gate will never have two zeros at the same time. A 1 is generated at the output of the additional AND gate whether the input is 1 or 0 and, therefore, the hazard is eliminated.

In the function $A\ B\ +\ C\ \overline{A}\ +\ BCD$ shown in the four variable map from Figure **??**, we see that two prime implicants $AB + C\overline{A}$ emerge. We include the term BC, however, to avoid a hazard.

5in 5in

Figure 1.68: Map for $AB + C\overline{A} + BC$

1.38 Highly Unspecified Boolean Functions

A Karnaugh map with 32 rectangles is required for five variables. Six variables would have to be put into a map containing 64 rectangles, etc. The size of the map doubles with every variable that is added. On the other hand, the "Quine-McCluskey" method is appropriate for dealing with large numbers of variables. If a function, however, contains many variables but is defined by few minterms and maxterms, it will have a large number of "don't-cares" and can best be minimized by the tabulation method that we will now discuss. In this case, all the prime implicants are generated by comparing all of the minterms with all of the maxterms in the function.

In the following function, six minterms and four maxterms for a five variable function are given. This means that the function contains 22 don't-cares.

$$F^1 = \begin{array}{c} 00011 \\ 01100 \\ 01101 \\ 10000 \\ 11001 \\ 11101 \end{array} \qquad F^0 = \begin{array}{c} 00001 \\ 01010 \\ 10101 \\ 11100 \end{array}$$

We construct a table in which the maxterms denote the rows. The minterms are the columns.

$\downarrow$ 0's, $\rightarrow$ 1's	00011	01100	01101	10000	11001	11101
abcde 00001	d	$bc\overline{e}$	bc	$a\overline{e}$	ab	abc
01010	$\overline{b}e$	$c\overline{d}$	$c\overline{d}e$	$a\overline{b}\ \overline{d}$	$a\ \overline{d}e$	$ac\overline{d}e$
10101	$\overline{a}\ \overline{c}d$	$\overline{a}\ b\ \overline{e}$	$\overline{a}b$	$\overline{c}\ \overline{e}$	$b\overline{c}$	b
11100	$\overline{a}\ \overline{b}\ \overline{c}de$	$\overline{a}$	$\overline{a}e$	$\overline{b}\ \overline{c}$	$\overline{c}e$	e
	$(de), (\overline{b}d)$	$(\overline{a}c)$	$\overline{a}c, be$	$a\overline{c}, \overline{b}\ \overline{e}$	$ae,\ a\overline{c}$	be

Table 1.18: Table of highly unspecified function

Next, compare the first row (maxterm) with the first column (minterm), then with the second column, etc.

minterm: maxterm:	00011 00001	01100 00001	01101 00001	10000 00001	11001 00001	11101 00001
	d	$bc\overline{e}$	bc	a	$ab\overline{e}$	abc

The variable in the minterm that differs from the maxterm is recorded at the appropriate minterm-maxterm coordinate in the table. Note that in minterm 10000 the first and the last variables differ from the maxterm. The first variable in the minterm that differs is 1 and, therefore, a. The last variable is, however, 0 in the minterm and, therefore, $\overline{e}$ is recorded. This procedure is continued until all of the comparisons are completed.

Only one variable is needed for a minterm-maxterm match and it can be used for several matches. In the first column, for example, d appears three times: in rows 00001, 10101, and 11100. It will cover all three of them. In the first column, we could then choose d as well as another variable for row 01010: this could be either $\overline{b}$ or e. Therefore, either (de) or $(\overline{b}d)$ for column one. The same procedure is carried out for the other columns.

Finally, the terms from all of the columns are put together into one SOP expression. But, also here, we select the terms so that only minimal cover is accomplished. No term should be used twice.

We, therefore, choose

$f \ = \ (de) + (\overline{a}c) + (a\overline{c}) + (be)$

Instead of (de), we could have also chosen $(\overline{b}d)$ or we could replace $(a\overline{c})$ with $(\overline{b}\,\overline{e})$.

1.39 Problems

- **1.1**

  Prove De'Morgan's laws:

  a) $\neg(a \vee b) \ \Leftrightarrow \ (\neg a \ \wedge \ \neg b)$,

  b) $\neg(a \wedge b) \ \Leftrightarrow \ (\neg a \ \vee \ \neg b)$.

- **1.2**

  Check the following tautologies:

  a) $(p \Leftrightarrow q) \ \Leftrightarrow \ (\neg p \wedge q)$,

  b) $(p \Leftrightarrow q) \ \Leftrightarrow \ [(p \Rightarrow q) \ \wedge \ (q \ \Rightarrow \ p)]$,

  c) $(p \Leftrightarrow q) \ \Leftrightarrow \ [\neg(p \vee q) \ \vee \ \neg(\neg q \vee \neg p)]$,

  d) $[(p \ \Rightarrow q) \ \wedge \ (r \Rightarrow s)] \ \Rightarrow [(p \Rightarrow r) \ \Rightarrow \ (q \wedge s)]$.

- **1.3**

  Test if the following reasonings hold true:

  a) If I work, I earn money. If I do not work I am happy. Then, if I do not earn money I am happy.

  b) If somebody belongs to our group he can be trusted. He does not belong to our group. Then he cannot be trusted.

  c) If $x + 3 \ = \ \sqrt[2]{3 - x}$, then $x^2 + 6x + 9 = 3 - x$. But $x^2 + 6x + 9 = 3 - x$ if and only if $x = -6$ or $x = -1$. It means that -6 and -1 are the roots of the equation $x + 3 \ = \ \sqrt[2]{3 - x}$.

- **1.4**

  Allison, Brown and Clifton testify in a court of law.

  Allison says: Brown is guilty, Clifton is not.

Brown: If Allison is guilty, then so is Clifton.

Clifton: I am not guilty, but at least one of them is.

a) Is it possible for the three statements to be true?

b) If so, who is guilty?

c) One of the men's statements results from the statement of a second one. Whose statement allows us to conclude that?

d) If no one is guilty, who gave false testimony?

e) If the innocent man tells the truth and the guilty one lies, who is guilty?

- **1.5**

Five astronauts A, B, C, D, and E are prepared to go to the moon. The following conditions must be satisfied for their selection to the crew.

– A or B must go, but not together,

– C and D must either both go or stay,

– if D goes, then B also must go,

– A and C must be together (they go together or they remain together on Earth.)

–If E goes, then C and D go as well.

Denote: A - A goes, $\overline{A}$ - A does not go. Write equations and check who can go.

- **1.6**

*. Check if the following algebraic systems

$$< B, +, \cdot, \circ, i >$$

are Boolean algebras.

a) Algebra of sets where B - all subsets of set {a, b, c} it means $\{\{a,b,c\},\{a,b\},\{b,c\},\{a,c\},\{a\},\{b\},\{c\},\emptyset\}$
$\emptyset$ - empty set. Operations $+, \cdot, -$ - are defined respectively as $\cap$ (product), $\cup$ (sum), complement, of sets and $0 = \emptyset$. $i = \{a,b,c\}$.

b) Set of natural numbers with operations of addition and multiplication.

c) Four element system $B = \{1,2,3,6\}$ with elements 0 and i respectively 1, 6, operations $+, \cdot, -$ - defined as follows:
$a + b =$ least common multiple of numbers a and b,
$a \cdot b =$ greatest common divisor of numbers a and b,
$\overline{a} = 6/a$.

d) Algebra of binary pairs where $B = \{0,1\} \times \{0,1\}$ is a Cartesian product of elements of two element Boolean algebra. As element 0 we can take the pair $< 0, 1 >$, as element i the pair $< 1, 0 >$. The operations are defined as follows:
$< a_1, a_2 > + < b_1, b_2 >=< a_1 + b_1, a_2 \cdot b_2 >$,
$< a_1, a_2 > \cdot < b_1, b_2 > = < a_1 \cdot b_1, a_2 + b_2 >$,
$< a_1, a_2 > = < \overline{a}, \overline{a_2} >$.

e) Algebra of binary n-tuples, where $B = \{0,1\} \times \{0,1\} \times ... \times \{0,1\}$ $0 = < 0, 0, ..., 0 >$,, $i =< 1, 1, ..., 1 >$,
$< a_1, a_2, ..., a_n > + < b_1, b_2, ..., b_n >= = a_1 + b_1, a_2 + b_2, ..., a_n + b_n >$
$< a_1, b_2, ..., a_n > \cdot < b_1, b_2, ..., b_n >= =< a_1 \cdot b_1, a_2 b_2, ..., a_n \cdot b_n >$
$\overline{< a_1, a_2, ..., a_n >} = < \overline{a}, \overline{a_2}, ...\overline{a_n} >$.

- **1.7**

Prove that the algebraic system $< K, +, - >$, where $+$ is a two argument operation, and - is a one argument operation, for which the following axioms are satisfied.

$$\forall a, b, c \in K, \quad a + b = b + a \tag{1.20}$$

$$(a + b) + c = a + (b + c) \tag{1.21}$$

$$\overline{(\overline{a} + \overline{b})} + \overline{(\overline{(a + b)} + b)} = a \tag{1.22}$$

is a Boolean algebra.

- **1.8**

An eight-element Boolean algebra is given on the set $B = \{1, 2, 3, 5, 6, 10, 15, 30\}$:

a) define the operations $+$ and $\cdot$.

b) define the negation, -, of each element and the elements 0 and i.

- **1.9**

A Boolean algebra $< B, \cup, \cap, -, 0, 1 >$ is given. Prove that the algebraic system $< B, \cap, \cup, -, 1, 0 >$ is a Boolean algebra as well.

- **1.10**

Prove that a three-element Boolean algebra cannot exist.

- **1.11**

For the algebraic system

$$< B, +, \cdot, -, o, i >; \ o, i \in B;$$

where $\forall\ a, b, c \in B$

$a + b \in B$, (1.1a)

$a + b = b + a$, (1.2a)

$a(b + c) = ab + ac$, (1.3a)

$a + o = a$, (1.4a)

$a \cdot b \in B$, (1.1b)

$a \cdot b = b \cdot a$ (1.2b)

$a + bc = (a + b)(a + c)$, (1.3b)

$a \cdot i = a$ (1.4b)

$\forall a \in B\ \exists \overline{a} \in B$

$a + a = i$, (1.5a)

$a \cdot a = o$ (1.5b)

Prove the following theorems:

$a + (b + c) = (a + b) + c$, (1.6a)

$a + ab = a$, (1.7a)

$a + ab = a + b$, (1.8a)

$a + a = a$, (1.9a)

$a + i = i$, (1.10a)

$\overline{a + b} = \overline{a}\ \overline{b}$, (1.11a)

$\overline{o} = i$, (1.12a)

a = a (1.13)

$a(bc) = (ab)c$, (1.6b)

$a(a + b) = a$, (1.7b)

$a(a + b) = ab$, (1.8b)

$a \cdot a = a$, (1.9b)

$a \cdot o = o$ (1.10b)

$\overline{ab} = \overline{a} + \overline{b}$ (1.11b)

$\overline{i} = o$, (1.12b)

$\overline{\overline{a}} = a$ (1.13)

- **1.12**

  Prove the following theorems:

  a) In any Boolean algebra there exist exactly one element 0 and exactly one element i.

b) In any Boolean algebra for each element x there exists exactly one element x such that $x+\overline{x}=i$ and $x.\overline{x}=0$.

- **1.13**

  Verify the following identities:

  a) $(x+\overline{y}+xy)(x+\overline{y})\overline{x}y=0$,

  b) $(x+\overline{y}+x\overline{y})(xy+\overline{x}z+yz) \;=\; xy+\overline{x}\,\overline{y}z$,

  c) $(ab+c+d)(\overline{c}+d)(\overline{c}+d+e)=ab\overline{c}+d$,

  d) $(\overline{a}+b+\overline{c})=\overline{a}(\overline{b}+\overline{c})+bc+a\overline{c}$,

  e) $ab+ac+ab=(a+b)(a+c)(b+c)$.

  f) $b\overline{c}+\overline{a}c+\overline{a}b=b\overline{c}+\overline{a}c$.

- **1.14**

  Prove that the following identities are true.

  a) $\overline{w}+\overline{z}+(x+wz)(y+z)=1$,

  b) $xy+\overline{x}y\overline{z}+yz=y$,

  c) $(x\overline{y}+z)(\overline{x}+y)\overline{z}=0$,

  d) $x\overline{y}+z+(\overline{x}+y)\overline{z}=1$.

- **1.15**

  Prove the identities:

  a) $(a+b)[x(a+y)+\overline{x}(a+c)+(y+z)za]=a+b(xy+\overline{x}c)$,

  b) $ab(z+x)+\overline{a}b(\overline{x}+z)+x(\overline{a}+b)=\overline{a}(x+b)+b(x+z)$,

  c) $x[(y+b)x+y(z+\overline{x}y)]+\overline{x}[(y+c)x+(a+\overline{a}\,\overline{x})y] \;=\; y(a\overline{b}+\overline{a}\,\overline{x})$,

  d) $a\overline{b}(xy+\overline{x}y\overline{z})+\overline{x}[\overline{a}y+\overline{b}(x+y)a]=y(a\overline{b}+\overline{a}\,\overline{x})$.

- **1.16**

  Simplify the following expressions.

  a) $(x+\overline{y})[xyz+\overline{y}(z+x)]+xy\overline{z}(x+\overline{x}y)$,

  b) $(x+\overline{y}\,\overline{x})[xz+x\overline{z}(y+\overline{y})]$,

  c) $x\overline{z}y+(x\overline{z}y+z\overline{x})[y(z+x)+\overline{y}z+\overline{y}x\overline{z}]$,

  d) $(a\downarrow b)\uparrow[(a+(\overline{a}+xz)\overline{a})(\overline{z}+a)]$.

- **1.17**

  Prove:

  a) $ifa\oplus b=c$, then $a\oplus b\oplus c=0$,

b) $ifa \oplus b = c$, then $a \oplus c = b$.

- **1.18**

  Verify the following identities.

  a) $ab + \overline{a}c + bc = ab + \overline{a}c$,

  b) $(a + b)(\overline{a} + c)(b + c) = (a + b)(\overline{a} + c)$.

- **1.19**

  Prove the following identities.

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

  b) $ab + \overline{a}\ \overline{b} + bcd = ab + \overline{a}\ \overline{b} + \overline{a}cd$,

  c) $\overline{a}d(\overline{b} + c) + (b + \overline{c})\overline{a}\ \overline{d} + (\overline{b} + c)(b + \overline{c}) = (\overline{a}\ \overline{d} + \overline{b} + c)(\overline{a}d + b + \overline{c})$,

  d) $bc + abd + a\overline{c} = bc + a\overline{c}$,

  e) $abc + a\overline{b}\ \overline{c} + b\ \overline{c}d + \overline{b}cd + ad = abc + a\overline{b}\ \overline{c} + \overline{b}cd + b\overline{c}d$,

  f) $ab + \overline{c}\ \overline{d} + \overline{a}\ b\ c\ \overline{d} + a\overline{b}\ \overline{c}d = (a + \overline{d})(b + \overline{c})$.

- **1.20**

  Verify the following identities.

  a) $a(b \oplus c) = ab \oplus ac$,

  b) $a \oplus (b \oplus c) = (a \oplus b) \oplus c$,

  c) $a \downarrow (b \downarrow c) = (a \downarrow b) \downarrow c$,

  d) $a \downarrow b = b \downarrow a$.

  e) $a \oplus \overline{b} = b \oplus \overline{a} = \overline{a \oplus b}$.

- **1.21**

  Prove that if $ab = ac$ and $a + b = a + c$, then $b = c$.

- **1.22**

  Express the function of $f = x \oplus y$ using Sheffer's operators (operators $\uparrow$ or NAND).

- **1.23**

  Prove that the following functions constitute functionally full systems:

  a) Pierce function $(a \downarrow b)$,

  b) Sheffer function $(a \uparrow b)$,

  c) Implication and 0 $(a \leftarrow b$ and $0)$,

  d) Nonsymmetric difference and 1 $(a \triangle b$ and $1)$.

- **1.24**

  Write the following functions in their canonical forms:

  a) $x + y$,
  b) $\overline{(x+y)} \ \cdot \ (x \leftarrow y)$,
  c) $(x \sim y)\overline{(z \leftarrow t)}$.

- **1.25**

  Prove that functions g and f are dual.

  a) $f = 0; \ g = 1$,
  b) $f = x; \ g = x$,
  c) $f = \overline{(x+y)}; \ g = \overline{(xy)}$,
  d) $f = x \oplus y; \ g = \overline{(x \oplus y)}$.

- **1.26**

  Prove that the following holds true for any given Boolean function.

  a) $x_1 f(x_1, x_2, ..., x_n) \ = \ x_1 f(1, x_2, ..., x_n)$,
  b) $x_1 \ + \ f(x_1, x_2, ..., x_n) \ = \ x_1 \ + \ f(0, x_2, ..., x_n)$,

- **1.27**

  Prove by induction, that for each n, the function

  $f(x_1, x_2, ..., x_n) \ = \ x_1 \oplus x_2 \oplus ... \oplus x_n + 1$

  is equal to 1 iff an odd number of variables $x_1, x_2, ...x_n$ has the value 1.

  ?????????????????

- **1.28**

  Prove that each Boolean function can be represented in the form:

  a) $f(x_1 x_2 ... x_n = (x_1 + f(0, x_2, ..., x_n))(\overline{x_1} + f(1, x_2, ...x_n))$,
  b) $f(x_1 x_2 ... x_n = x_1 \ f(1, x_2, ..., x_n)) \ + \ \overline{x_1} + f(0, x_2, ...x_n)$.

- **1.29**

  The Relation $\leq$ is defined as follows:

  $$x \leq y \quad iff \quad x + y = y.$$

  a) Prove that the definition

  $$x \leq y \ iff \ x = xy$$

  is equivalent to the original one.

b) Prove that $x \leq y + z$ iff $x \leq y$ or $x \leq z$.

c) Prove that $x \leq yz$ iff $x \leq y$ and $x \leq z$.

d) Prove that if $x_1 \leq y_1$ and $x_2 \leq y_2$, then $x_1 + x_2 \leq y_1 + y_2$ and $x_1 x_2 \leq y_1 y_2$

- **1.30**

Prove that each function $f(x_1, ..x_n)$ can be written in the form:

$$f(x_1, ...x_n) \; = \; g_0 \oplus g_1 x_1 \oplus g_n x_n \oplus g_{n+1} x_1 x_2 \oplus .... \oplus g_{2^n - 1} x_1 \cdots x_n$$

,

where $g_i \in \{0, 1\}$; $i = 0, 1, 2, ..., 2^n - 1$.

Give examples for:

a) $f(x_1,\ x_2) = x_1 \ + \ x_2$.

b) $f(x_1,\ x_2, x_3) = x_1 x_2 \ + \ x_3 \ + \ \overline{x_1} x_3$.

c) $f(x_1,\ x_2) = x_1 \ \uparrow \ x_2$.

- **1.31**

The standard sum of products form for a function is given by $f(x_1, ..., x_2) = \sum_{i=0}^{2^n - 1} a_i k_i$, where k_i are full products.

a) Find the relation: $g_i \; = \; h_1(a_0, a_1, ..., a_{2^n - 1}); \quad i = 0, 1, ..., 2^n - 1$,
where g_i are the coefficients of the expansion of the function from the previous problem.

b) Find the relation:

$$a_i \; = \; h_2(g_0, g_1, ..., g_{2^n - 1}); \quad i = 0, 1, ..., 2^n - 1$$

- **1.32**

Let us denote:

$$x_j{}^{e_j} \; = \; \begin{cases} \overline{x_j}, & when \ e_j = 1; \\ x_j, & when \ e_j = 0 \end{cases} \tag{1.23}$$

Prove that:

$$x_1{}^{e_1} \oplus x_2{}^{e_2} \oplus ... \oplus x_n{}^{e_n} \; = \; \begin{cases} x_1 \oplus x_2 \oplus ... \oplus x_n, & if \bigoplus_{i=1}^{n} e_i \; = \; 0, \\ \overline{x_1 \oplus x_2 \oplus ... \oplus x_n} & if \bigoplus_{i=1}^{n} e_i \; = \; 1, \end{cases} \tag{1.24}$$

- **1.33**

Solve the Boolean equations:

a) $ax + b(x + a) = x$,

b) $a(x + b\overline{x}) = bx + a\overline{b}\,\overline{x}$, where x is the dependent variable.

c) $ax_1 + bx_2 + \overline{a}(b\overline{x_2} + b\overline{x_1}) = (\overline{a} + \overline{b})x_1x_2 + a\overline{x_2}(b + \overline{x_1})$,

d) $x_2(a + x_1) + \overline{a}bx_2 = \overline{x_1}(\overline{a} + \overline{b}) + ax_1\overline{x_2}$,
where x_1 and x_2 are the dependent variables.

- **1.34**

Solve the following sets of Boolean equations. Dependent variables are x_1, x_2 and x_3.

a) $\overline{b}(\overline{x_1} + \overline{a}x_2) = b\overline{x_2} + \overline{b}x_1x_2$,
$\overline{a}b + x_2(x_1 + b) = \overline{b}\,\overline{x_1} + b(x_2 + ax_1)$;

b) $a\overline{b} + b(\overline{x_1}\,\overline{x_2} + x_2x_3) + \overline{b}\,\overline{x_2}\,x - 3 = \overline{x_3}(b + x_1 + \overline{a}x_2) + \overline{b}x_3$,
$a(b + \overline{x_2}\,\overline{x_3}) + \overline{a}x_3 = \overline{a}b + a[x_3(x_1 + x_2) + \overline{b}\,\overline{x_1}\,\overline{x_2}]$;

c) $x_1x_2 = ca + ab$,
$x_2x_3 = a + bc$,
$x_1x_3 = b + ac$;

d) $x_1[abcx_2 + \overline{a}(x_2 + \overline{b})] + b\overline{c}\,\overline{x_1} = \overline{a}(x_1 + \overline{a}\,\overline{b}\,\overline{c}) + bx_1\overline{x_2} + a\overline{x_1}x_2$,
$\overline{a}(\overline{b}\,\overline{x_1} + cx_1) + abc\overline{x_2} = x_1(a + \overline{b}x_2 + b\overline{x_2}) + \overline{a}bc$;

e) $a(x_2 + \overline{x_1}) + \overline{x_2} = \overline{a}\,\overline{b}x_2 + x_1x_2$,
$x_2(ab + c\overline{b}) = bx_2 + cx_1$,
$c(a\overline{x_2}bx_1) + x_1x_2 = a\overline{b}x_1 + ab$.

- **1.35**

Given the function $y(a, b, c) = ab\overline{c} + a\overline{b}c + \overline{a}bc$. What other functions or constants shall be added to form a functionallly full system?

- **1.36**

Use any logic circuits simulator to verify tautologies from problems **??**, **??**, **??**, **??**, **??**, **??**.

Hint. Show that for all combinations of input signals the respective networks have constant value 1 at their output.

1.40 Answers and Solutions to Problems

- **??**. The proof can be carried out using the truth table shown in Table **??**.
- **??**. a) no, b) yes, c) yes, d) no.

a	b	$\neg a$	$\neg b$	$\neg(a \vee b)$	$\neg a \vee \neg b$	$\neg(a \wedge b)$	$\neg 1 \vee \neg b$
0	0	1	1	1	1	1	1
0	1	1	0	0	0	1	1
1	0	0	1	0	0	1	1
1	1	0	0	0	0	0	0

Table 1.19: Table to Problem **??**

- **??**. Denote: W - I work, E - I earn money, H - I am happy. The reasoning can be described as follows:

$$[(W \Rightarrow E) \ \wedge \ (\neg W \Rightarrow H)] \ \Rightarrow \ (\neg E \Rightarrow H).$$

Now, using the truth table from Table **??** we test than the reasoning is true.

1	2	3	4	5	6	7	8
W	E	H	$W \Rightarrow E$	$\neg W \Rightarrow H$	$4 \wedge 5$	$\neg E \Rightarrow H$	$6 \Rightarrow 7$
0	0	0	1	0	0	0	1
0	0	1	1	1	1	1	1
0	1	0	1	0	0	1	1
0	1	1	1	1	1	1	1
1	0	1	0	1	0	1	1
1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1

Table 1.20: Table to Problem **??**

b) not true, c) not true

- **??**. Denoting: A - Allison is not guilty, B- Brown is not guilty, C - Clifton is not guilty, we have:

The testimony of Allison: $\neg B \wedge C$.

The testimony of Brown : $\neg A \Rightarrow \neg C$.

The testimony of Clifton: $C \wedge (\neg A \vee \neg B)$.

a) It is possible that all these statements are true.

b) Brown is guilty.

c) After checking all possible implications we conclude that only one of them is always true.

$$(\neg B \wedge C) \ \Rightarrow \ [C \wedge (\neg A \vee \neg B)]$$

which means that Allison's testimony implies Clifton's testimony

d) Allison and Clifton.

e) Allison and Clifton.

- ??. The conditions have the following form:
  $A\overline{B} + B\overline{A}$,
  $CD + \overline{C}\ \overline{D}$,
  $D \Rightarrow B$, equivalent to $\overline{D} + B$,
  $AC + \overline{A}\ \overline{C}$,
  $E \Rightarrow CD$, equivalent to $\overline{E} + CD$
  We write the logic function

  $$F = (A\overline{B} + B\overline{A})(CD + \overline{C}\ \overline{D})(\overline{D} + B)(AC + \overline{A}\ \overline{C})(\overline{E} + CD) = 1$$

  This function can be equal to 1 for the values of variables corresponding to the selected crew. After solving, we get

  $$F = \overline{A}\ \overline{C}\ \overline{D}\ \overline{E}\ B \quad = \quad 1$$

  which means that only B can go to satisfy the requirements.

- ??.

  a) It must be checkeded whether axioms 1 - 5 of Boolean algebra are satisfied (see p. XXXXX1.2) Axioms 1, 2, 4 are not fulfilled which results directly from the known definitions of operations $\cup, \cap$ and the definition of empty set 0. Complement of set $x \in B$ is set $\overline{x} = \{a, b, c\}$ - x. From these, and from the definition of operations $\cup$, $\cap$ and set $\emptyset$ results axiom 5.

     Truthfullness of axiom 3 can be proved by use of truth functional calculus (we leave that to the reader). It was then shown that the system considered is a Boolean algebra.

  b) No. Axiom 3b is not fulfilled.

  c) Yes.

  d) Yes.

  e) Yes.

- ??. Let us introduce definitions: $a \cdot b = \overline{\overline{a} \ + \ \overline{b}}$, $0 = a \cdot \overline{a}$, $i = a + \overline{a}$. It must be checked whether such defined operations satisfy the axioms of a Boolean algebra. We leave it to the reader.

- ??. Compare with problem **??**c.

- ??. The proof results from replacement of axioms with index a into axioms with index b and vice versa. This is the dual algebra for the initial one.

- ??. Let us assume that $B = \{0,1,A\}$ while $o = 0$ and $i = 1$. We will look now for the negation of element A. Three particular cases are possible:

  1. $\overline{A} = 0$, Then $A + \overline{A} = A + 0 = A$ (Axiom 4) which is not consistent with Axiom 5.
  2. $\overline{A} = 1$, Then $A \cdot \overline{A} = A \cdot 1 = A$, which contradicts axiom 5.
  3. $A = \overline{A}$, Then $A + \overline{A} = A + A = (A + A) \cdot 1 = (A + A)(A + \overline{A}) == A + A \cdot \overline{A} = A + 0 = A$

     By subsequent transformations, the axioms 4b, 5a, 3b, 5b and 4a were respectively applied. At last we get $A + \overline{A}$ which is inconsistent with axiom 5a. We have then proved, that it is impossible to find the negation of element A in such a way, that all axioms of Boolean algebra were satisfied. The three element Boolean algebra cannot exist.

- ??. **Hint**. Assume that there exist two different elements o and o^*. Now you must prove that $o = o^*$ must hold. Do the same for i.

- ??.

  a) $(x + \overline{y} + xy)(x\overline{x}y + y\overline{y}x) = (x + \overline{y} + xy)(0 + 0) = 0$

  b) $(x + \overline{y} + x\overline{y})(xy + \overline{x}z + yz) == (x + \overline{y})(xy + \overline{x}z + yz) = xy + x\overline{y}y + x\overline{x}z + \overline{x}\ \overline{y}z + xyz + \overline{y}yz = xy + xyz + \overline{x}\ \overline{y}z = xy + \overline{x}\ \overline{y}z$,

  c) Yes.

  d) Yes.

  e) Yes.

  f) Yes.

- ??.

  a) $\overline{w} + \overline{z} + wz + xz + xy = (\overline{w} + \overline{z}) + \overline{(\overline{w} + \overline{z})} + xz + xy == 1 + xz + xy = 1$,

  d) $x\overline{y} + z + (\overline{x} + y)\overline{z} = x\overline{y} + z + \overline{\overline{\overline{x} + y} + z} = x\overline{y} + z + \overline{x\overline{y} + z} = 1$

- ??.

  d) $a\overline{b}(xy + \overline{x}y\overline{z}) + \overline{x}[\overline{a}y + \overline{b}(x + y)a =$
  $= a\overline{b}xy + \overline{a}\ \overline{b}\ \overline{x}\ y\ \overline{z} + \overline{x}\ \overline{a}y + \overline{x}x\ \overline{b}a + \overline{x}\ a\ \overline{b}y =$
  $= a\overline{b}y(x + \overline{x}) + a\overline{b}\ \overline{x}y(\overline{z} + 1) + \overline{a}\ \overline{x}y == a\overline{b}y(\overline{x} + 1) + \overline{a}\ \overline{x}y = y(a\overline{b} + \overline{a}\ \overline{x})$.

- ??.

  a) $x + z\overline{y}$

  b) x

  c) $xy\overline{z} + z\overline{x}$

d) $a+b$

- ??.

a) $a\oplus b=c$, $a\oplus b\oplus c=a\oplus b\oplus a\oplus b$. Because $a\oplus a=0$ (show!) and $0\oplus 0=0$, then $a\oplus b\oplus c=0$.

b) $a\oplus b=c$, $a\oplus c=a\oplus a\oplus b=0\oplus b=b$.

- ??.

a) $ab+\overline{a}\,\overline{b}+bc=ab+\overline{a}\,\overline{b}+bc(a+\overline{a})=ab+\overline{a}\,\overline{b}+\overline{a}bc+abc\ =\ ab(1+c)\ +\ \overline{a}(\overline{b}+bc)=$
$=ab+\overline{a}(\overline{b}+c)=ab+\overline{a}\,\overline{b}+\overline{a}c$,

c) $\overline{a}d(\overline{b}+c)+(b+\overline{c})\overline{a}\,\overline{d}+(\overline{b}+c)(b+\overline{c})=$
$=\overline{a}\,\overline{b}d+\overline{b}\,\overline{c}+\overline{a}dc+bc+\overline{a}\,\overline{d}b+\overline{a}\,\overline{d}\,\overline{c}+b\overline{b}+c\,\overline{c}+\ +\overline{a}\,\overline{d}\,\overline{a}d=\overline{a}\,\overline{d}(\overline{a}d+b+\overline{c})+$
$\overline{b}(\overline{a}d+b+\overline{c})++c(\overline{a}d+b+\overline{c})\ =\ (\overline{a}d+b+\overline{c})(\overline{a}\,\overline{d}+\overline{b}+c)$.

- ??.

a) $a(b\oplus c)=a(b\overline{c}+\overline{b}c)=ab\overline{c}+a\overline{b}c=$
$=ab\overline{c}+\overline{a}ab+a\overline{b}c+\overline{a}ac=ab(\overline{a}+\overline{c})+ac(\overline{a}+\overline{b})=$
$=ab\,\overline{ac}+ac\,\overline{ab}=ab\oplus ac$,

b) Yes,

c) No,

d) Yes.

- ??. $ab\ =\ ac$,

$\overline{ab}\ =\ \overline{ac}$,

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

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

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

$a\oplus b\ =\ a\oplus c$,

$a\oplus a\oplus b\ =\ a\oplus a\oplus c$,

$b\ =\ c$

- ??. Applying De'Morgan's Law we have:
$f\ =\ x\oplus y\ =\ x\overline{y}+\overline{x}y\ =\ \overline{\overline{\overline{x\overline{y}}}\,\overline{\overline{\overline{x}y}}}=((x|y)|x)|((x|y)|y)$.

- ??. To prove that the function generates functionally full system, it is sufficient to show that Boolean sum, product, and negation can be created from these functions:

a) $a\downarrow b\ =\ \overline{a+b}$,
$\overline{a}=a\downarrow a$,
$a+b=(a\downarrow b)\downarrow(a\downarrow b)$,
$a\cdot b=(a\downarrow a)\downarrow(b\downarrow b)$;

b) $a|b \; = \; \overline{ab},$

$\overline{a} \; = \; a|a,$

$a+b \; = \; (a|a)|(b|b),$

$a \cdot b \; = \; (a|b)|(a|b);$

c) $a \Rightarrow b \; = \; \overline{a} + b,$

$\overline{a} = a \Rightarrow 0,$

$a+b \; = \; (a \Rightarrow 0) \Rightarrow b,$

$a \cdot b \; = \; [a \Rightarrow (b \Rightarrow 0)] \Rightarrow 0;$

d) $a \triangle b = a\overline{b},$

$\overline{a} = 1 \triangle a,$

$a+b = 1 \triangle [(1 \triangle a) \triangle b],$

$a \cdot b \; = \; a \triangle (1 \triangle b).$

- **??**.

  a) $x+y = x(y+\overline{y}) + y(x+\overline{x}) = \overline{x}y + x\overline{y} + xy,$

  b) $\overline{x}\ \overline{z}\ \overline{y} + \overline{x}\ \overline{z}\ y$, XXXXX???????

  c) $\overline{t}\ \overline{x}\ \overline{y}z + \overline{t}xyz.$

- **??**. The dual function to function $f(x_1, x_2, ..., x_n)$

  $y_D = f(x_1, x_2, ...x_n),$

  a) $f = 0; f_D = f = 1 = g,$

  b) $f(x) = x; fPVDPV = f(x) = x = x = g.$???????????

- **??**. Let us present function $f(x_1, x_2, ...x_n)$ in the canonical form. We can then write

  $f(x_1, x_2, ..., x_n) = \sum_{i=0}^{2^n-1} a_i K_i = a_i K_i^{x_1} + \sum_{i=p+1}^{2^n-1} a_i K_i^{x_1}$

  where: K_i - full product,

  $K_i x_1$ - full product including literal x_1,

  $K_i V x_1$ - full product including literal x_1,

  $0 < p < 2^n - 1;$

  of course, $x_1 K_i^{x_1} = 0; \quad K_i x_1 (x_1 = 1) = 0,$

  and $x_1 K_i^{x_1} \; = \; K_i^{x_1} = x_1 K'^{x_i} = x_1 \cdot 1 \cdot K'^{x_1} = x_1 K_i (x_1 = 1),$

  where: K_i' - full product after removal of literal x_1.

  $K_i(x_1 = 1), K_i(x_1 = 1)$ - respective full products by condition

  ???????????

  i.e., $x_1 K_i^{\overline{x_1}} + x_1 K_i^{x_i} = x_1 K_i^{x_i}(x_1 = 1) + x_1 K_i^{x_1}(x_1 = 1)$

  for $i = 0, 1, ..., 2^n - 1.$

Then, $x_1 f(x_1, x_2, ..., x_n) = x_1(\sum_{i=0}^{p} a_i K_i^{x_1} + \sum_{i=p+1}^{2^n-1} a_1 K_i^{x_1} =$

$x_1(\sum_{i=0}^{p} a_i K_i^{\overline{x_1}}(x_1 = 1) + \sum_{i=p+1}^{2^n-1} a_1 K_i^{x_1}(x_1 = 1)) =$

$x_1 f(1, x_2, ..., x_n)$.

[b)] The proof is similar to the proof in part a.

- **??**. We will prove by induction.

  $n = 2$.

  $f(x_1, x_2) = x_1 \oplus x_2 = x_1\overline{x_2} + \overline{x_1}x_2$.

  As we see, the function is equal to 1 only when $x_1 = 1$, or $x_2 = 1$ and $x_1 x_2 + x_1 x_2 = 0$. The first inductive condition is then satisfied. We will assume now that for $n = p$, the theorem is true, then $f(x_1, x_2, ..., x_p) = x_1 \oplus ... \oplus x_p = f_p$ has value 1, because an odd number of variables has value 1.

  Let us then consider the function

  $f(x_1, x_2, ..., x_p, x_{p+1} == x_1 \oplus x_2 \oplus ... \oplus x_p \oplus x_{p+1} = f_{p+1}$

  The following four cases are possible:

  a) $f_p = 1$ (odd number of variables $x_i = 1$) and $x_{p+1} = 1$,
     when $f_{p+1} = 1 \oplus 1 = 0$, but even number of variables is equal to 1;
  b) $f_p = 1$ and $x_{p+1} = 0$, then, $f_{p+1} = 1 \oplus 0 = 1$;
  c) $f_p = 0$ and $x_p = 0$, $f_{p+1} = 0 \oplus 0 = 0$;
  d) $f_p = 0$, $x_{p+1} = 1$, $f_{p+1} = 0 \oplus 1 = 1$.

  It is easy to check, that in all these cases the assumption of the theorem is satisfied, then the second inductive condition is satisfied.

- **??**. The proof is similar to the proofs in problem **??**.

- **??**.

  a) It is sufficient to show that $x + y = y$ is equivalent with $x = xy$. We assume that $x + y = y$ ($x \leq y$ acccording to definition 1), then $xy = x(x + y) = x + xy = x(1 + y) = x, xy = x$, ($x \leq y$ according to definition 2). The proof from the other side is similar.
  b) From definition 2

     $x \leq y + z \Leftrightarrow x = x(y + z),$

     $x = xz + xy$

     This equality is satisfied when

     $x = xz$ or $x = xy$,

     then, if $x \leq z$ or $x \leq y$;

d) $x_1 \le y_1 \Leftrightarrow x_1 = x_1 y_1$,
$x_2 \le y_2 \Leftrightarrow x_2 = x_2 y_2$,
i.e.,
$x_1 x_2 = x_1 y_1 x_2 y_2$,
$x_1 x_2 = x_1 x_2 y_1 y_2$,
i.e., $x_1 x_2 \le y_1 y_2$.
The second part of the proof is dual.

- **??**. Each Boolean function can be written in a form $f(x_1, x_2, ..., x_n) = \sum_{i=0}^{2^n-1} a_i K_i$. It can be proved that the logic sum in this expression can be replaced by a modulo 2 sum (show this!!). Then $f(x_1, x_2, ..., x_n) = \bigoplus_{i=0}^{2^n-1} a_i K_i$, where symbol $\bigoplus$ denotes a modulo 2 sum. Full product K_i can include complemented literals. Making use of rule $\overline{a} = 1 \oplus a$ and multiplying expressions (using rule $a(b \oplus c) = ab \oplus ac$ and reducing $(a \oplus a = 0,\ 0 \oplus a = a)$ we obtain the required form of the function.

  a) $x_1 \oplus x_2 \oplus x_1 x_2$,

  b) $x_3 \oplus x_1 x_2 \oplus x_1 x_2 x_3$.

  c) $1 \oplus x_1 x_2$.

- **??**. We will assume the following notation for the coefficients:

$$g_{J_1, J_2, ..., J_n} = \begin{cases} 1, & \textit{when } x_i \textit{ remains in the product;} \\ 0, & \textit{otherwise} \end{cases} \tag{1.25}$$

$$J_1 = \begin{cases} 1, & \textit{when } x_i \textit{ remains in the product;} \\ 0, & \textit{otherwise} \end{cases} \tag{1.26}$$

$a_{\alpha_1, \alpha_2, ..., \alpha_n}$

$$\alpha_1 = \begin{cases} 1 & \textit{when } x_i \textit{ remains in the product,} \\ 0 & \textit{when } \overline{x_i} \textit{ remains in the product} \end{cases} \tag{1.27}$$

  a) These rules can be proved using the proof for problem **??**. Applying the above notation we obtain

$$g_{J_1, J_2, ..., J_n} = \bigoplus_{\alpha_1 \le J_1, ... \alpha_n \le J_n} a_{\alpha_1, \alpha_2, ... \alpha_n}$$

where $\bigoplus_{\alpha_1 \le J_1, ..., \alpha_n \le J_n}$ means modulo 2 sum over all coefficients separately. Relation $\le$ was defined in problem **??**. For instance, in the example from problem **??**a:
$a_{00} = 0, \quad g_{00} = a_{00} = 0$,
$a_{01} = 1, \quad g_{01} = a_{00} \oplus a_{01} = 1$,
$a_{10} = 1, \quad g_{10} = a_{00} \oplus a_{10} = 1$,
$a_{11} = 1, \quad g_{11} = a_{00} \oplus a_{01} \oplus a_{10} \oplus a_{11} = 1$.

b) $a_{\alpha_1,\alpha_2,\ldots,\alpha_n} = \sum_{J_1 \le \alpha_1,\ldots J_n \le \alpha_n} g_{J_1,J_2,\ldots,J_n}$.
For example, from problem **??**c:
$g_{00} = 1, \quad a_{00} = g_{00} = 1,$
$g_{01} = 0, \quad a_{01} = g_{00} \oplus g_{01} = 1,$
$g_{10} = 0, \quad a_{10} = g_{00} \oplus g_{10} = 1,$
$g_{11} = 1, \quad a_{11} = g_{00} \oplus g_{10} \oplus g_{01} \oplus g_{11} = 0.$

- **??**. Advise: ${x_j}^{e_j} = x_j \oplus e_j$.
- **??**. A boolean equation $f(x_1, ..., x_n, a_1, ..., a_m) = g(x_1, ..., x_n, a_1, ..., a_m)$,

where:

$x_1, ..., x_n$ are dependent variables,

$a_1, ..., a_n$ are parameters,

can be transformed to the form

$$\overline{f}\,\overline{g} + fg = 1,$$

because this equation will be satisfied iff f and g are equal to 1 or when f and g are equal to 0.

a) $f = ax + b(x + a)$
$g = x$
We will solve the equation using Karnaugh maps (see p.2.1?????)
We will treat functions f and g as dependent in variable x and parameters a and b. In figure **??**a and b the functions $f(x, a, b)$ and $g(x, ab)$ are shown. Figure **??**c shows function $\overline{f} \cdot \overline{g}$, Figure **??**d – function $f \cdot g$ and Figure **??**e – function $\overline{f}\,\overline{g}\,fg$. In order to find a solution the function $\overline{f}\,\overline{g}$ must be reduced to the form

$$\overline{f}\,\overline{g} + fg = r_1 + r_2 + ... + r_1,$$

where a characteristic for functions r_i is that their Karnaugh maps have in each row one and only one minterm.

Figure **??** f-j shows functions $r_1 - r_4$ (all possible functions). From each function r_i one solution of an equation can be found. Because in each row there exists a minterm – function $x(a, b)$ is defined. Because in each row exactly one minterm exists - function x can be defined univocally.
From function r_1 we see that $x = 1$ when $ab = 1$, i.e. $x_1 = ab$. From function $r_2 : x_2 = a + b$; From function $r_3 : x_3 = b$; From function $r_4 : x_4 = a$. We have found, then, four solutions to the problem.

5in 5in

Figure 1.69: Figure 1.1 Solution of an Equation From Problem **??**a

5in 5in

Figure 1.70: Figure 1.2 Solution to problem **??**b

b) The table of function $\overline{f}\ \overline{g} + fg$ for this equation is presented in Figure **??**a. We see, that in row ab =10 there is no minterm. It can be then concluded that there exists no solution for all values of a and b. There exists no such x, which for $a =$ 1 and $b = 0$ will satisfy the equation.

If we assume condition $a\overline{b} =\neq 1$, then in row 10 we can put don't cares (value of function in this row is not important - see Figure **??**b) Now treating don't cares once as true minterms and once as false minterms, we can find four solutions of the equation:

$$x_1 = ab, \quad x_2 = ab + \overline{a}\ \overline{b}, \quad x_3 = a, \quad x_4 = a + \overline{b}$$

by the condition $ab = 0$.

c) $x_1^1 = 1, \quad x_2^1 = \overline{a}b + a\overline{b}$
$x_1^2 = \overline{a} + b, \quad x_2^2 = \overline{a}b + a\overline{b}$

d) $x_1^1 = \overline{a}\ \overline{b}, \qquad x_1^2 = a\overline{b} + \overline{a}b$
$x_1^2 = \overline{a}, \qquad x_2^2 = a\overline{b}.$

- **??**. Sets of equations:

$f_1(x_1, ..., x_n, a_1, ..., a_m) = g_1(x_1, ..., x_n, a_1, ..., a_m),$

$f_2(x_1, ..., x_n, a_1, ..., a_m) = g_2(x_1, ..., x_n, a_1, ..., a_m),$

. . .

$f_k(x_1, ..., x_n, a_1, ..., a_m) = g_k(x_1, ..., x_n, a_1, ..., a_m),$

can be, analogously to the single equation, reduced to the form

$$\begin{aligned} \overline{f_1}\ \overline{g_1} + f_1 g_1 &= 1 \\ \overline{f_2}\ \overline{g_2} + f_2 g_2 &= 1 \\ &\dots \\ \overline{f_k}\ \overline{g_k} + f_k g_k &= 1 \end{aligned}$$

This set of equations is equivalent to the equation

$$(\overline{f_1}\ \overline{g_1} + f_1 g_1)(\overline{f_2}\ \overline{g_2} + f_2 f_2)...(\overline{f_k}\ \overline{g_k} + f_k g_k) = 1$$

The above equation can be solved similarly as in problem **??**.

a) There exist four solutions
$x_1^1 = \overline{b}, \quad x_2^1 = b.$
$x_1^2 = \overline{b} + a, \quad x_2^2 = b.$
$x_1^3 = \overline{b} + \overline{b}, \quad x_2^3 = b.$
$x_1^4 = 1, \quad x_2^4 = b.$

b) $x_1 = a + b, \quad x_2 = a\overline{b}, \quad x_3 = b.$

c) $x_1^1 = \overline{a} + b + c, \quad x_2^1 = a + c, \quad x_3^1 = a + b.$
$x_1^2 = b + c, \quad x_2^2 = a + c + \overline{b}, \quad x_3^2 = a + b.$
$x_1^3 = b + c, \quad x_2^3 = a + c, \quad x_3^3 = a + b + \overline{c}.$
$x_1^4 = b + c, \quad x_2^4 = a + c, \quad x_3^4 = a + b.$

d) $x_1 = \overline{a} + bc, \qquad x_2 = b\overline{c} + \overline{a}c;$

e) Three solutions with additional condition $\overline{a}b = 0$. Without satisfying this condition the solutions do not exist.
$x_1^1 = a, \quad x_2^1 = a,$
$x_1^2 = a + \overline{c}, \quad x_2^2 = a,$
$x_1^3 = a, \quad x_2^3 = a + \overline{c}.$

Chapter 2

Digital Devices

2.1 Multiple Outputs

So far in this text, we have mainly dealt with SSI devices and functions with only one output. Most of the remaining studies in this course, however, will be concerned with MSI and LSI technology: multiple output devices.

Recall that we defined SSI (small-scale Integration) as integrated circuit packages containing up to 12 gates each, while the MSI (medium-scale ICs) technology had up to 100 and LSI (large-scale ICs) accommodated up to 1,000 gates per package. The TTL devices illustrated on the previous pages are examples of SSI technology, of which the AND and OR gates are the most significant. As we were able to demonstrate, however, the NAND and NOR gates are very prominent in the design of digital circuits: the NAND gate is said to be the universal gate. This is because any digital circuit can be implemented with it. Figure **??** reinforces this statement and shows how the inverter, AND gate, and OR gate can be implemented using only NAND gates.

Figure 2.1: (NAND gates)

MSI devices perform specific digital functions commonly employed in the design of digital circuits and are, as a rule, available commercially in IC packages. By selecting an MSI device already available, the designer not only saves time but also reduces the number of IC packages, as well as interconnecting wires, needed in the project. Adders, comparators, decoders, encoders, etc. are only some examples of such devices that perform the functions of adding, comparing, coding, decoding, etc. Of course, should an MSI device not be available to perform the function needed, a good designer should be able to synthesize the circuit from the gates on hand.

We have already had an opportunity to design a device with multiple outputs: the magnitude comparator. Recall that this device had to find solutions such as: $A = B$, $A < B$, $A > B$. The SN54L85, 4-bit magnitude comparator, used for this example is an MSI device with 33 gates. Had we wanted to realize this function using SSI technology, possibly 4-5 gate packages with multiple wire connections would have been needed.

Even an adder is a multiple output device. When two numbers are added together, a sum such as $1 + 1 = 2$ (or binary 10) must result. One output would be for the sum but we also need a "carry function" for such operations. $9 + 1 = 10$: nine plus one equals zero, carry one. A second output, therefore, is provided for: the "carry" function.

To introduce the procedure of designing multiple output devices, we will use the following "overly simplified" example:

We will assume that we need to design a function that will control six sets of flood lights on a football field. As shown in Table **??** below, there is one set of lights at each end of the field with two sets on each of the sides. We label these sets as a, b, c, d, e, f. See Figure **??**.

Figure 2.2: Sets of lights for the "overly simplified" example

Therefore, we need to design six different functions (or six different outputs of a function with each output behaving as a separate function). If we wanted to light up the east side of the field, b and c would need to be switched on. a, b and e, d light the opposite corners, a, b, c, d makes a half moon around the field and it takes a, b, c, d, e, f to turn on all of the lights. We can, therefore, construct the following table.

	Lights on on						
	a	b	c	d	e	f	comment
ALL	1	1	1	1	1	1	(all lights on, all = 1)
EAST	0	1	1	0	0	0	(0 = not the east side)
CORNERS	1	1	0	1	1	0	(a, b, d, e are corners)
HALF MOON	1	1	1	1	0	0	(e and f are not on)

Table 2.1: Table for lights

Given two inputs X and Y, we can construct a truth table and create a function and a map for each output.

The outputs a and d are the same above. They are both "on" in all cases except in the case "east". It is, therefore, easier to choose the "maxterm", POS form.

X	Y	output for a	comment
0	0	1	
0	1	0	(when $\overline{A}B$, $a = 0$) Therefore, $\overline{a} = \overline{A}B$
1	0	1	and, by De Morgan's Laws, $a = A + \overline{B}$.
1	1	1	

Table 2.2: Truth table

The output d is identical to output a: $d = A + \overline{B}$.

Similarly, all of the outputs can be placed in the truth table at the same time. The remaining outputs are:

Truth table		outputs				
X	Y	b	c	d	e	f
0	0	1	1	1	1	1
0	1	1	1	0	0	0
1	0	1	0	1	1	0
1	1	1	1	1	0	0

The output b must be equal to "1" at all times.

$$b = (\overline{X}\overline{Y}) + (\overline{X}Y) + (X\overline{Y}) + (XY) = \overline{X}(\overline{Y} + Y) + X(\overline{Y} + Y) = \overline{X}(1) + X(1)$$

We see that $b = X + \overline{X} = 1$.

As with outputs a and d, we realize output c in the POS form. When $X\overline{Y}$, $c = 0$. Therefore, $\overline{c} = X\overline{Y}$ and by De Morgan's Laws, $c = \overline{X} + Y$.

The outputs e and f" can be taken from the maps below.

Number of inputs

The number of input variables used in the the truth table or the map is dependent on the number of combination possibilities desired. In the above example, only four solutions were desired: Light the east, the corners, the half moon, or all of the lights. These four possibilities (with two states possible, ON or OFF), can be realized with two inputs since 2 squared equals 4. There are a total of six individual lights, however. Had we wanted to be able to turn each one on separately, two variables would not have been enough. The next step would have been three inputs since 2 raised to the power of 3 = 8. In a function requiring from 9 to 16 combination possibilities, 4 variables would be called for, etc.

Determining the output from the map

When the function is determined by using the Karnaugh map, each output requires a separate map. In the above case, with six outputs, six maps are required. Since four results are desired, all of the maps are of the two variable size. The map coordinates are provided by the inputs. A 1 is placed in the coordinate if there is a 1 listed in the corresponding place in the output column. If a 0 is in that spot, a 0 is placed in the appropriate rectangle. All other values not given are (-), don't cares. Observe the output c, for example. We place a 1 in the map at the coordinates 00, 01, and 11 but there is a 0 in the c column at coordinate 10.

When we read the function from the map, we can choose to read it in SOP or in POS form. It would be more convenient to read the map c below in the POS form since there is only one 0 at coordinate 10. This produces $\overline{c} = X\overline{Y}$, which by De Morgan's Laws is equivalent to $c = \overline{X} + Y$. On the other hand, the f output map is perfect for the SOP form. There is only one 1 at the coordinate 00: $f = \overline{X}\ \overline{Y}$. We would like to have OR gates at all the outputs, however, and, therefore, change output f, according to De Morgan's Laws, to $\overline{X + Y}$.

The corresponding circuit is shown below in Figure **??** the maps for each output.

Figure 2.3: (maps) (circuit)

2.2 Factorization

2.3 Adders

To facilitate getting use to the design of multiple output circuits, we will concentrate on simplier devices for the moment and become acquainted with **Adders** and **Subtractors**.

The arithmetic operations performed directly by a computer are simply addition and subtraction. All of the other operations, such as multiplication, division, etc. can be carried out by repeated addition or subtraction. In fact, as we will see later on, subtraction is also simply addition of the complement.

2.4 Binary Addition

The base 10 number system contains the digits 0, 1, 2, 3, 4,....9. When one is added to eight, the result is nine. Nine, however, is the highest digit. Adding 1 to 9 produces 0 in that position and the next position to the left is increased by 1. Therefore, $1 + 9 = 10$. In the base two system, however, the highest digit is ”1”. Therefore, adding 1 to 1 in that position will also produce ”0” and the next position to the left is increased by 1. This is the ”carry” that you are familiar with. Therefore, in binary addition, $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, but $1 + 1 = 0$ and carry 1 to the next position so that $1 + 1 = 10$; which equals decimal 2. Note the following examples:

If we designate the ”augend” as input A and the ”addend” as input B, we can construct a truth table to obtain the ”sum” as the output; which we will denote by S.

```
carries:     1  1         111      1 1111
augend:       1101      110101      101101
addend:    +  1001    +    111    + 100111
             ------      ------      ------
sum:         10110      111100     1010100
```

Inputs		Outputs	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

2.5 Half Adder

If we added only the least significant bit positions of A and B, the most that would result would be the "sum" of 0 with a "carry" of 1: $1 + 1 = 10$. We would then have the two inputs A and B, and the two outputs "sum" and "carry". This would constitute a "half adder" since there is no provision for a "carry" to be added into the operation, there is only a carry out. Its truth table with the two outputs appears as follows.
Constructing maps for the two outputs as shown below yields a function for the "carry" of $C = AB$.

```
Adding   A + B
         0 + 0 yields Carry = 0
         0 + 1   "       "   = 0
         1 + 0   "       "   = 0
         1 + 1   "       "   = 1
```

There are, however, several possibilities for the "sum" function: $S = A\overline{B} + \overline{A}B$ or $S = (A+B)(\overline{A} + \overline{B})$ or simply the EXOR operation $S = A \oplus B$.

```
A + B
0 + 0 yields sum = 0
0 + 1   "     "  = 1
1 + 0   "     "  = 1
1 + 1   "     "  = 0 with a carry of 1.
```

The corresponding circuits are shown with the maps.

Figure 2.4: Circuits corresponding to the maps

BLOCK Designs

Once it is understood what the device consists of, it is usually shown as a block only with its inputs and outputs. The half-adder block symbol is shown below with the inputs A and B, and the outputs S and C.

Figure 2.5: (half-adder block symbol)

2.6 Full Adder

In reality, an adder needs more than two inputs because adding two digits together is not enough to accomplish the actual operation. This usually consists of at least three inputs because a `"carry"` needs to be provided along with the two bits to be added. This increases the inputs to A, B, and C (for the carry-in operation). The outputs, however, remain the same: `sum` and `carry-out`. This is the full adder and its truth table will consist of three input variables and two outputs as shown below.

Figure 2.6: (truth table)

The ”sum” and the ”carry” maps for the full adder provide us with the outputs:

$$S = (\overline{a}\overline{b}c) + (\overline{a}b\overline{c}) + (abc) + (a\overline{b}\,\overline{c})$$

and

$$C = (ab) + (bc) + (ac) \; .$$

Figure 2.7: (maps) (sum and carry circuit)

The full-adder can also be constructed from two half-adders as the circuit below shows.

Figure 2.8: (two half-adder circuits)

In this case, the outputs would be

$$S = A \oplus B \oplus C$$

and

$$C = AB + C(A \oplus B)$$

2.7 Binary Parallel Adder

In the above full-adder, only one pair of bits were added and a "carry-out" was generated. To add two 2-bit numbers, two such circuits, of course, would be needed and adding two 4-bit numbers calls for four full-adders. In the 2-bit example below, the first full-adder has no "carry-in". It adds 1+1 for a sum = 0 and an "carry-out" of 1 is generated which serves as the "carry-in" for the next full-adder.

```
Full-adder:       #2   #1
carry-in:     1    1    0
augend:            1    1
addend:            1    1
                   ------
sum:               1    0
carry-out:         1    1
```

Figure 2.9: (two full-adder block designs)

The 2-bit binary full-adder SN5482 made by Texas Instruments is shown below for your information.

Figure 2.10: (SN5482)

2.8 Carry Propagation

Recall in our discussions on hazards, we mentioned that propagation time was critical. Outputs will not be correct unless the signals have enough time to propagate through all the gates from input to output. Adding two 4-bit numbers would require four full-adders in parallel, which means that carry #5 would have 2 x 4 = 8 gate levels from carry #1 to carry #5 through which to propagate. If we assumed, for example, 10 nano seconds propagation time for each level, it would take carry #5 eighty ns to propagate through all eight levels from input to output. By that time, different augends and addends could be at the inputs causing an incorrect sum. This delayed carry propagation time is reduced by using a so called "look-ahead carry" such as the Texas Instrument's SN54182 as shown below.

Figure 2.11: (SN54182)

In a full-adder composed of two half-adders, the sum from the first half-adder becomes the carry propagation $P(i)$, as shown in the diagram below, because this term is associated with the propagation of the carry from $C(i)$ to $C(i+1)$. The carry-out of the first half-adder becomes the carry-generate $G(i)$ because it produces an output carry when both A and B are one, regardless of the input carry. The consequence is that the last carry does not have to wait for the other carries to propagate. It is propagated at the same time.

The look-ahead carry generator can be used between four half-adders with an EXOR gate at each output. The circuit shown below, however, is the Texas Instrument SN5483A, a 4-bit binary full-adder with fast carry.

Figure 2.12: (SN5483A)

The 2-bit full-adder SN5482 shown above has a maximum propagation time of 27 ns. It is 10 ns in the look-ahead carry generator but the 4-bit binary full-adder with fast carry needs only 16 ns.

2.9 Subtractors

2.10 Binary Subtraction

In the base ten number system, 9 - 7 = 2. Because the seven was contained in the nine, it could be extracted. The smaller number was subtracted from the larger number. For non-negative numbers, 7 - 9 would normally not be possible. It becomes possible, however, when 10 is borrowed from the next higher stage. Then 17 - 9 = 8.

Binary subtraction is similar. In the case of 0 - 0 = 0, 1 - 0 = 1, and 1 - 1 = 0, the subtrahends are contained in the minuends and can, therefore, be subtracted. In 0 - 1, however, the subtrahend is larger than the minuend and 10 must be borrowed from the next higher stage to yield 10 - 1. In the binary number system, however, 10 is equal to the decimal 2. Therefore, 10 - 1 = 1. When 10 is taken from a higher stage, it is also reduced by 10. Therefore, 100 - 1 = 11, which is 4 - 1 = 3 in decimal. The middle zero borrowed 2 squared = 4 from the far left position because it had zero. The middle position gave up, however, 2 raised to the first power = 2 to the far right position because a 1 in its position is worth 2.

The following examples will illustrate this point.

```
borrow:       (none)      10  10      10 10
minuend:      11011       10110       101101
subtrahend: - 1001      - 1101      -100111
              -----       -----       ------
difference:   10010        1001          110
```

Half Subtractor

By designating the "minuend" as X and the "subtrahend" as Y we can construct the truth table for the half-subtractor. There are two outputs: difference denoted by D and borrow denoted by B. After placing these values into the maps D and B, we can read the function

$$D = \overline{X}Y + X\overline{Y} \text{ and } B = \overline{X}Y.$$

Figure 2.13: Truth table and maps

The difference circuit for the half-subtractor is identical to the sum circuit for the half-adder. The borrow circuit for the half-subtractor is the same as the carry circuit for the half-adder except that one of its inputs is inverted.

2.11 Full Subtractor

Similar to the case with the full-adder, the full-subtractor must ”borrow-in” as well as a ”borrow-out”. This allows for borrowing from lower stages. Therefore, three inputs and two outputs are needed in this case. We designated the inputs as X, Y, and Z(for the borrow-in) as well as the outputs D and B as above. The truth table is constructed and the values are placed in the maps shown below.

Figure 2.14: Truth table and maps 46

Note that also here the ”difference” circuit of the full-subtractor is identical to the ”sum” circuit of the full-adder. The only difference between the circuits is between the ”carry” and the `"borrow"` functions.

The above subtractors were introduced only to give the student design practice. Normally, subtraction is performed by adders. The complement of the subtrahend is taken and then it is added to the minuend.

2.12 Magnitude Comparator

We revert to the magnitude comparator since it also has three outputs. At this time, we are able to show its truth table so that the reader will be able to see how we arrived at this design. SN54L85 is shown below for your convenience. Recall that two four bit numbers were being compared: $A0$ $B0$, $A1$ $B1$, $A2$ $B2$, $A3$ $B3$. The (=) function, was being perform by four EXNOR functions:

$$\overline{[(\overline{A}\ B)\ +\ (A\overline{B})]}$$

If we allow X to represent this function, then $(A\ =\ B) = 4$ times the EXOR function: X3X2X1X0.
and then

$$(A > B)\ =\ A3\ \overline{B3}\ +\ X3\ A2\ \overline{B2}$$
$$+\ X3\ X2\ A1\ \overline{B1}\ +\ X3\ X2\ X1\ A1\ \overline{B1}$$

and

$$(A < B)\ =\ \overline{A3}\ B3\ +\ X3\ \overline{A2}\ B2$$
$$+\ X3\ X2\ \overline{A1}\ B1\ +\ X3\ X2\ X1\ \overline{A0}\ B0$$

If we allow X to equal the EXNOR we do not have to write:

$$(A < B)\ =\ (A3\ B3)\ +\ (A\ \odot\ B)\,(A2\ B2)\ +\ (A\ \odot\ B)\,(A\ \odot\ B)\,(A1\ B1)\ +$$
$$(A\ \odot\ B)(A\ \odot\ B)(A\ \odot\ B)(A0\ B0)$$

The EXNOR map as well as the $A\ \overline{B}$ and $\overline{A}B$ maps are given for reference.

Figure 2.15: maps 47

The truth table below shows the logic of the comparison. If, for example, sub-bit $A(3) = 1$ and the corresponding sub-bit $B(3) = 0$, then, of course, the number $A > B$ number. When the bit $A = B$ bit in the far left position(3), and bit A in the position to its right(2) $= 1$ while bit B in that position $= 0$, then $A > B$, etc.

A one in the far left position = 8. A one in the position to the right of it = 4. A one in the next position to the right = 2, and a one in the last position = 1.

Table 2.3: Truth table for magnitude comparator

	Bit position				output		
	#3	#2	#1	#0	$A > B$	$A < B$	$A = B$
	$A > B$				1	0	0
	$A < B$				0	1	0
	$A = B$	$A > B$			1	0	0
	$A = B$	$A < B$			0	1	0
	$A = B$	$A = B$	$A > B$		1	0	0
	$A = B$	$A = B$	$A < B$		0	1	0
	$A = B$	$A = B$	$A = B$	$A > B$	1	0	1
	$A = B$	$A = B$	$A = B$	$A < B$	0	1	0
	$A = B$	$A = B$	$A = B$	$A = B$	0	0	1

Note that the above truth table is in descending order. In the highest position $A > B$, then $A < B$, and finally, $A = B$. Once $A = B$ in the higher position, it remains equal until all bits are equal.

2.13 Codes

The various digital systems use different codes but the output of one system may serve as the input of another. Terminal outputs, for example, are used as computer inputs. Computer outputs, conversely, are also used as inputs to printers. It is, therefore, often necessary to appropriately convert the data being transmitted by a conversion circuit to meet the requirements of the system receiving it. One binary code is then converted to another. The inputs supply the bit combinations that the original device calls for and the corresponding bit combinations of the other code are generated. A combinational circuit performs this function through logic gates.

2.14 BCD Adder

The NBCD (natural binary-coded decimal) is the code which represents decimal digits by their binary equivalents. This code is also called the 8421 BCD or simply BCD code, which is similar to the binary number system used previously in this text. In BCD, however, four binary digits represent one decimal position. This means that the decimal number 683, for example, would be represented in BCD by three separate bites. The first bite on the right represents the 0-position, etc.

1. The 3 as LSD would be represented by the bite 0011 in BCD,

2. the 8 would be represented by the bite 1000 in BCD,

3. and the 6 would be represented by the bite 0110 in BCD.

Therefore, the decimal number 683 = 0110 1000 0011 in BCD code.

The binary sum is similar but not identical to the BCD sum. The decimal number 683 = 0010 1010 1011 in binary. There are three decimal positions in the number 683 and only one of the digits 0, 1, 2, 3, ...9 can be used in each position. In the binary sum equivalent, the first bite in the LSD position, 1011, equals 11. The BCD code, however, cannot express more than the decimal 9 per bite, or decimal position. The equivalents 10 through 15 (1010 through 1111) in binary are, therefore, not used in the BCD sum. In BCD, when the bite = decimal 10, the one is carried to the next bite to the left and the first bite = 0. Therefore, decimal 10 = the BCD sum of 0001 0000. The one in the second bite was the carry. In addition, this means

```
carry: 1
          1001  binary augend
        + 0001  binary addend
          ----
    0001  0000  BCD sum
```

Note that when no output carry is generated, the BCD sum (0-9) is identical to the binary sum.

To convert a binary sum to a BCD sum, we add 0110 to the binary sum. In the circuit shown below, we use one 4-bit binary adder with the augend A and the addend B as inputs. A second 4-bit adder is then placed at the output to accept the four outputs of the first adder. The four outputs from the first adder are four of the eight inputs into the second adder. The output adder, however, also has two inputs from the output carry as well as two "zero" inputs: a total of eight inputs. The consequence is that when there is a one in the second and fourth, or in the third and fourth, or in the second, third, and fourth digit, or in the carry out of the binary sum (either 1010 or 1100 or 1110 or carry out), an output carry as input into two of the addend inputs of the second 4-bit adder will be generated. The other two addend inputs are set constant at 0. This provides for the addend of 0110. With a carry output of 0, the addend in the second 4-bit adder will be 0000 and, therefore, the second bite has no value: nothing will be added to the augend, there will be no change. With a carry output of 1, however, the addend in the 4-bit output adder will have a value of 0110.

The addition of 6 + 3 = 9, 9 + 1 = 10, and 8 + 4 = 12 would then be carried out as shown below.

```
1st 4-bit adder:

         augend:    0110          1001        1000
         addend:    0011          0001        0100
                    ----          ----        ----
            sum:    1001          1010        1100
    Output carry:     0             1           1

2nd 4-bit adder:
```

```
augend:    1001         1010        1100
addend:    0000         0110        0110
           ----         ----        ----
   sum:    1001    0001 0000   0001 0010
```

Figure 2.16: (circuit)

2.15 Code Converter

The BCD adder in the example shown above illustrated how binary sums could be added and then converted into the equivalent BCD sum. There is, however, a general purpose method that can be used to design a circuit that will convert any code, containing any number of digits (inputs), to any other code, containing any number of digits (outputs): multiple inputs and multiple outputs. To introduce the Code Converter we will design a circuit to convert the BCD code into the EXCESS-3 code which are shown below. As a multioutput circuit, each output has its own map.

Figure 2.17: BCD and Excess 3 codes

Since BCD code is being converted to Excess-3 code, BCD provides four inputs and Excess-3 is represented by four outputs each of which must be put in a separate map as shown below.

Figure 2.18: Four maps

$$z = \overline{d},\ y = cd + \overline{(c + d)},\ x = \overline{b}(c + d) + b\,\overline{(c + d)},\ w = a + b(c + d)$$

Figure 2.19: Circuit

4-bit full-adder MSI circuit used as a code converter

The above code converter circuit could be designed using far less hardware by implementing it with a 4-bit full-adder MSI circuit. With the inputs A and B, the input B is set constant at 0011 while the BCD digit is applied to input A. This is possible because there is in all cases a difference of 0011 between the BCD code and the Excess-3 code.

Figure 2.20: Circuit

Other BCD codes such as 2421BCD, reflective (Gray) code, 84-2-1BCD, and Biquinary 5043210 are given for your information in the Table **??**.

Table 2.4: Table

2.16 Alphanumeric Codes

When data is transmitted in alphabetical characters and symbols such as A, B, C, $+/$, etc., a code is needed which can represent it. Such codes need at least six bits since 26 letters plus 10 digits = 36 representations. Five bits would provide a maximum of 32 combination possibilities. Six bits allow for 64 combinations. By adding the possibility to process both lower and upper case letters, 62 of these are used up. Special symbols, punctuation, carriage return, etc., however, must also be included, and this makes a seven bit code more practical with 255 combination possibilities. This is why the ASCII (American Standard Code for Information Interchange) code is the most common alphanumeric code used to represent characters. The EBCDIC (Extended BCD Interchange code) and the "internal code", which, with a few variations, is used in many computers to represent alphanumeric characters internally, are shown with the ASCII code in the table below.

Figure 2.21: Numeric codes

2.17 Parity

The ASCII code listed consists of seven bits but is, for all practical purposes an 8-bit code because an eigth bit is invariably added for parity. This is an additional bit that checks to determine whether the data has been transmitted correctly. As you will have noticed from the beginning of this text, EXOR gates are of great significance. The EXOR or EXNOR gates are also very usefull in systems that require **"error-detection"** such as in the case of the parity check. The Texas Instrument 9-bit odd/even parity generator/checker is shown below.

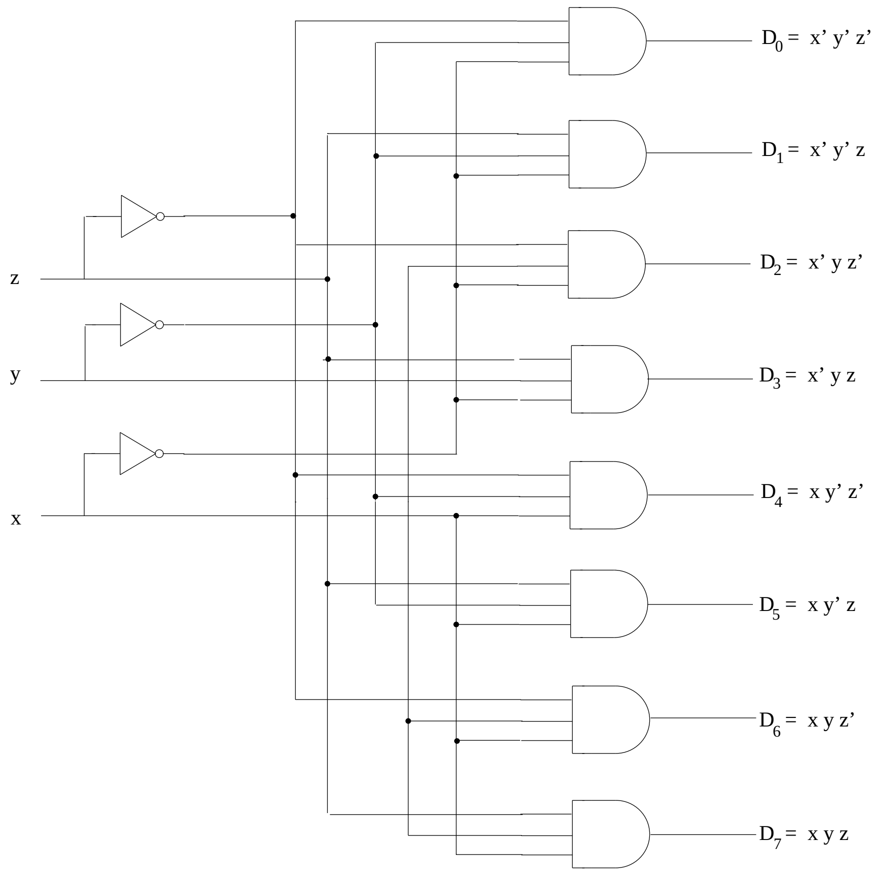


Figure 2.22: Fig3.22.xfig.ps = SN54180

At the output of the SN54180, the message received is checked with the message transmitted. An error is detected if the checked parity does not correspond to the one transmitted. The circuit generating the parity in the transmitter = the **parity generator.** The circuit checking the parity in the receiver = the **parity checker. Even parity** is when the sum of all the bits in the word including the parity bit is even (or the number of ones in the word is even). Conversely, **odd parity** is when the sum of all the bits in the word including the parity bit is odd (or the number of ones in the word is odd).

2.18 Decoders

3-to-8 line decoder seven-segment, LED/Lamp driver
A decoder is a combinational circuit that converts binary inputs into one or more (multiple) outputs. Decoders have a wide variety of applications in digital technology. They can serve as code converters or may be used to route input data to a specified output line. This could be the case, for example, when a specified memory location in the computer is addressed. Recall at the outset of our discussion on multiple outputs, we used an example of lighting a football field using six separate sets of flood lights. This example was chosen to introduce

decoders, or rather the seven-segment and BCD-to-decimal decoders.

A watch or calculator, for example, may display the digits by illuminating separate light-emitting diodes (LEDs) or crystals. In this case, it would utilize a seven-segment display for each digit. In the football field lighting example, only six segments were used. The seventh segment cuts the circle in the middle to form an 8 as shown in Figure **??**.

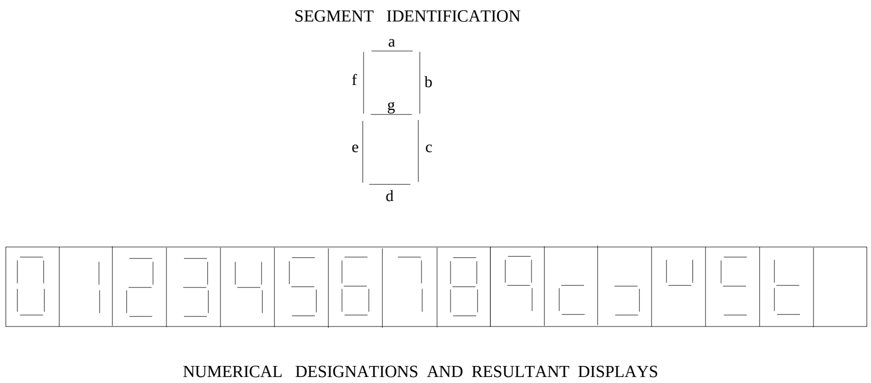


Figure 2.23: Fig3.23.xfig.ps == Seven Segments

The three inputs needed for this decoder are X, Y, and Z with the outputs a, b, c, d, e, f, and g connected to one segment each. The design procedure is the same as with any multiple output except that seven 3-variable maps are needed in this case. Recall that six 2-variable maps were used in the football field lighting example.

We, therefore, show only the block symbol of the decoder from Figure **??**, since it is understood that it contains the circuit obtained by realizing the function that the map procedure produces. The segment pattern is given as well which shows which segment needs to be lit to produce the digit desired. Only the digit 8 contains all seven segments.

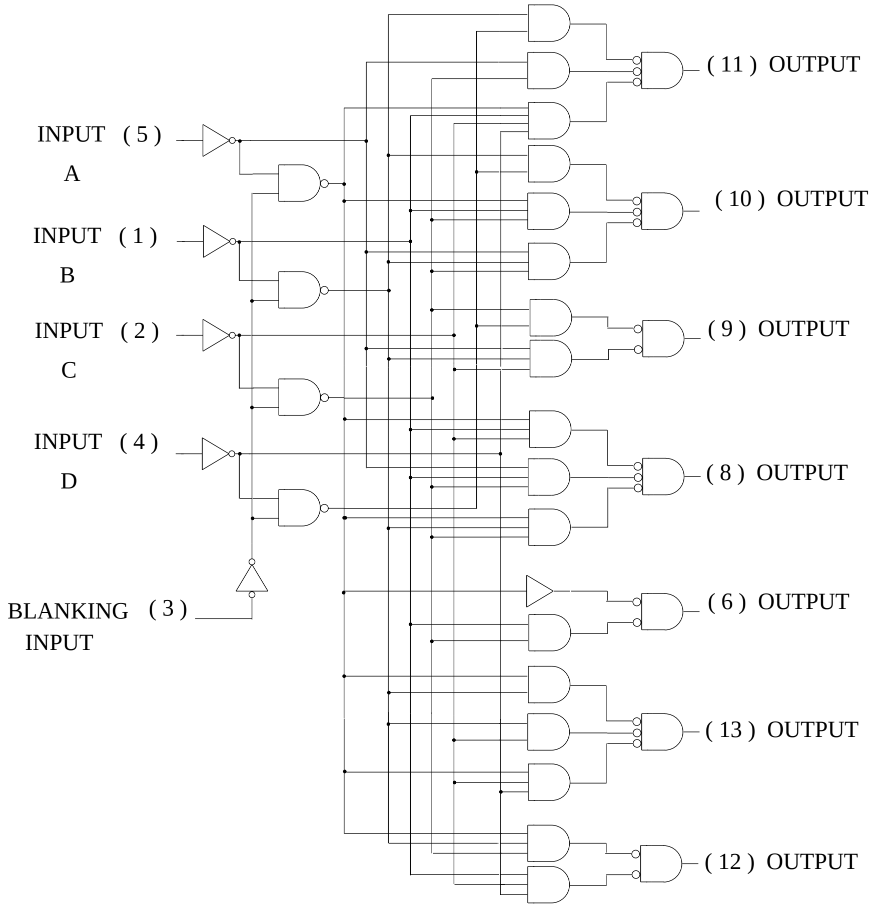


Figure 2.24: Fig3.24.xfig.ps == BCD-to-7 segment decoder

A 3-to-8 line decoder could be used to realize the BCD-to-7 function since 3 inputs can have upto 8 outputs.

BCD-to-decimal decoder

Designing a BCD-to-decimal circuit is simply an extension to the above examples. Since more than eight combinations are needed, however, four inputs would be required. A Karnaugh map procedure would call for ten 4-variable maps since there are 10 (0-9 digits) outputs. The Texas Instrument BCD-to-decimal decoder, SN54145, is shown in Figure **??** for your information. For the seven-segment design, simply deduct one input and three outputs. The football field lighting example could be implemented with two fewer inputs and four fewer outputs.

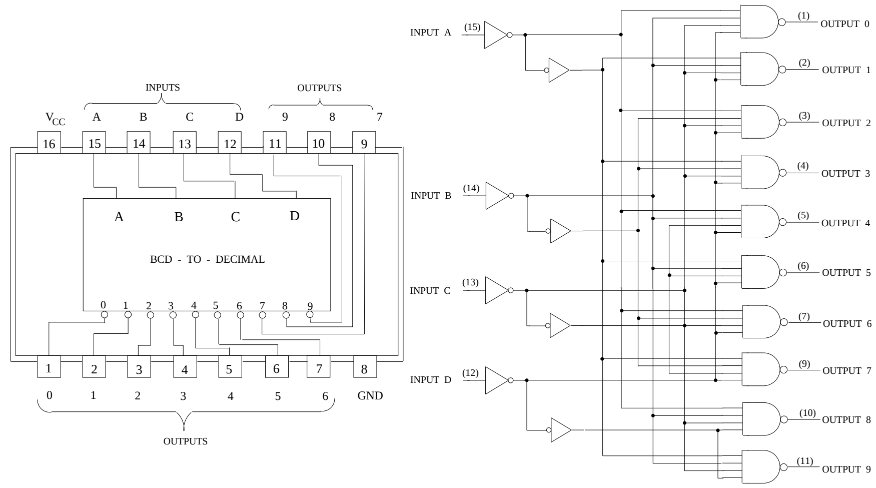


Figure 2.25: Fig3.25.xfig.ps = SN54145

Decoder as full-adder
A full-adder circuit could also be implemented using a 3 x 8 decoder and two OR gates as shown in Figure **??**.

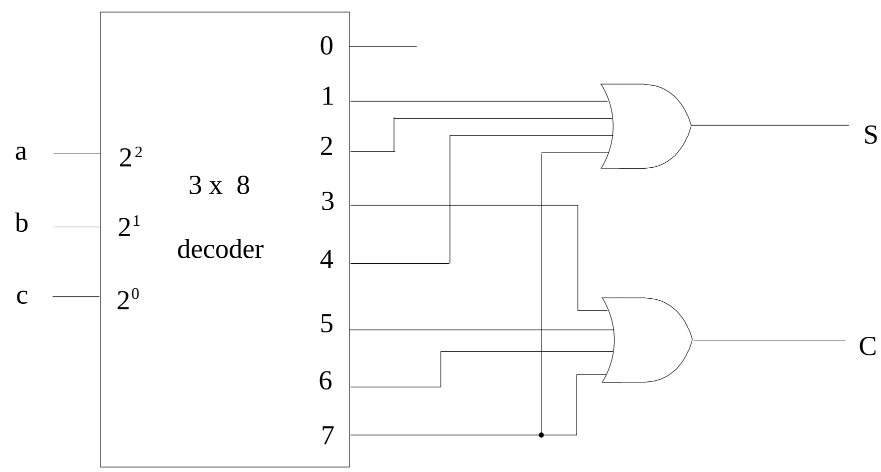


Figure 2.26: Fig3.26.xfig.ps == 3x8 decoder as a full-adder

Note that with three decoder inputs (A, B, C) the sequential outputs and their corresponding binary and decimal equivalents are as shown below:

$$\overline{a}\,\overline{b}\,\overline{c} = 000 = 0$$
$$\overline{a}\,\overline{b}\,c = 001 = 1$$
$$\overline{a}\,b\,\overline{c} = 010 = 2$$
$$\overline{a}\,b\,c = 011 = 3$$
$$a\,\overline{b}\,\overline{c} = 100 = 4$$

$$a\,\overline{b}\,c = 101 = 5$$
$$a\,b\,\overline{c} = 110 = 6$$
$$a\,b\,c\; = 111 = 7$$

2.19 Encoders

An encoder is the opposite of a decoder. It takes one or more (as a rule multiple) inputs of information and converts it to binary, which can also be one or more outputs. The following truth table shows the decimals 0 to 9 as inputs and A, B, C, and D as four binary outputs. This requires 16 combination possibilities. Notice that the decimal $9 = A$ and D, which is $A\overline{BC}D$ or binary 1001. Therefore, to convert 9 to binary, gates A and D would have to be activated at the same time. $3 = D$ and C, which is $\overline{AB}CD$ or binary 0011. $5 = D$ and B, which is $\overline{A}B\overline{C}D$ or binary 0101.

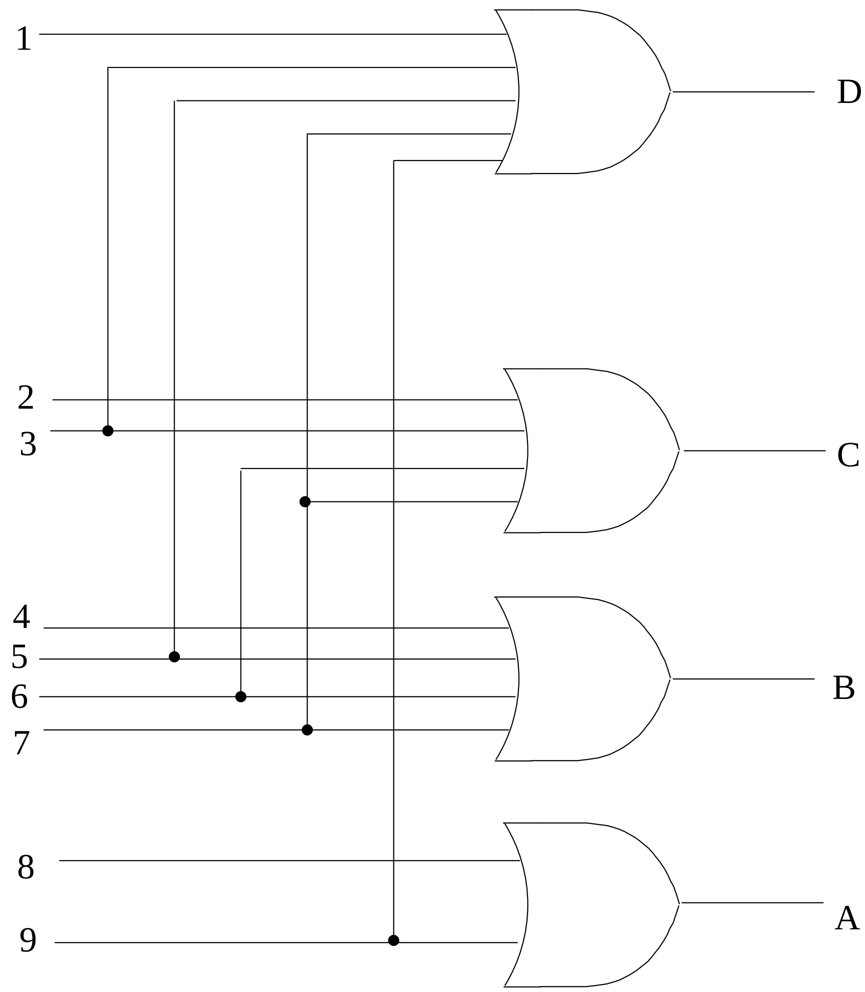


Figure 2.27: Fig3.27.xfig.ps. (Encoder for decimal to binary)

Note that the input and output values are the opposite of those given in the decoder table above. 7, for example, produces the output of $\overline{A}BCD$ or 0111.

The above encoder only has meaning if the restriction is placed on it that only one input can equal 1 at any given time. If all the combination possibilities for 10 inputs were allowed, 1,024 input combinations would be possible. This type of circuit is not available commercially but it can easily be constructed using SSI technology.

Priority encoder

Encoders can be used, for example, in a time share system to which several users have access. In such a case, priorities regarding the use can be assigned and access denied or granted by a priority encoder. The highest priority input line would then be encoded. The Texas Instrument priority encoder SN74147 is shown in Figure **??**.

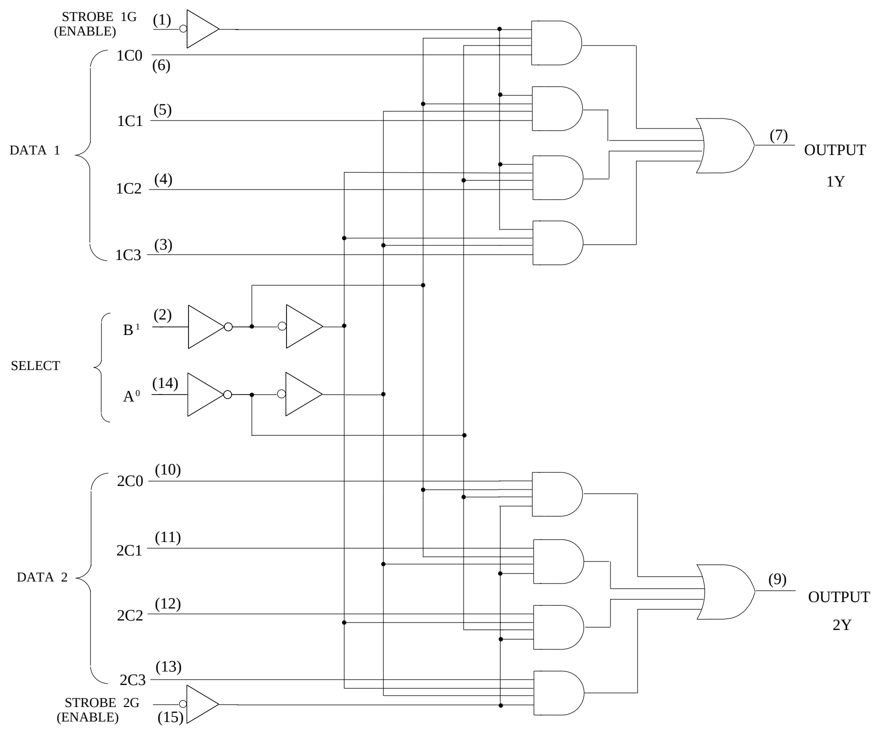


Figure 2.28: Fig3.29.xfig.ps SN74147

2.20 Multiplexers

A digital multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. In this respect, it is similar to the encoder discussed above because the encoder design has only meaning when it is restricted to accept only one of the inputs at any given time. A set of **selection lines** select a particular input line. There are 2 to the n input lines with n selection lines. An **enable** (also called strobe) is provided that will **"disable"** the output when it is in a given binary state. The circuit functions as a normal multiplexer, however, when the enable is in the opposite state: the enable state. The Texas Instrument SN54153, 4-line-to-1 multiplexer is shown in Figure ?? as a referrence.

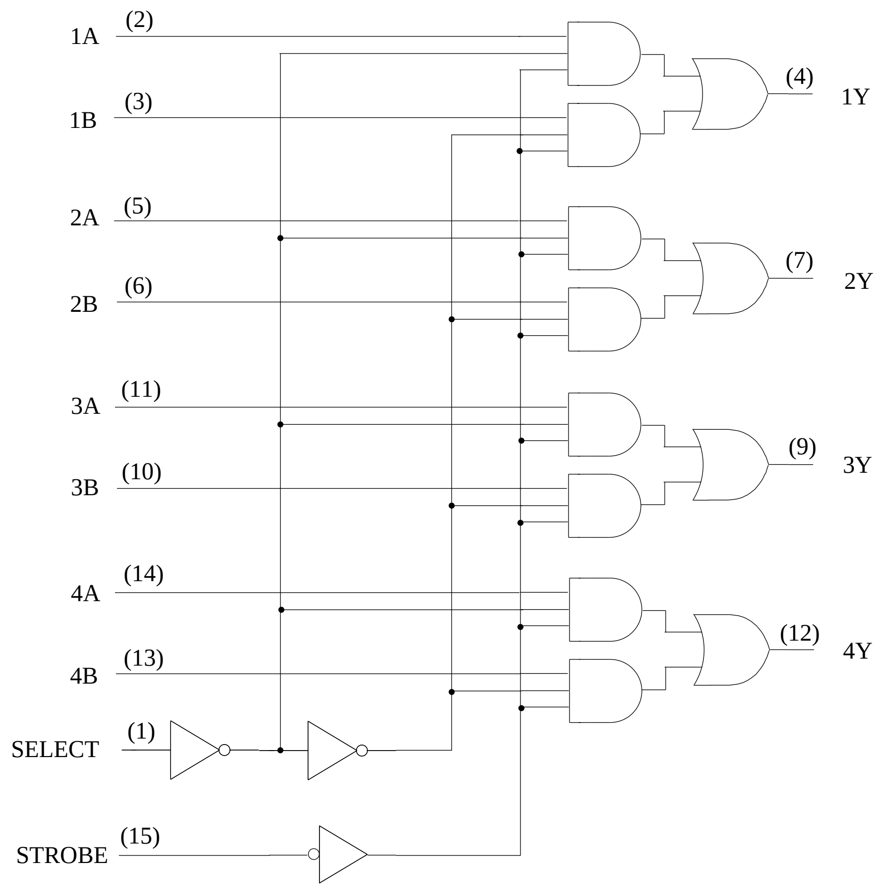


Figure 2.29: Fig3.30.xfig.ps. — SN54153

When the above multiplexer is in the "enable" state and the select lines $AB = 01$, the AND gate associated with the input $2c2$ has two of its inputs equal to 1 and the third input is connected to the input $2c2$. Therefore, the AND gate output = 1 and the selected input $2c2$ has a path to the output through the OR gate. As you can see from the SN54153 shown above, two or more multiplexers can be enclosed in one IC package. The SN54153 is a **"dual"** package. In such cases, the select lines and enable are shared by both or all the multiplexers in the package. The Texas Instrument SN54157 shown in Figure ?? is a quadruple design. The IC package contains four 2-line-to-1 line multiplexers.

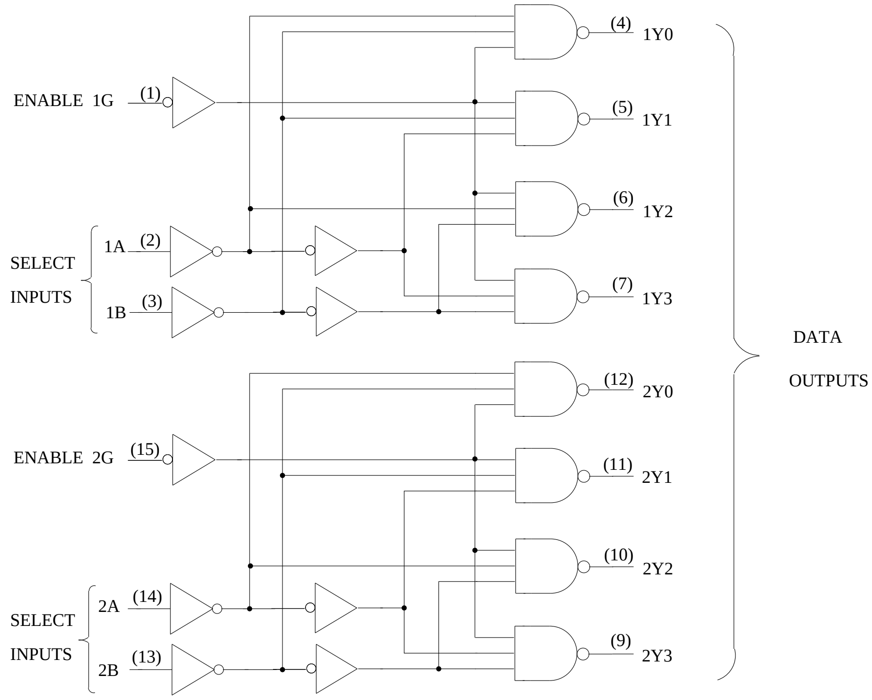


Figure 2.30: Fig3.31.xfig.ps ====SN54157

When comparing the multiplexer with the decoder, many similarities can be seen. The only difference is that outputs of the multiplexer are directed to a single OR gate to provide the 1-line output. Otherwise, it is a decoder. The multiplexer would be a multiple output device if the output OR gate were not included in the design. Multiplexers, however, are usually used to form a selected path between multiple sources and a single destination while decoders are used to decode binary information. One of the most important applications of the multiplexer, which is essentially an electronic switch that can connect one out of n inputs to the output, is as a **"data selector"**. Data selectors have numerous applications. One use, for example, could be to connect one of several terminals to the computer input.

2.21 Demultiplexers

In the previous Binary-to-decimal decoder example, we took four bits (inputs) of binary information and, with the combination possibilities given, transmited it to one of ten possible outputs. The **demultiplexer** is a similar circuit but it selects information from a single line and transmits it to one of 2 to the nth power (multiple) possible outputs. In this respect, a demultiplexer is a decoder being used for "data distribution". Recall that one of the most important applications of the multiplexer was as a "data selector" and that the multiplexer selects binary data from one of many input lines and directs it to a single output line. In a demultiplexer, the selection of the output line that is to receive the information is performed by n **select** lines. This is an n-bit input address. With 2 squared = 4 outputs, there would be $2(n)$ selection lines. Eight (2 to the 3rd) outputs require $3(n)$ select lines, etc.

Decoders and demultiplexers have essentially the same circuits. A decoder can be equipped with an **enable** input (strobe) and can be set to the values of 1 or 0. to make the circuit a demultiplexer. Such a circuit is referred to as a decoder/demultiplexer. The Texas Instrument decoder/demultiplexer SN54S139 is shown in Figure **??**. When the NAND gates are used on the outputs, the inputs must be inverted, otherwise this is the same type of circuit as the binary-to-decimal decoder. When the enable = 0 in this circuit, it operates as a normal decoder with complemented outputs. When the enable is on, = 1, however, it operates as a demultiplexer. In this case, all of the four outputs will equal one regardless of what the values of the inputs A and B are. When A and B are used as "select" lines, however, the output gate that should receive the "1" can be determined. When $A = 1$ and $B = 0$, output $2y1$ is the same as the enable input value. Decoder/demultiplexer circuits can also be connected to form larger decoder circuits as the SN54S139 shows.

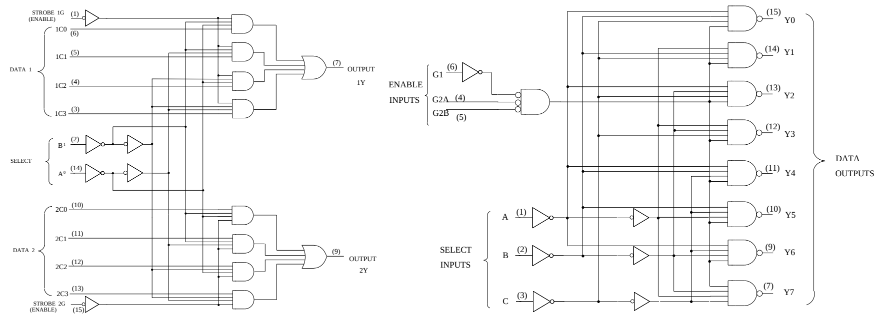


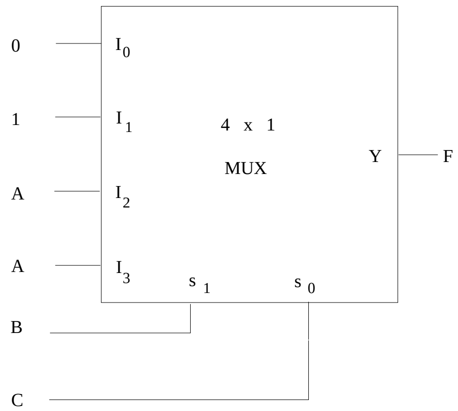
Figure 2.31: Fig3.32.xfig.ps === SN54S139

2.22 Uses of Multiplexers and Demultiplexers

As we saw from the SN54153, 4-line-to-1 line data selector/multiplexer above, a multiplexer has multiple inputs but only one output. We could picture various terminals as the inputs with the output connected to a computer input. A demultiplexer would then serve as the receiver since it has one input. with multiple selected outputs. The multiplexer can transmit several messages via its output by selecting the inputs in succession. It sends the first bit of each message, then the second bit, and the third, etc. of each input in sequence until they have all been transmited completely.

The demultiplexer at the receiving end separates the multiplex signals and channels the various bits of each message into one continuous signal in the appropriately selected output line. The select signals can be alternately 1 and 0 so that the inputs are selected in succession. The only requirement is that the signal must be long enough in duration to allow the input to reach its destination.

To help you visualize this process, we have placed the SN54153, 4-line-to-1 data selector/multiplexer beside the SN545138, decoder/demultiplexer in Figure **??**.



(a) Multiplexer Implementation

Minterm	A	B	C	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

(b) Truth table

	I_0	I_1	I_2	I_3
A'	0	1	2	3
A	4	5	6	7
	0	1	A	A'

(c) Implementation Table

Figure 2.32: Fig3.33.xfig.ps- SN54S139

The multiplexer inputs are $1c0$, $1c1$, $1c2$, $1c3$, with the select lines A and B and the output $1y$ which is the demultiplexer input at the enable input $1g$. The demultiplexer select lines A and B determine which of the outputs $1y0$, $1y1$, $1y2$, $1y3$ -which correspond to $1c0$, $1c1$, $1c2$, $1c3-$ receives the bit.

2.23 Multiplexer Implementation as a Boolean Function

Suppose we wanted to implement the function

$$f \;=\; (\overline{a}\;\overline{b}\;c) \;+\; (\overline{a}\;b\;c) \;+\; (a\;b\;\overline{c}) \;+\; (a\overline{b}\;c)$$

using a multiplexer. From the Karnaugh map, we can see that this corresponds to the minterms 1, 3, 5, and 6. Therefore,

$$f(a,b,c) \;=\; \sum\;(1,3,5,6)$$

Since the function contains three variables, we can take one of them (we choose to take a) and its complement ($\overline{a}$), and allow them to serve as two of the inputs. This leaves us with two variables to serve as select lines. Therefore, in this case, n select lines $= 2$, and 2 to the n-th power $= 2$ squared $= 4$ inputs total. Since a and $\overline{a}$ serve as two of the four inputs, we need two more which we assign the value of 0 and 1 respectively. Therefore, we can implement this function using a 4-to-1 multiplexer. The variables b and c will be connected to the select lines with b connected to the first and c connected to the second select line.

We create Table **??** as shown below in which the inputs ($I0$, $I1$, $I2$, $I3$) are the columns and the variable a in its complemented form ($\overline{a}$) is the first row while a in its normal form is the second row.

Table 2.5: Truth Table

Since there are three variables, there are eight combination possibilities. This means that the first variable, a, has four zeros listed in a truth table and then four ones following them. Therefore, the first four minterms (the zero minterms) 0, 1, 2, and 3 are placed under the four inputs in the a's row. Conversely, the last four minterms (the one minterms) 4, 5, 6, and 7 are placed under the four inputs in the a row.

Recall that in a three variable truth table, the vaule of the first variable in the first four minterms is 0 while its value in the last four = 1. The second variable value alternates, 00, 11 while the value of the third variable alternates 0 1.

The Implementation Procedure.

1. Circle the minterms of the function (1, 3, 5, 6).
2. If two minterms in a column are not circled, assign an input value of 0 to the input.
3. If two minterms in a column are circled, assign an input value of 1 to the input.
4. If the minterm in the a row is circled but the minterm in the a's row is not circled, assign an input value of a to the input.
5. If the minterm in the a' row is circled but the minterm in the a row is not circled, assign an input value of a' to the input.

The block diagram below shows the proper assignments. The circuit diagram of SN54153 4×1 MUX is also shown so that this design can be confirm. When b and c are the select lines and their values are 00, the input $I0$ is selected and the output = 0 since $I0 = 0$. When $bc = 01$, the input $I1$ is selected and the output = 1 since $I1 = 1$. However, when $bc = 10$ or 11 there are two possibilities each. When $bc = 10$ input $I2$ is selected and the output = a. Therefore, if $a = 1$, the output = 1. If $a = 0$, the output = 0. The same applies when $ab = 11$. The output = $\overline{a}$. Therefore, the output = 0 if $\overline{a} = 0$. Conversely, the output will equal 1, if $\overline{a} = 1$.

In the function $f \;=\; (\overline{a}\;\overline{c}\;\overline{b}) \;+\; (\overline{a}\;b\;c) \;+\; (a\overline{b}\;c) \;+\; (a\;b\;\overline{c})$
which is

$$f(a,b,c) \;=\; \sum\;(0,3,5,6)$$

one variable is used for all of the inputs into the multiplexer with the remaining two variables serving as select lines.

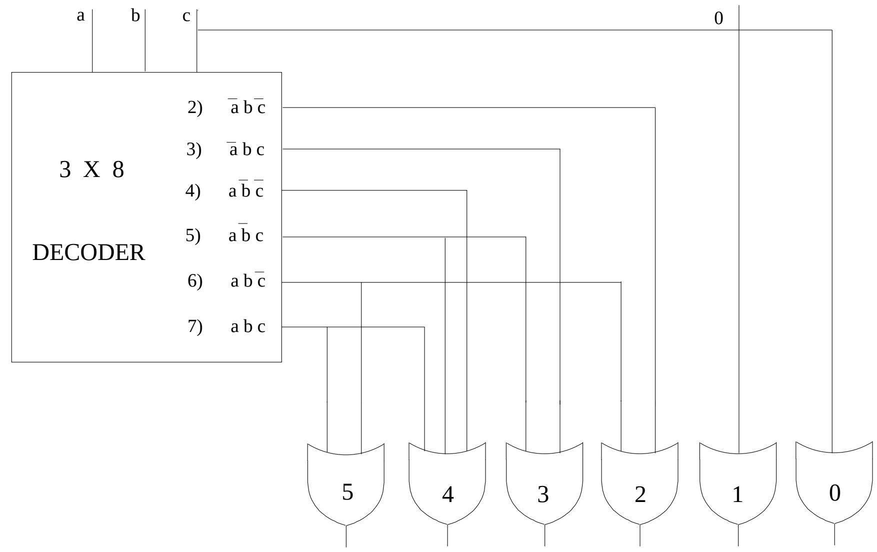
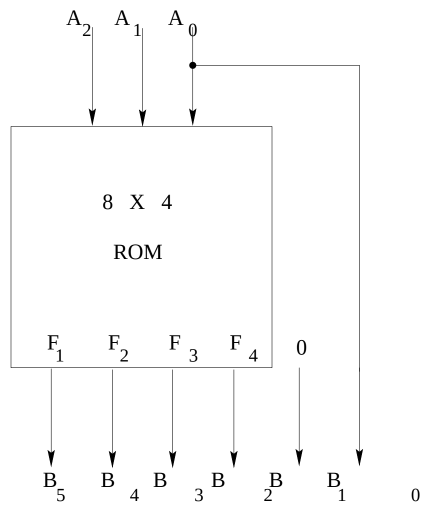


Figure 2.33: Fig3.38.xfig.ps = Multiplexer implementation table

When implementing the multiplexer as illustrated in the above example, any one of the variables could be chosen to serve as input. The table, however, would need to be set up accordingly. If we chose b as input, the minterms 0,1,4, and 5 would have to be listed in the b' row because these are the minterms in which the value of $b = 0$. Conversely, the minterms 2,3,6, and 7 would need to be listed in the b row because these are the minterms in which the value of $b = 1$. If c were chosen as input, the $\overline{c}$ row would contain the minterms 0,2,4, and 6 while the c row would contain the minterms 1,3,5, and 7.



(a) Block Diagram

A_2	A_1	A_0	F_1	F_2	F_3	F_4
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	1	1	0
1	1	0	1	0	0	1
1	1	1	1	1	0	0

(b) ROM Truth table

Figure 2.34: Fig3.39.xfig.ps == Truth table, b table, and c table

When implementing the multiplexer as illustrated in the above example, any one of the variables could be chosen to serve as input. The table,

The minterms of the function (in this case, 1,3,5, and 6) are circled and the preceeding procedure is followed to provide the values needed to implement the function.

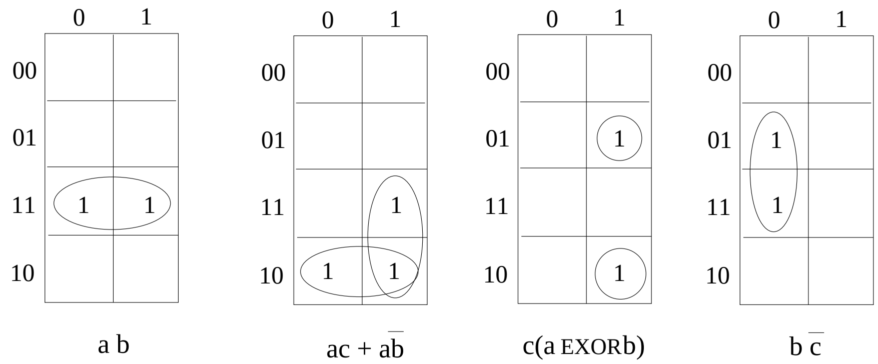


Figure 2.35: Fig3.40.xfig.ps == Multiplexer with b and multiplexer with c

When implementing the multiplexer as illustrated in the above example,

The function

$$(\overline{a}\,\overline{b}\,\overline{c}\,\overline{d}) \;+\; (\overline{a}\,b\,\overline{c}\,\overline{d}) \;+\; (a\,b\,c\,d) + (a\,\overline{b}\,\overline{c}\,\overline{d}) \;+\; (a\,\overline{b}\,\overline{c}\,d)$$

contains the minterms 0,1,4,8, and 9.
This corresponds to

$$f(a,b,c,d) \;=\; \sum\;(0,1,4,8,9)$$

With four variables, there are 16 combination possibilities. When a four variable function is implemented using a multiplexer, and one of the variables serves as input, three variables remain to serve as select lines. This means that 2 raised to the power of 3 = 8 inputs total would be required and, therefore, a 8 x 1 MUX would be needed to implement this function. Otherwise, the same rules apply that we used in the previous examples.

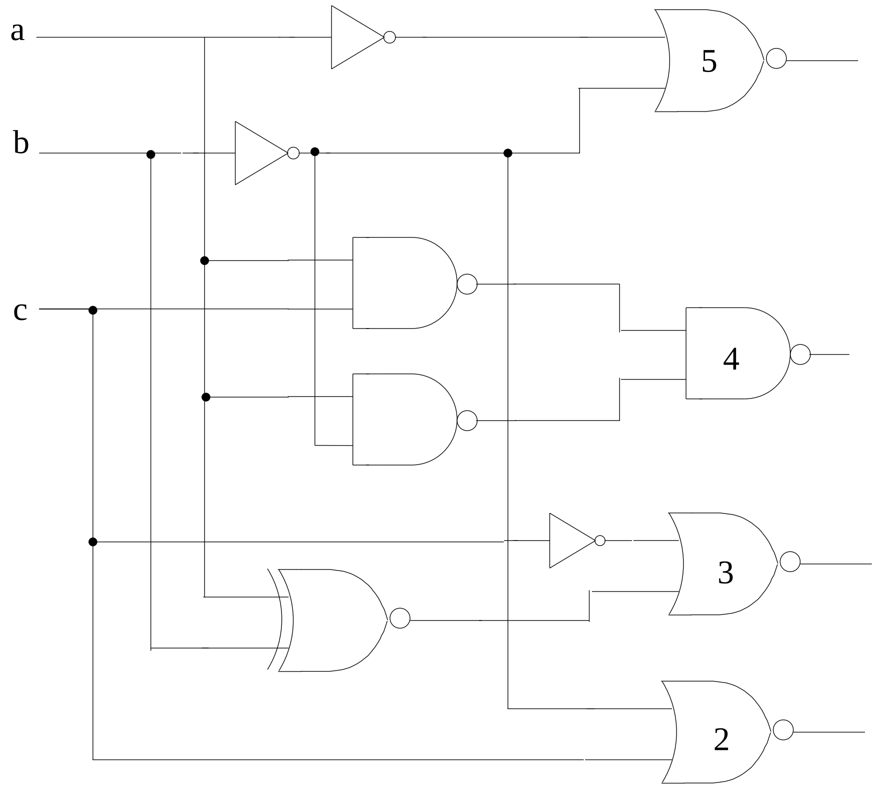


Figure 2.36: Fig3.41.xfig.ps === Karnaugh map, truth table, and 8 x 1 MUX

2.24 ROM or Read Only Memory

To introduce this device, we would like to draw your attention again to the 3 x 8 decoder. Recall that it had three inputs that produced eight outputs through eight AND gates. These eight binary combinations, 000 through 111, were the equivalents to the decimal numbers 0 through 7. In the multiplexer, we also saw how an OR gate could be used (with select lines) to direct any of these generated minterms to the output. A Read-Only-Memory (ROM) is a device that includes the decoder as well as a number of OR gates in a single IC package. The size of the decoder depends on the number of inputs that the particular combinational circuit requires to perform the desired function. The decoder AND gates and the output OR gates in a ROM are, however, connected in such a manner that a specific function is performed. According to the configuration desired, a rule is established so that a specific set of outputs will be generated for a given set of inputs. A ROM, for example, could be a circuit that contains a trigonometric table or that generates the square of the input number. The latter device will be designed in a subsequent example. We can view a ROM as a device that implements any combinational circuit when each output is considered separately as the output of a Boolean function expressed in SOP form. We could also think of it as a storage device. The contents of the ROM can be placed on the output terminal (read) but cannot normally be changed. Hence, read-only-memory.

In a ROM, all of the decoder AND gates are connected to the output OR gates through special internal links that can be fused or broken as desired. The desired interconnection for a particular application requires that certain links be fused to form the required circuit paths. Once a pattern is established, however, it is normally irreversible, permanent, and cannot be altered even when power is turned off and then on again; the exceptions will be given later on. We refer to this as programming the ROM but it is strictly a "hardware" procedure. When the manufacturer performs the programming during the fabrication of the device, this is referred to as "mask programming".

A PROM (programable ROM), however, can be ordered in which all of the connections are intact and which the buyer can break as desired. This is done by applying current pulses through the output terminals. Normally, once a pattern is set, it cannot be changed. Alterable or erasable PROMs are, however, are also available. The erasable type, EPROM (erasable PROM), returns to its initial state when it is exposed to ultraviolet light for a certain period of time, allowing the short-wave radiation to discharge the internal gates that serve as contacts. Subsequently, it can be reprogrammed. The alterable type is called EAROM (electrically alterable ROM) because electrical signals are used to erase it.

Designing with a ROM

When using a 3 x 8 decoder in a circuit, there are three inputs (ABC) and eight AND output gates. In a ROM circuit, the eight AND gates would produce 8 words, addresses (0-7), or minterms 000 - 111. By connecting an OR gate to the appropriate address numbers, we can obtain the desired function. To realize, for example,

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

we connect the output OR gate, as shown in Figure **??** below, to address numbers 2,5,6, and 7. This would be an example of a 8 x 1 ROM since there are 8 words of one bit each.

Table 2.6: Table of ROM

Input (a, b, c)		
word number	input address as vector	input addres as minterm
0	000	$\overline{a}\ \overline{b}\ c$
1	001	$\overline{a}\ \overline{b}\ c$
2	010	$\overline{a}\ b\ \overline{c}$
3	011	$\overline{a}\ b\ c$
4	100	$a\ \overline{b}\ \overline{c}$
5	101	$a\ \overline{b}\ c$
6	110	$a\ b\ c$
7	111	$a\ b\ c$

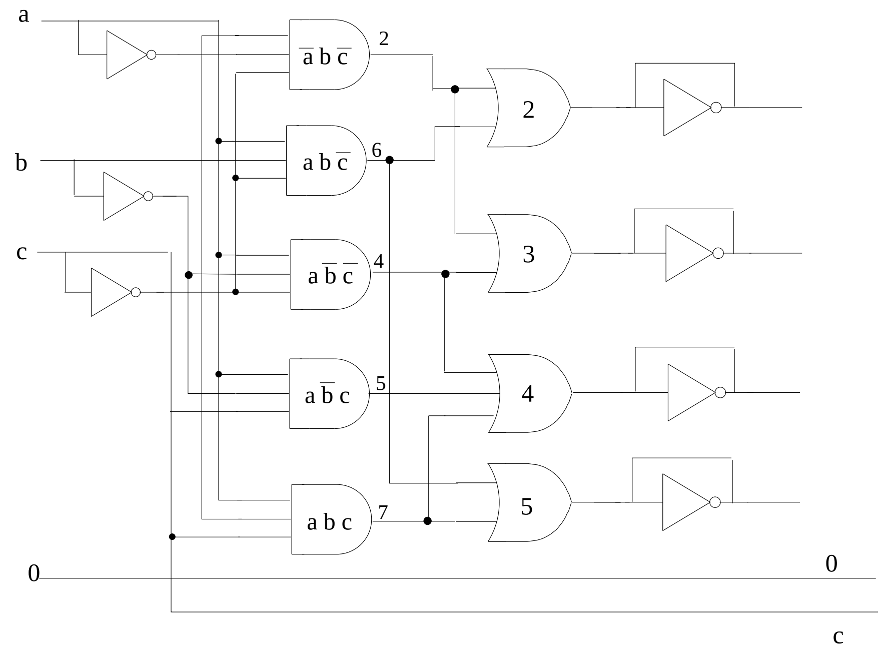


Figure 2.37: Fig3.42.xfig.ps -== Circuit

Such a simple design can, of course, be implemented using SSI technology. Then, the output would be one OR gate with four AND gates as its inputs.

On the other hand, by connecting six OR gates to the decoder outputs, we would have enough bits in the output to accommondate the square of the input address. Six binary bits equal upto 0011 1111, 63, but the highest address number in a 3 x 8 decoder is 7 and its square is 49.

If we picture the ROM output OR gates as bit positions, the following outputs would be required to express the square of the input address.

Inputs			Address	Number of output OR-gate						Square
A	B	C		5	4	3	2	1	0	
0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	1	1
0	1	0	2	0	0	0	1	0	0	4
0	1	1	3	0	0	1	0	0	1	9
1	0	0	4	0	1	0	0	0	0	16
1	0	1	5	0	1	1	0	0	1	25
1	1	0	6	1	0	0	1	0	0	36
1	1	1	7	1	1	0	0	0	1	49

Therefore, to obtain the square of the input address, connect the following input addresses to the corresponding OR-gates. When a "1" appears in the address row, this word is connected to the OR-gate number of the column.

Connect address number to OR-gate number

0	none
1	0
2	2
3	3 and 0
4	4
5	0,3,4
6	5 and 2
7	4,5,0

The connections to the gates not specified (those positions that are denoted by 0 in the table) are broken in order to obtain 0 at those bit-positions.

Figure 2.38: Circuit with 3 x 8 decoder and 6 OR gates

It is essential to set up such a truth table to establish the connections to be made. We can see, however, from the above truth table that position 1 remains constant 0 in all cases and that the values in position 0 are identical to those under input C. Therefore, if we connect position 0 direct to input C and set position 1 at a constant value of 0, we need only a 8 x 4 ROM for the remaining four outputs. The block diagram would then be as in Figure **??**.

Figure 2.39: 8 x 4 ROM implementation. Block diagram. ROM truth table

We have only used a 8 x 4 ROM based on a 3 x 8 decoder as an example. ROMs can, of course, be any size. The number of AND-gates will depend on the number of input variables and the number of words that they produce since there is one AND gate for each word or address number. The number of output OR-gates will depend on the number of bits that are required in the word. If we had four input variables, A, B, C, D, they would produce 2 to the fourth power = 16 possibilities and, therefore, 16 AND gates would be used. If the function required was also to produce the square of each number from 0 to 15, 225 would be the highest number squared. Eight output OR gates would then be needed since 225 = 1110 0001 meaning that eight bits are needed. This design, therefore, calls for a 16 x 8 ROM based on a 4 x 16 decoder.

SSI implementation

If the above function were to be implemented using SSI technology, it would be a multiple output circuit with three inputs and six outputs. The first two outputs would simply be direct connections to C and "0". The other four functions would be

$$(a\ b\ \overline{c}) \quad + \quad (a\ b\ c)$$

and

$$(a\ \overline{b}\ \overline{c}) \quad + \quad (a\ \overline{b}\ c) \quad + \quad (a\ b\ c)$$

and

$$(a\ \overline{b}\ c) \quad + \quad (\overline{a}\ b\ c)$$

and

$$(\overline{a}\ b\ \overline{c}) \quad + \quad (a\ b\ \overline{c})$$

When we put these four separate functions into maps as shown in Figure **??**, however, we see that they are reduced to (ab) and $(ac)\ +\ (a\overline{b})$ and $C(a\ \oplus\ b)$ and $(b\overline{c})$

Figure 2.40: Maps

In the circuit diagram shown in Figure **??**, however, we have used NOR gate implementation on circuits 5, 3, and 2 while a NAND gate implementation was appropriate for circuit 4.

Figure 2.41: circuit

2.25 PLA or Programmable Logic Array

The PLA is similar to the ROM in concept but it does not provide full decoding of the variables and does not generate all of the minterms as the ROM does. The PLA is used in cases in which a number of don't-care conditions is excessive. Its size is determined by the number of inputs, the number of product terms (AND-gates), and the number of outputs required (OR-gates). It is superior to a ROM for circuits that have a large number of don't-care conditions, which obviously may not need to be, and perhaps should not be, generated. If a device were required, for example, to convert a 12-bit code into a 6-bit code, 4096 AND-gates would be needed because 2 raised to the power of 12 = 4096 product terms. This would call for a 4096 x 6 ROM since 6 OR-gates would produce the 6 bits needed in the output for the 6-bit code. Only 47 words would, however, be used since there would be only 47 valid entries. The remaining 4049 combinations would be don't-care conditions, representing a waste. A typical, commercially supplied PLA has 16 inputs, 48 product terms, and 8 outputs. A mask-programmable PLA can be ordered from the manufacturer but field programmable logic arrays (FPLAs) are also available and can be programmed by the user. The links between the inputs and products as well as the links between the products and the sums are intact in the PLA as in the ROM. The appropriate links are then broken accordingly depending on the function to be implemented.

The principle idea in the PLA is to generate and to utilize only those products that are needed for the function. To accomplish this, the outputs can share the products that are generated. A function's minterms as well as the maxterms of its complement can be considered when seeking to implement the function with the least number of products. If a maxterm is chosen to be generated by the AND-gate, it is reversed by an inverter after the output so that the proper function is retrieved.

To demonstrate the design of the PLA, we can use the previous ROM design as an example in which we obtained the square of the inputs. We will assume that, as before, position "0" is directly connected to input c and position "1" is set at constant "0". We then need only to concentrate on the remaining four outputs.

Recall that the 8 x 4 ROM in our previous example squared only the numbers from 2 to 7 since "0" squared = 0 and "1" squared = "1", which is postion "0", meaning input c. Therefore, the minterms for 2 to 7 are

$$\overline{a}\ b\ \overline{c}, \quad \overline{a}\ b\ c, \quad a\ \overline{b}\ \overline{c}, \quad a\ \overline{b}\ c, \quad ab\overline{c}, \quad abc\ .$$

Let us assume, strictly for illustration purposes, that

$$a\ \overline{b}\ c, \quad a\ \overline{b}\ \overline{c}, \quad \overline{a}\ b\ c, \quad \overline{a}\ b\ c, \quad \overline{a}\ \overline{b}\ c, \quad \overline{a}\ \overline{b}\ \overline{c}$$

are the corresponding maxterms from the complemented form of the function, which they are not.

Note that we use only the "maxterms". We do not use the product of sums (POS) form with which the maxterms are usually associated.

We need the following minterm or maxterm for the outputs indicated:

output	minterms	maxterms
2	$\overline{a}b\overline{c}$, $ab\overline{c}$,$a\overline{b}c$,	$\overline{a}\overline{b}c$
3	$\overline{a}bc$, $a\overline{b}c$, $a\overline{b}\overline{c}$,	$\overline{a}b\overline{c}$
4	$a\overline{b}\overline{c}$, $a\overline{b}c$, abc, $\overline{a}bc$,	$\overline{a}b\overline{c}$, $\overline{a}\overline{b}\overline{c}$
5	$ab\overline{c}$, abc, $\overline{a}\overline{b}c$,	$\overline{a}\overline{b}\overline{c}$

We can use one or the other, either the set of minterms or the corresponding set of maxterms, for each output in our attempt to minimize the number of products needed to be generated.

Note that the output 3 maxterms $a\ \overline{b}\ \overline{c}$ and $\overline{a}\ b\ \overline{c}$ are already covered under output 4 minterms and output 2 minterms respectively. This means that products for output 3 do not need to be generated. They already serve to cover the requirements for outputs 2 and 4 and can be taken from there to meet the needs of output 3. Since, it however, will be producing the complement of the function (maxterms were used), we must reverse the output with an inverter after the output OR gate. The design from Figure **??** illustrates the PLA construction of this function.

1	—	1	0
—	1	—	0
0	—	0	—
—	1	—	0

f 1

0	—	1	—
—	0	—	1
1	—	1	—
—	1	—	—

f 2

0	—	0	—
—	1	—	1
1	—	1	—
—	1	—	0

f 3

0	—	1	—
—	1	—	0
0	—	0	—
—	1	—	0

f 4

Figure 2.42: Fig3.43.xfig.ps == PLA circuit

Functions should always first be simlified by the map or by using some other procedure to obtain both the minterms and the maxterms. The examples that we show here, for demonstration purposes only, are, however, too simple to be considered for use in a commercially supplied PLA. These circuits could be constructed more economically using SSI technology. As already mentioned, the smallest PLA would generate 48 products with 8 outputs. Ideally, therefore, a computer program would be used to simplify the function and its complement to a minimum number of terms and then select a minimum number of "distinct" terms that cover all functions in their normal or complemented form.

Such a small circuit as in our above example, however, can be placed in a **"PLA table"** as shown below to derive the multiple output function.

Product	Inputs	Output OR-gates			
		f_2	$\overline{f_3}$	f_4	f_5
2	$\overline{a}b\overline{c}$	1	1	-	-
4	$a\overline{b}\overline{c}$	-	1	1	-
5	$a\overline{b}c$	-	-	1	-
6	$ab\overline{c}$	1	-	-	1
7	abc	-	-	1	1

Note that product #3 became redundant when the maxterms for f_3 were used in place of its minterms.

When synthesizing the circuit from the above PLA table, All of the inputs listed in the product row are connected to the appropriate AND-gate. If a literal is missing in the product term, a dash (-) is placed in that position and the connection is broken at the corresponding AND-gate input terminal.

If a term appears in an output (it belongs to the function), a 1 is placed in the appropriate row/column. Otherwise, a dash (-) is placed in that spot. If the maxterms of a function were used, it is denoted by $(\overline{f})$, as in $\overline{f_3}$ above, showing that the complement of the function has been used to satisfy the requirements. Then, in this case, the inverter after the OR-gate output is needed and its connection is left intact (as in $\overline{f_3}$). Otherwise, as with f_2, f_4, and f_5, the links on the inverter are broken.

As a challenge example of designing a PLA, we have created a theoretical function consisting of four inputs (a, b, c, d) and four outputs. Each output function, for the sake of practice, contains 8 (half of the minterms) don't-care conditions. The minterms and the maxterms are given below for each function. Those not listed are don't-care conditions.

$$
\begin{aligned}
f1(a,b,c,d) &= \sum(0,3,5,9), \quad \prod(2,6,10,12,15) \\
f2(a,b,c,d) &= \sum(3,6,9,12,15), \quad \prod(0,5) \\
f3(a,b,c,d) &= \sum(5,6,9,12,15), \quad \prod(0,3,10) \\
f4(a,b,c,d) &= \sum(3,5,9), \quad \prod(0,6,10,12,15)
\end{aligned}
$$

These functions are placed in the maps below in the appropriate manner (Fig. **??**).

Figure 2.43: 4 maps

Minimizing the functions, the minterms are grouped to form prime implicants and the maxterms are grouped to form the prime implicates as learned, which simplifies the functions to the following.

	Normal Form	Complemented Form
$f_1 =$	$\overline{ac} + \overline{b}\overline{c} + \overline{a}cd$	$ab\overline{c} + ac + c\overline{d}$
$f_2 =$	$ad + a + \overline{b}c + c$	$\overline{ac}$
$f_3 =$	$abd + a\overline{c} + \overline{c}d + bc$	$\overline{ac}\,\overline{d} + \overline{a}\,\overline{bd} + \overline{b}c$
$f_4 =$	$\overline{a}\overline{b}\overline{c} + \overline{b}\overline{c}d + \overline{c}d$	$\overline{c}\overline{d} + a\overline{bd} + a\overline{b}c + bc$

We can choose either the normal or the complemented form of the function to represent it depending on the advantages that the one or the other form offers. By inspection, we can see that the term ($\overline{b}\ c$) is contained in the normal form of f_2 as well as in the complemented form of f_3. Likewise ($\overline{c}d$) is in f_3 and f_4, and ($b\ c$) is in f_3 and f_4. The greatest advantage lies in the fact that ($\overline{a}\ \overline{c}$) appears in the normal form of f_1 while it is the only term in the complemented form of f_2. We, therefore, decide to use the normal forms from f_1, f_3, f_4, and f_5 but take the complemented form from f_2 to represent the function. By doing so, the following PLA table can be set up with the products shown.

Product	Input	Output OR-gates			
		f_1	$\overline{f_2}$	f_3	f_4
1	$\overline{a} - \overline{c} -$	1	1	-	-
2	$- \overline{b}\overline{c} -$	1	-	-	-
3	$\overline{a} - cd$	1	-	-	-
4	$ab - d$	-	-	1	-
5	$a - \overline{c} -$	-	-	1	-
6	$- - \overline{c}d$	-	-	1	1
7	$- bc -$	-	-	1	-
8	$\overline{a}\overline{b}c -$	-	-	-	1
9	$- \overline{b}\overline{c}d$	-	-	-	1

Only nine products were formed. Therefore, only nine AND gates are needed. The corresponding circuit is shown in Figure **??**. Only the links around the inverter after the output of $\overline{f2}$ are broken since the complemented form was used in this case only.

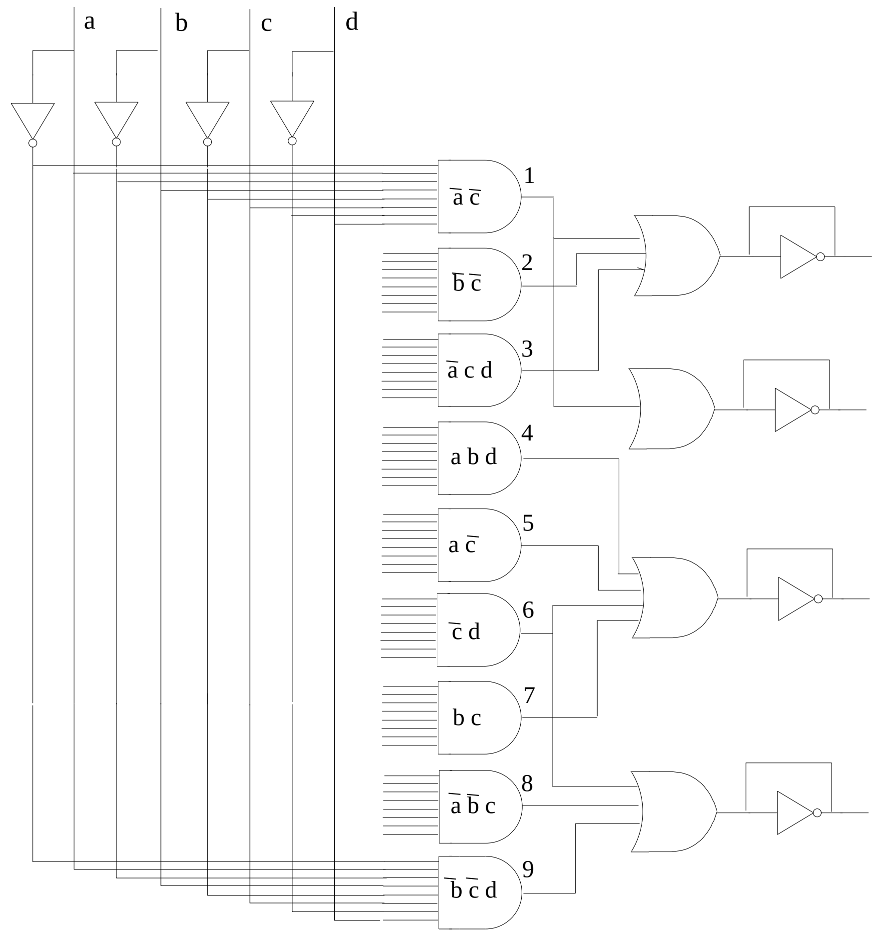


Figure 2.44: Fig3.44.xfig.ps == (PLA circuit)

8888888

Chapter 3

FIGURES 21p.1 - =

ab \ c de	000	001	011	010	110	111	101	100
00	1	0	0	-	-	0	0	-
01	0	-	1	1	0	0	1	1
11	0	1	0	1	0	0	-	0
10	0	0	1	1	1	-	0	0

ab \ c de	000	001	011	010	110	111	101	100
00	0	1	1	0	-	1	1	-
01	0	0	0	-	1	1	1	0
11	1	1	1	-	1	1	1	1
10	0	-	1	0	0	1	0	-

Figure 3.1: Fig21p.1.xfig.ps

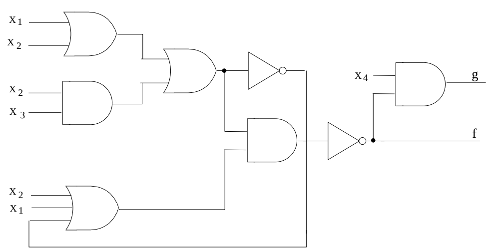


Figure 3.2: Fig21p.2.xfig.ps

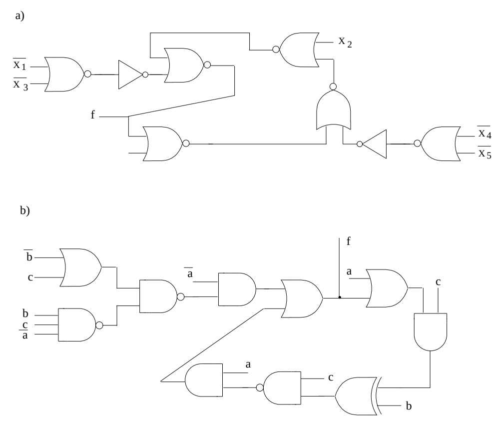


Figure 3.3: Fig21p.3.xfig.ps

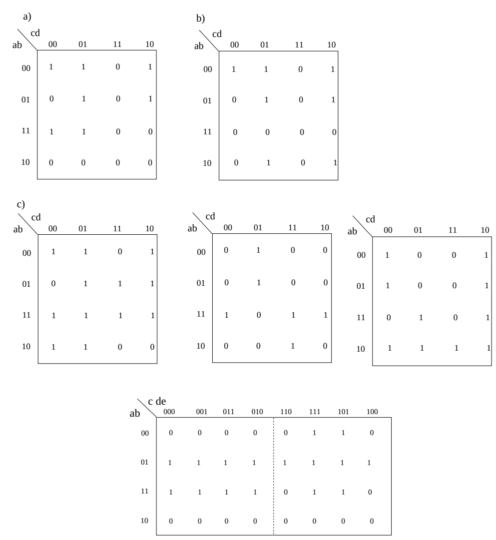


Figure 3.4: Fig21p.4.xfig.ps

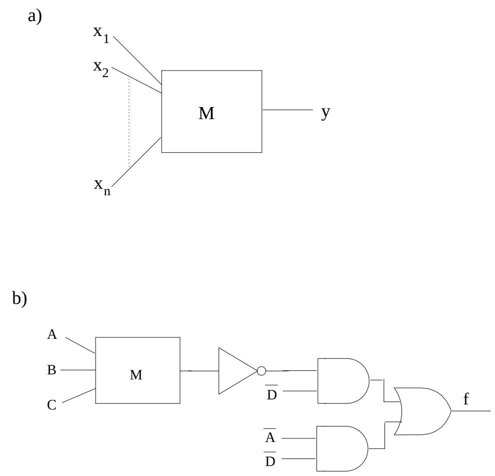


Figure 3.5: Fig21p.5.xfig.ps

Code "2 out of 5" $X_1 X_2 X_3 X_4 X_5$	Code "8421" $Y_1 Y_2 Y_3 Y_4$
1 1 0 0 0	0 0 0 0
0 0 0 1 1	0 0 0 1
0 0 1 0 1	0 0 1 0
0 0 1 1 0	0 0 1 1
0 1 0 0 1	0 1 0 0
0 1 0 1 0	0 1 0 1
0 1 1 0 0	0 1 1 0
1 0 0 0 1	0 1 1 1
1 0 0 1 0	1 0 0 0
1 0 1 0 0	1 0 0 1

Figure 3.6: Fig21p.6.xfig.ps

X_1X_2 \ X_3X_4	00	01	11	10
00	1	1	0	1
01	0	1	1	0
11	0	0	1	0
10	1	1	1	1

Figure 3.7: Fig21p.7.xfig.ps

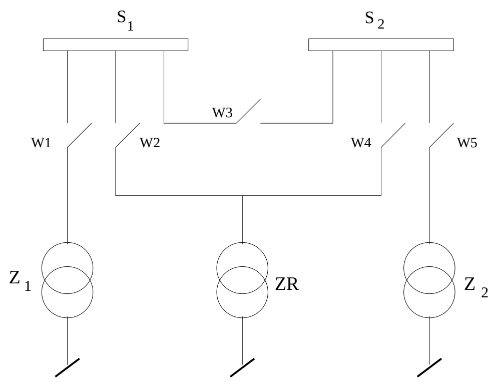


Figure 3.8: Fig21p.8.xfig.ps

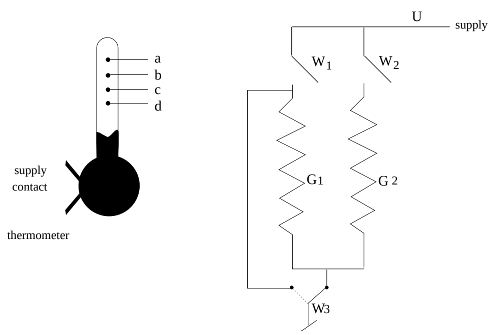


Figure 3.9: Fig21p.9.xfig.ps

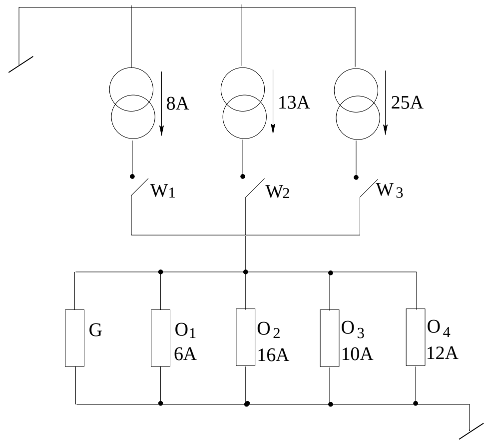


Figure 3.10: Fig21p.10.xfig.ps

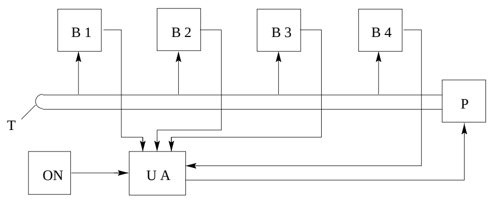


Figure 3.11: Fig21p.11.xfig.ps

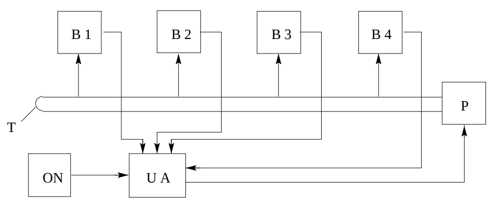


Figure 3.12: Fig21p.11.xfig.ps

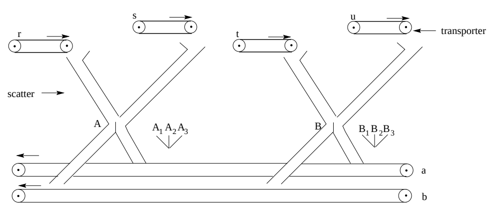


Figure 3.13: Fig21p.12.xfig.ps

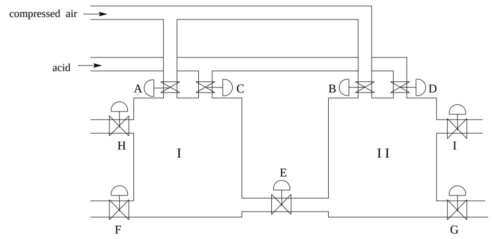


Figure 3.14: Fig21p.13.xfig.ps

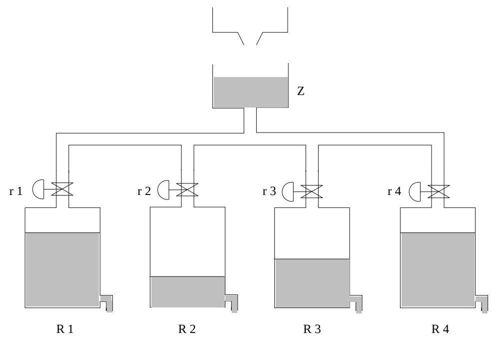


Figure 3.15: Fig21p.14.xfig.ps

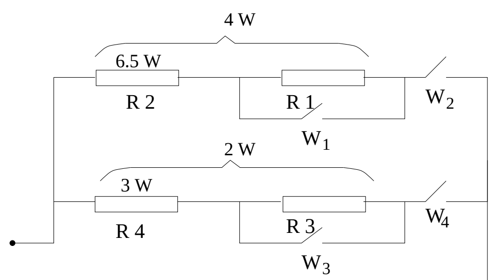


Figure 3.16: Fig21p.16.xfig.ps

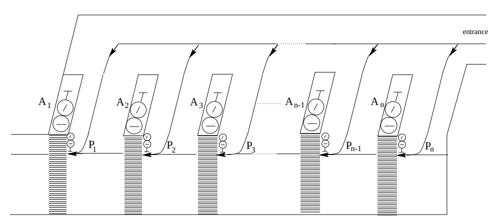


Figure 3.17: Fig21p.17.xfig.ps

abc \ de	00	01	11	10
000	1	1	0	0
001	1	1	0	0
011	1	1	0	0
010	0	0	1	1
110	1	0	1	0
111	1	1	0	0
101	0	1	0	1
100	0	1	0	1

Figure 3.18: Fig21p.18.xfig.ps

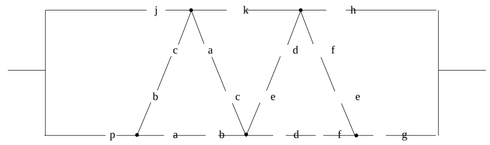


Figure 3.19: Fig21p.19.xfig.ps

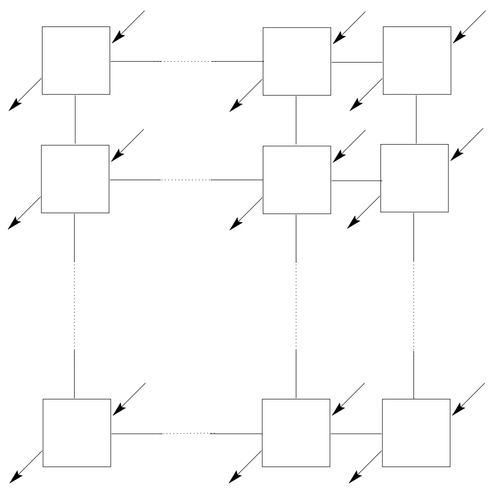


Figure 3.20: Fig21p.20.xfig.ps

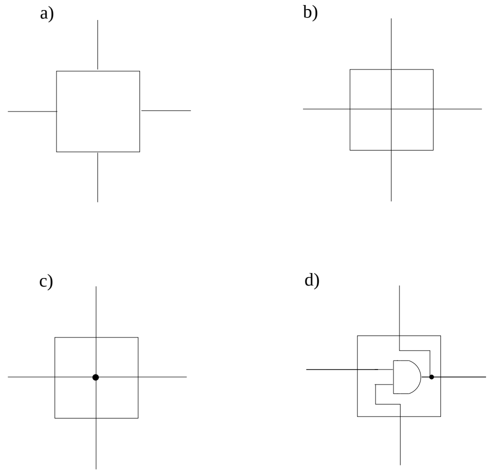


Figure 3.21: Fig21p.21.xfig.ps

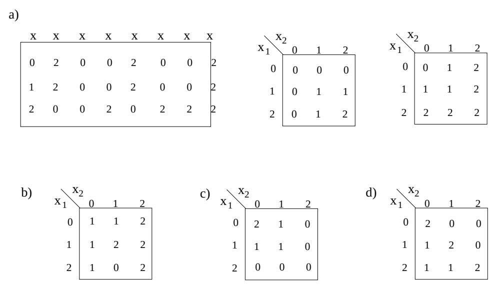


Figure 3.22: Fig21p.22.xfig.ps

ab \ cd	00	01	11	10
00	0	0	1	1
01	1	0	1	0
11	0	0	1	0
10	0	1	1	0

Figure 3.23: Fig21p.23.xfig.ps

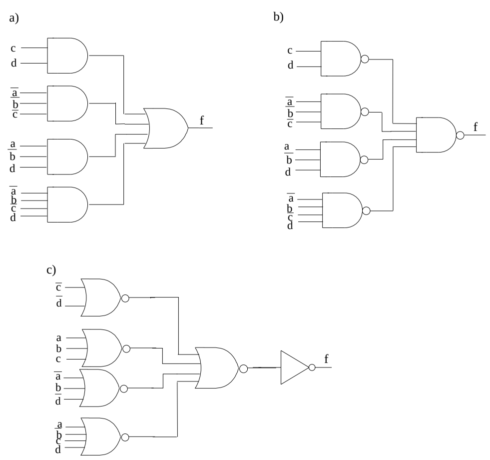


Figure 3.24: Fig21p.24.xfig.ps

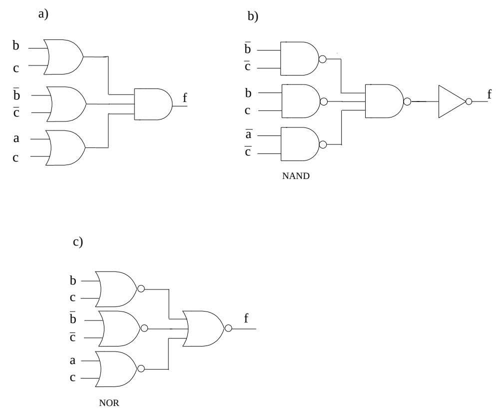


Figure 3.25: Fig21p.25.xfig.ps

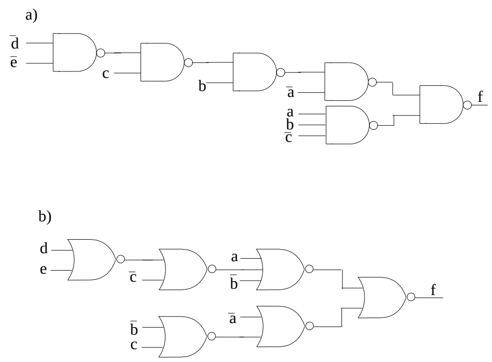


Figure 3.26: Fig21p.26.xfig.ps

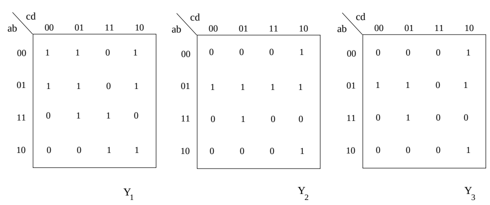


Figure 3.27: Fig21p.27.xfig.ps

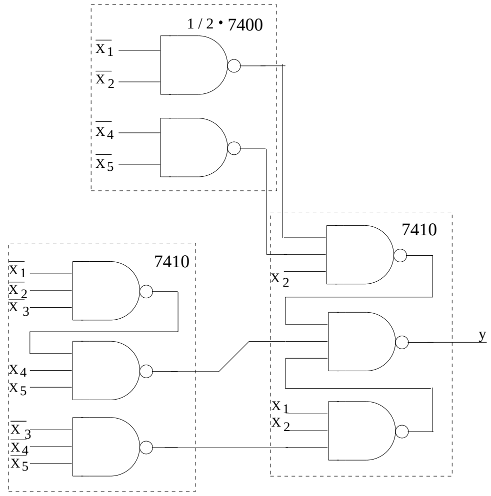


Figure 3.28: Fig21p.28.xfig.ps

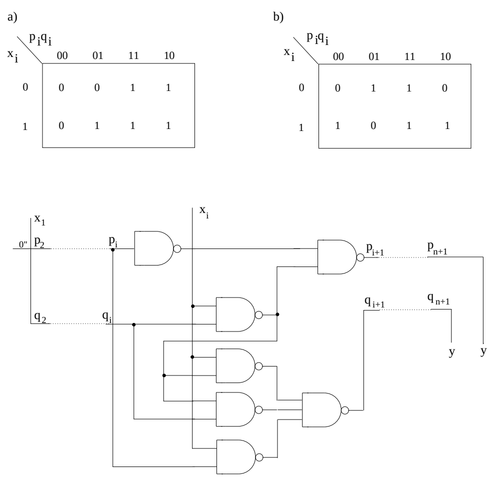


Figure 3.29: Fig21p.29.xfig.ps

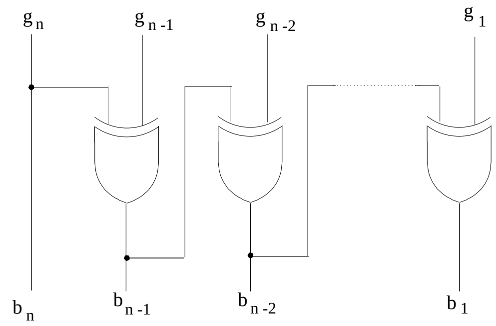


Figure 3.30: Fig21p.30.xfig.ps

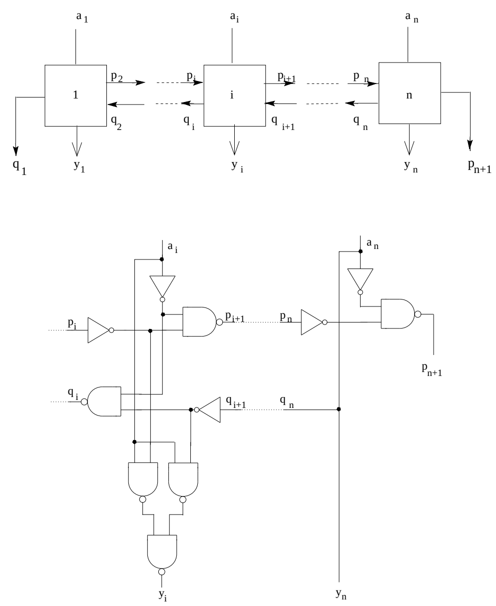


Figure 3.31: Fig21p.31.xfig.ps

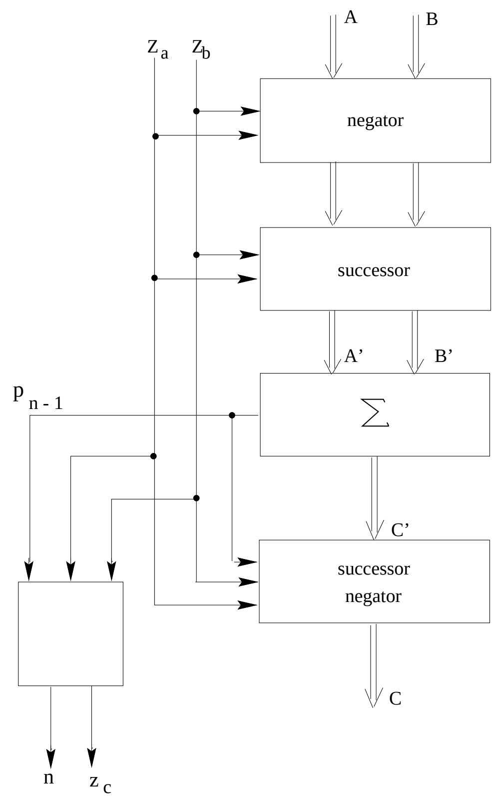
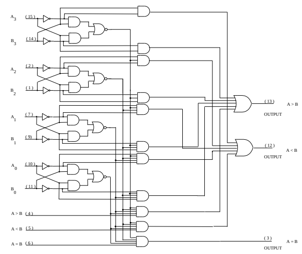
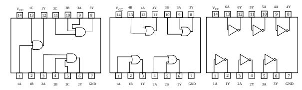
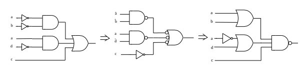
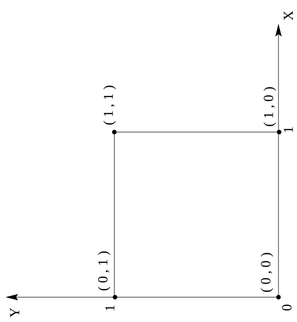


Figure 3.32: Fig21p.32.xfig.ps







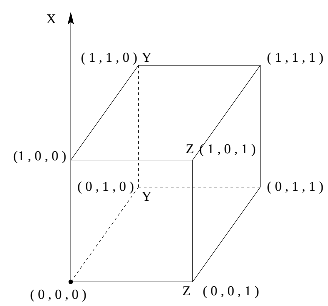

X
Y
(1,1)
(1,0)
(0,1)
(0,0)
1
0
1
0

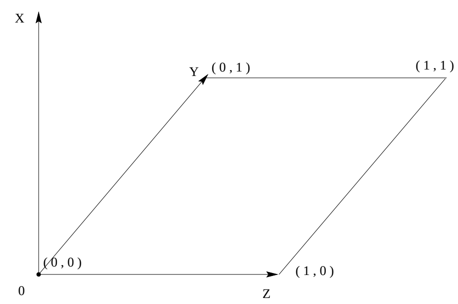
$X_1X_2X_3$ \ $X_4X_5X_6$	000	001	011	010	110	111	101	100
000	0	1	3	2	6	7	5	4
001	8	9	11	10	14	15	13	12
011	24	25	27	26	30	31	29	28
010	16	17	19	18	22	23	21	20
110	48	49	51	50	54	55	53	52
111	56	57	59	58	62	63	61	60
101	40	41	43	42	46	47	45	44
100	32	33	35	34	38	39	37	36

(a)

$X_1X_2X_3$ \ $X_4X_5X_6$	000	001	010	011	100	101	110	111
000	0	1	2	3	4	5	6	7
001	8	9	10	11	12	13	14	15
010	16	17	18	19	20	21	22	23
011	24	25	26	27	28	29	30	31
100	32	33	34	35	36	37	38	39
101	40	41	42	43	44	45	46	47
110	48	49	50	51	52	53	54	55
111	56	57	58	59	60	61	62	63

(b)





Chapter 4

FIGURES 21.18-21.39

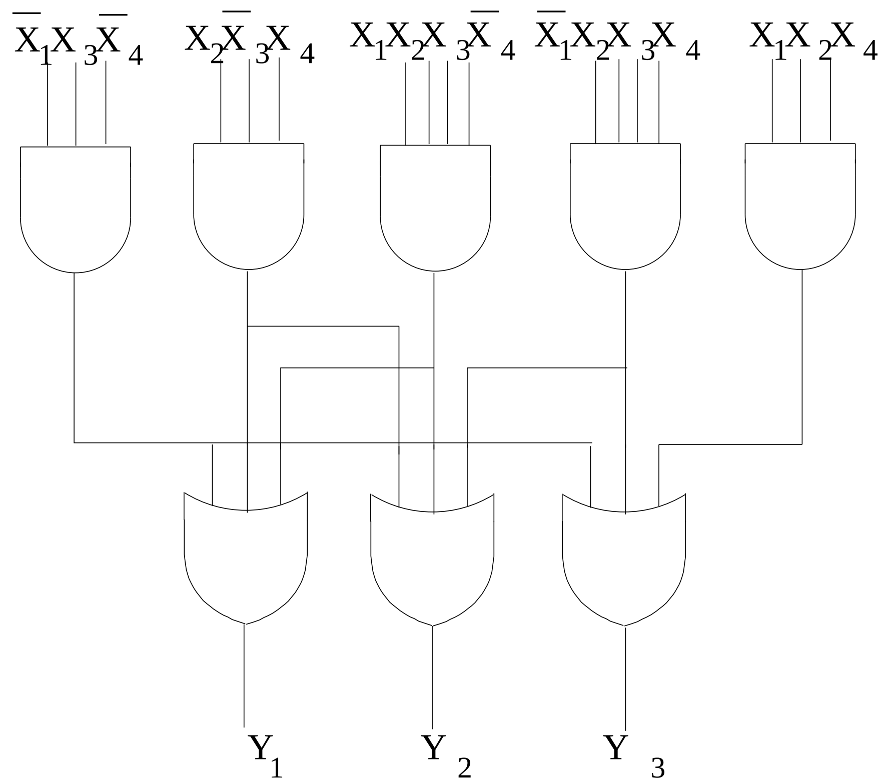


Figure 4.1: Fig21.18.xfig.ps

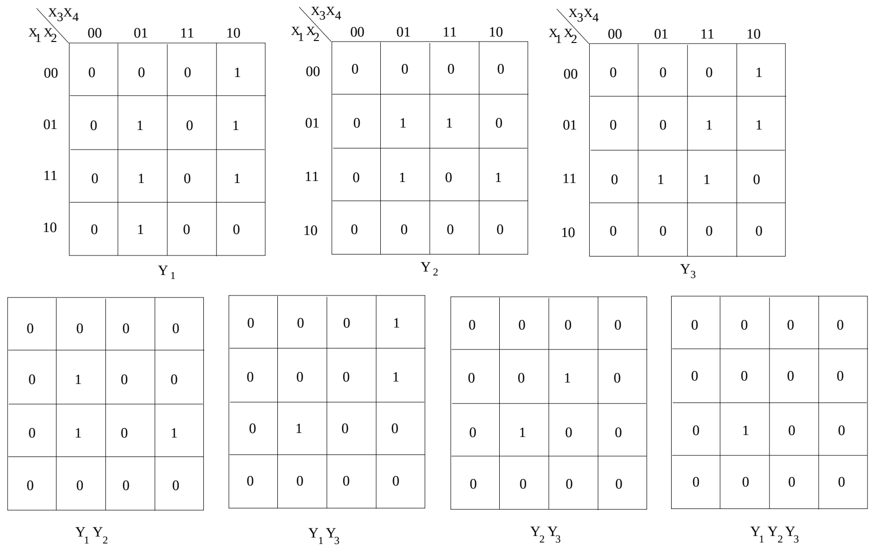


Figure 4.2: Fig21.19.xfig.ps

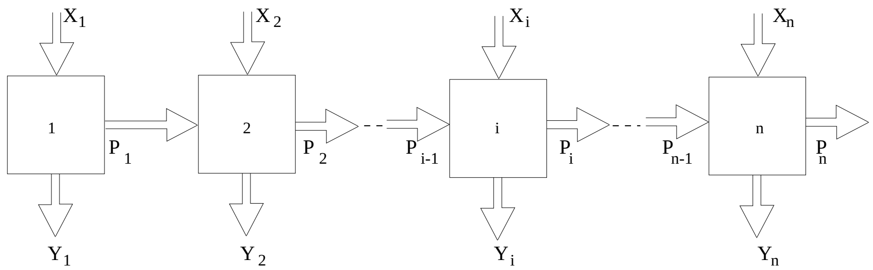


Figure 4.3: Fig21.20.xfig.ps

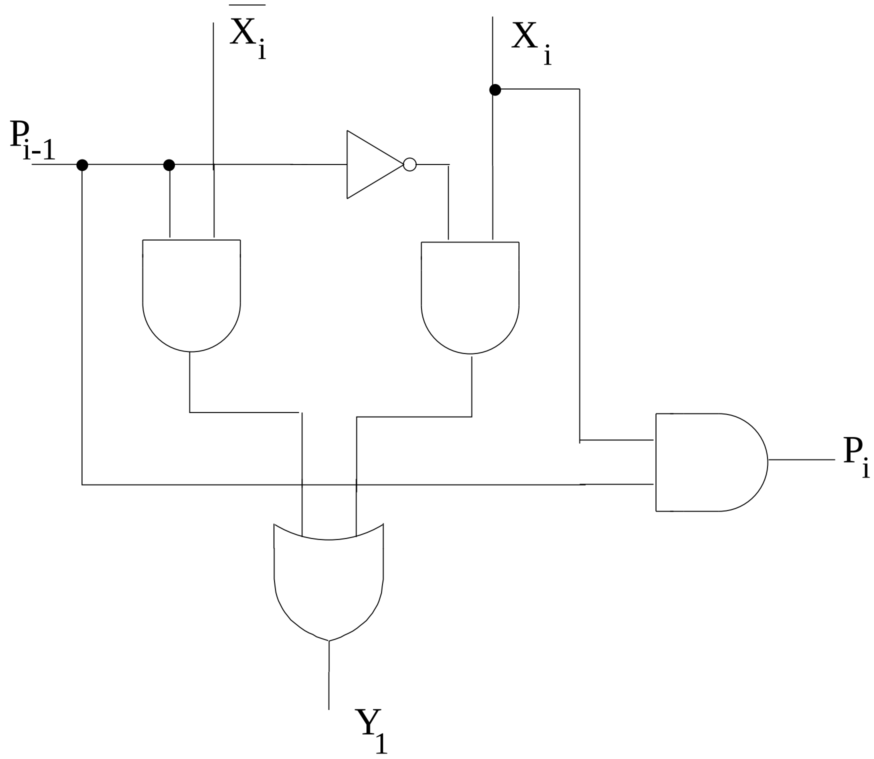


Figure 4.4: Fig21.21.xfig.ps

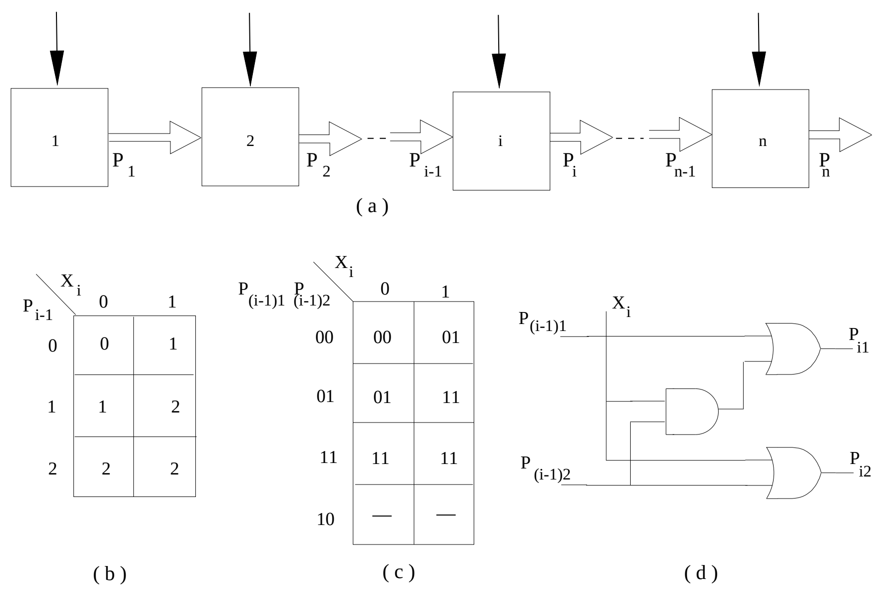


Figure 4.5: Fig21.22.xfig.ps

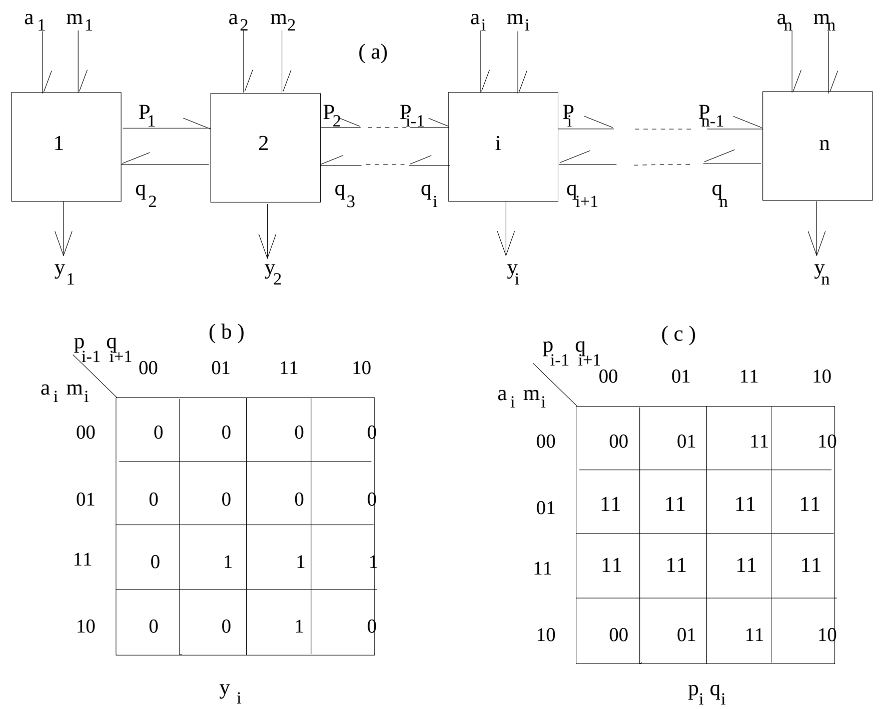


Figure 4.6: Fig21.23a.xfig.ps

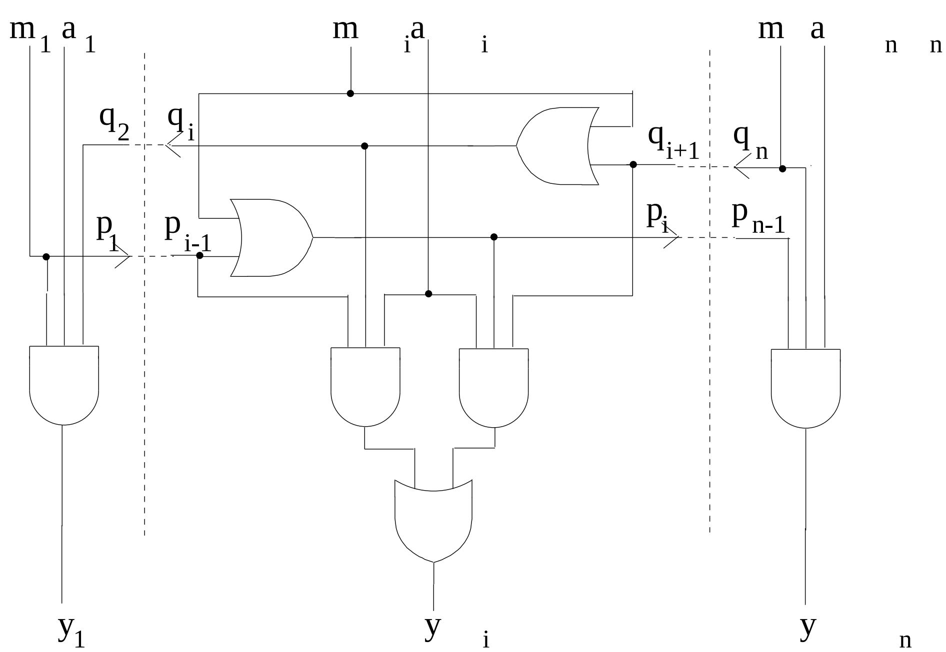


Figure 4.7: Fig21.23b.xfig.ps

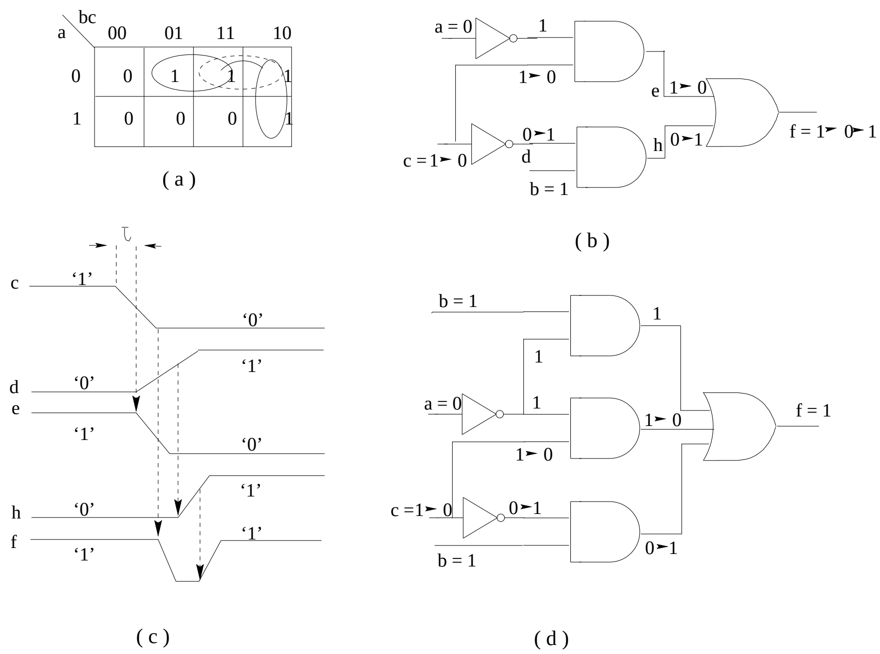


Figure 4.8: Fig21.24.xfig.ps

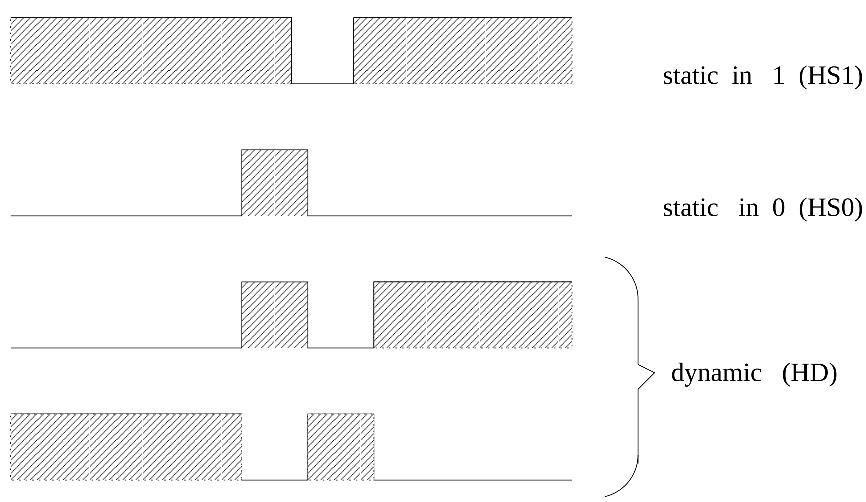


Figure 4.9: Fig21.25.xfig.ps

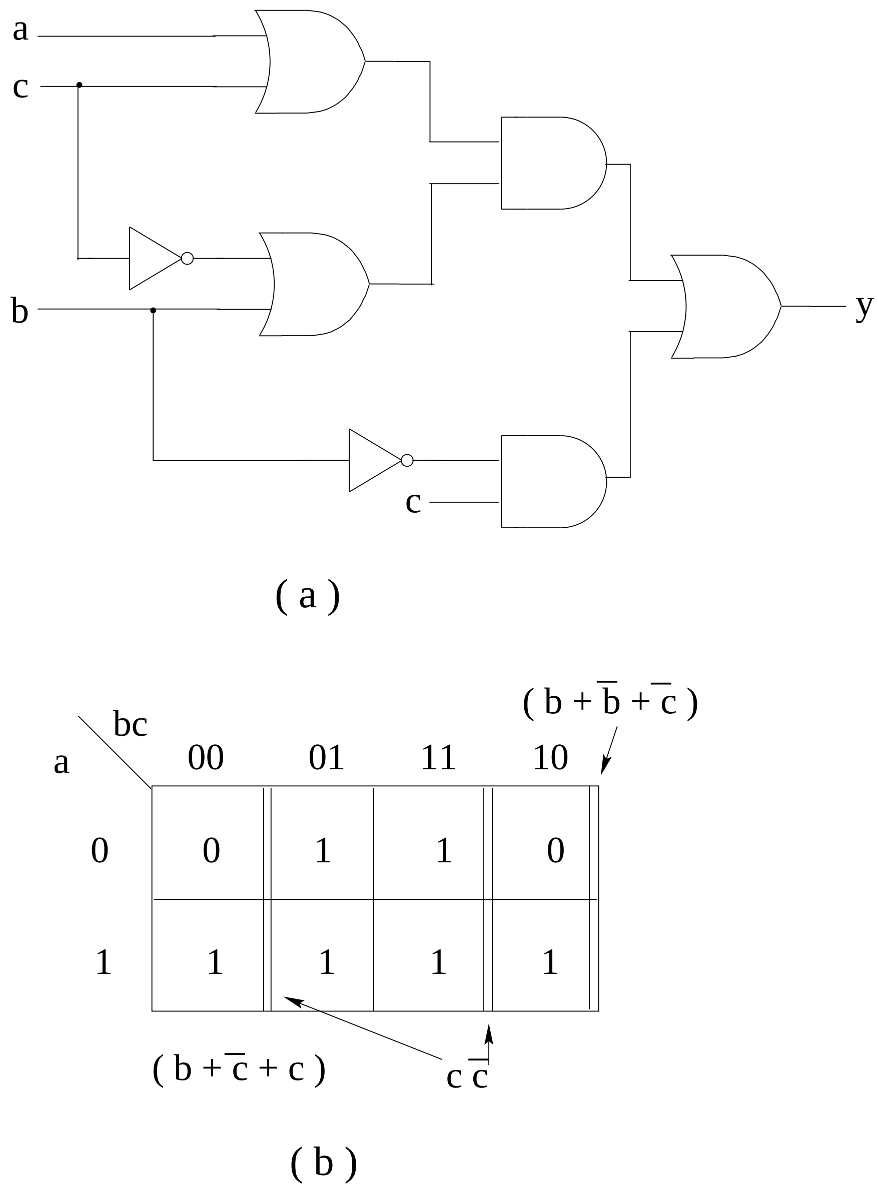


Figure 4.10: Fig21.26.xfig.ps

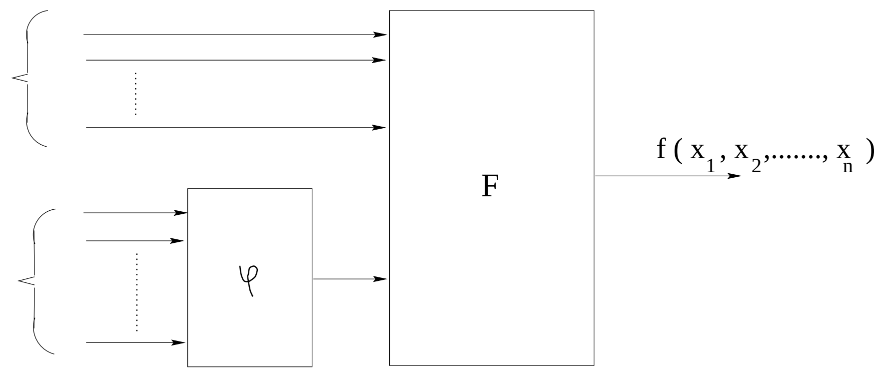


Figure 4.11: Fig21.27.xfig.ps

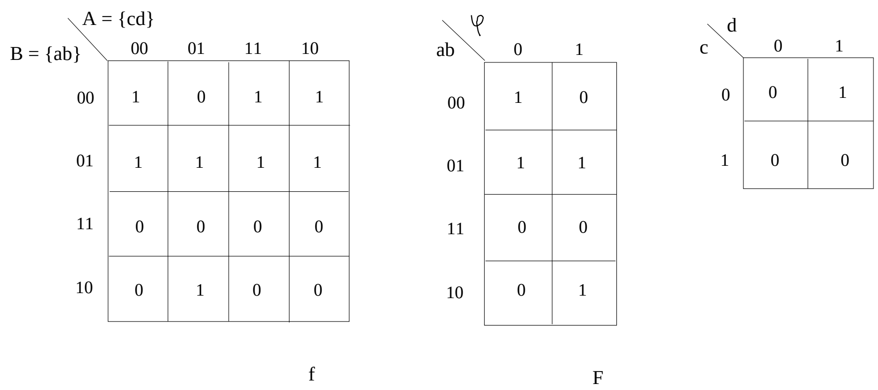


Figure 4.12: Fig21.28.xfig.ps

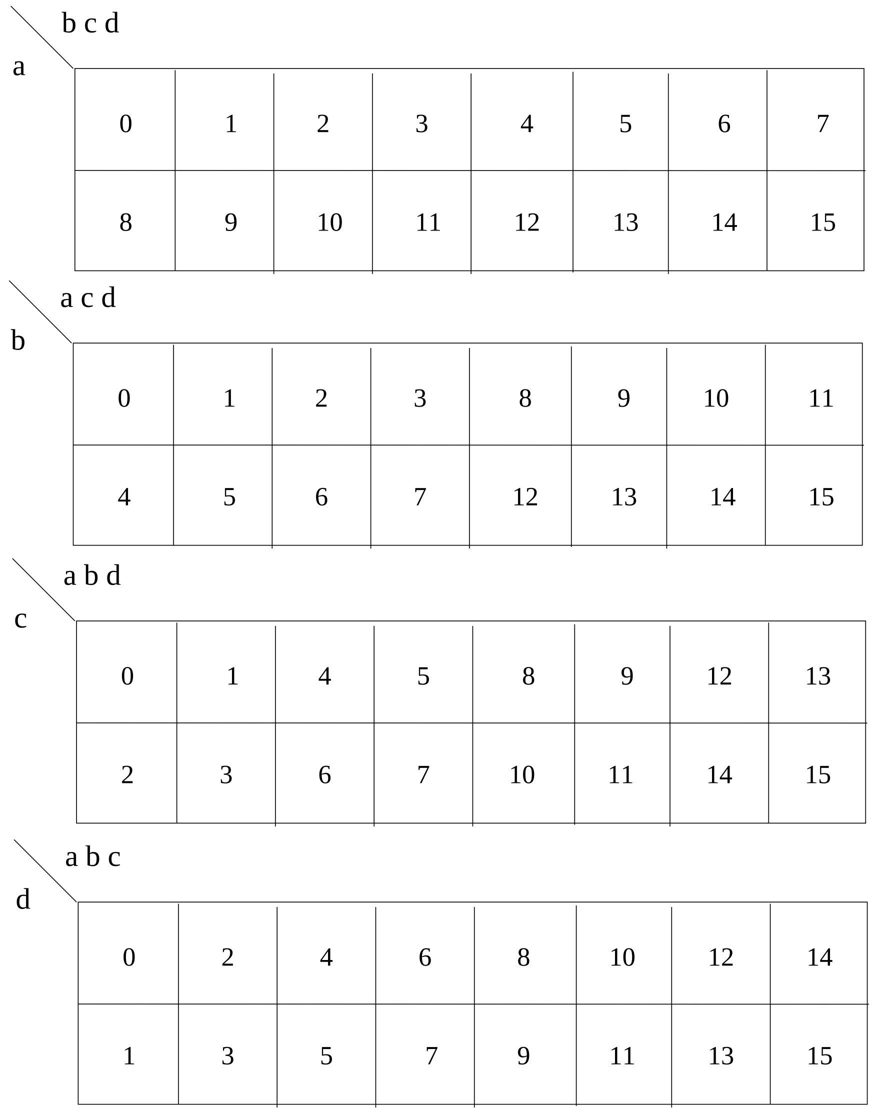


Figure 4.13: Fig21.29a.xfig.ps

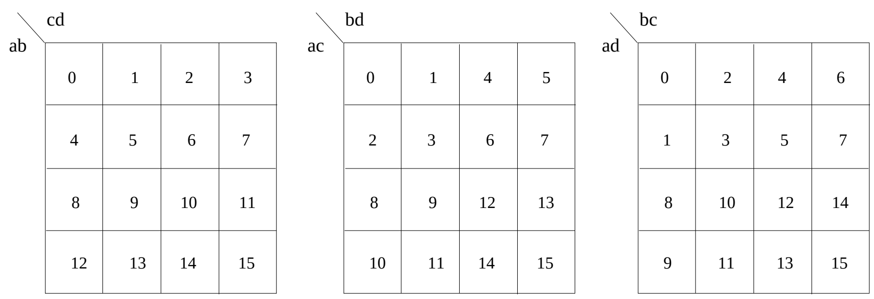


Figure 4.14: Fig21.29b.xfig.ps

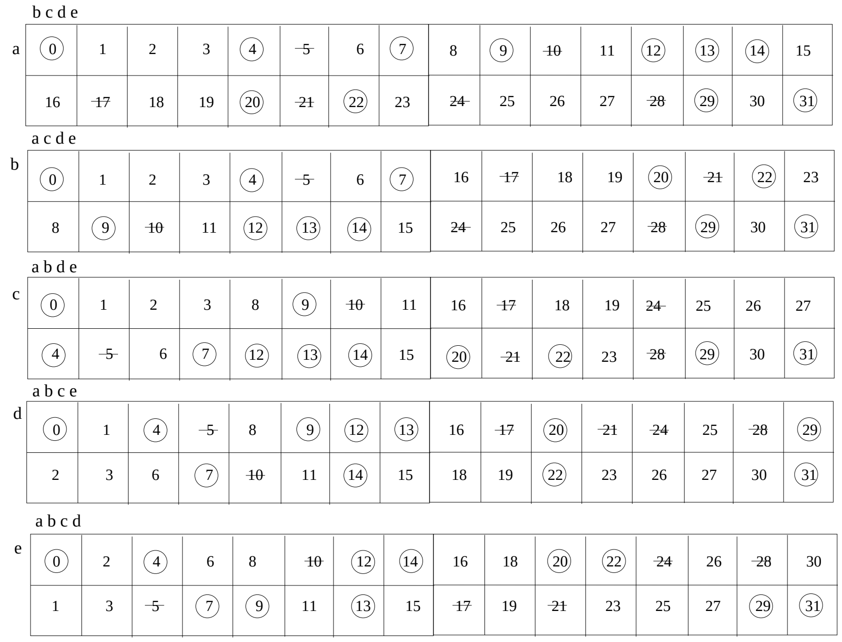


Figure 4.15: Fig21.30a.xfig.ps

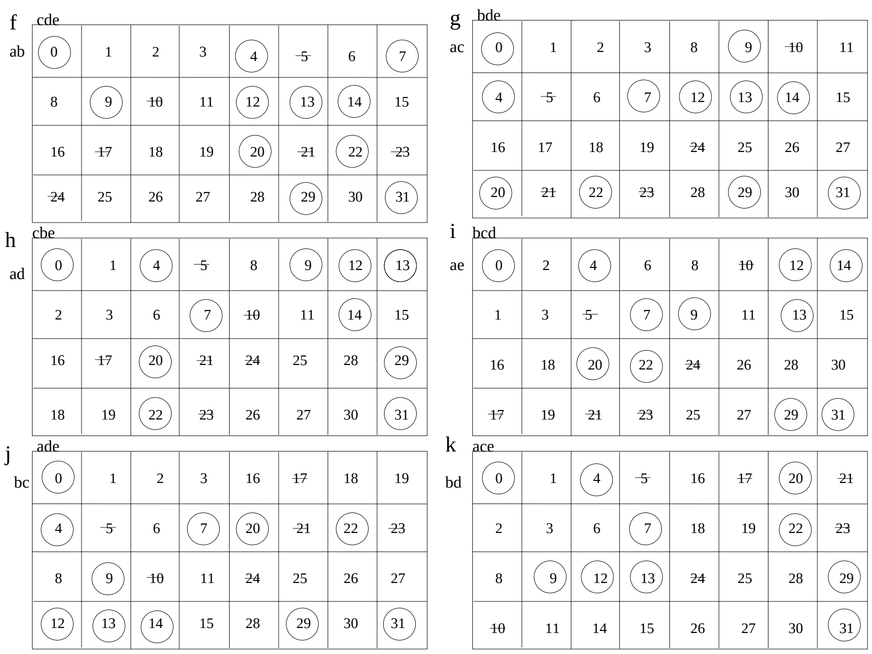


Figure 4.16: Fig21.30b.xfig.ps

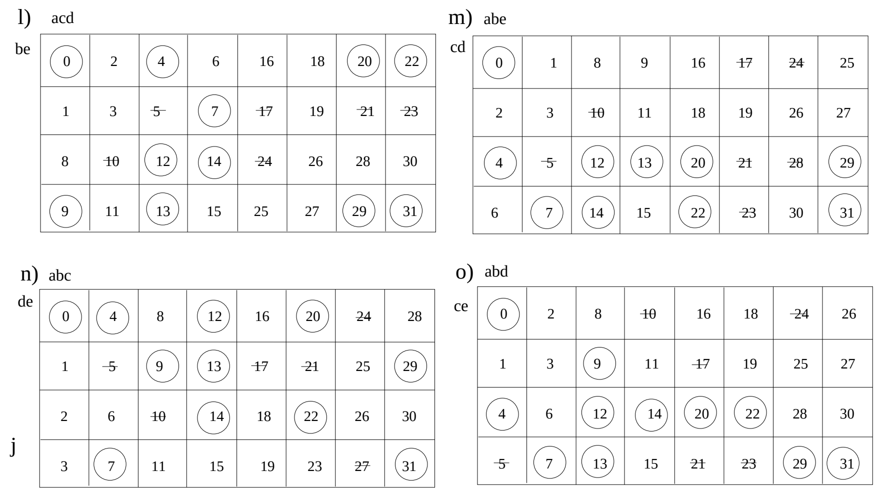


Figure 4.17: Fig21.30c.xfig.ps

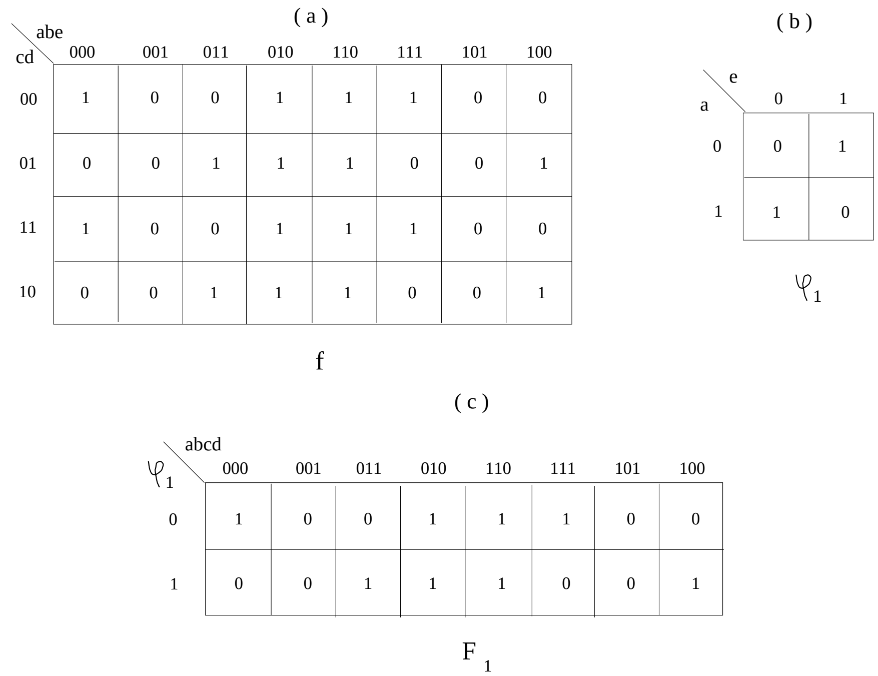


Figure 4.18: Fig21.31.xfig.ps

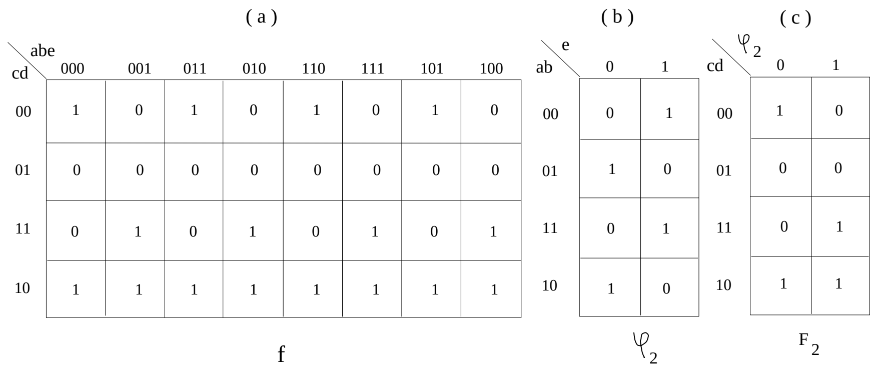
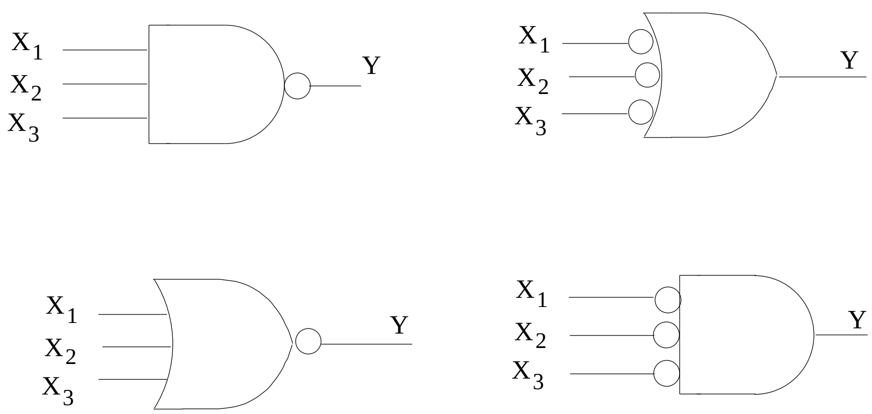


Figure 4.19: Fig21.32.xfig.ps



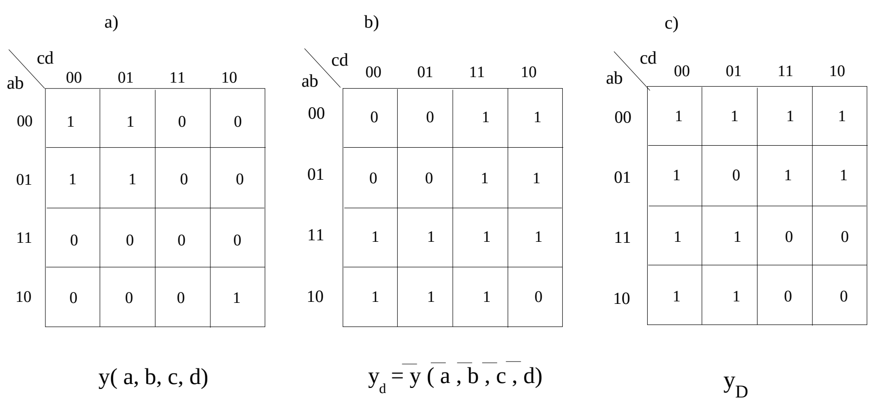


Figure 4.21: Fig21.35.xfig.ps

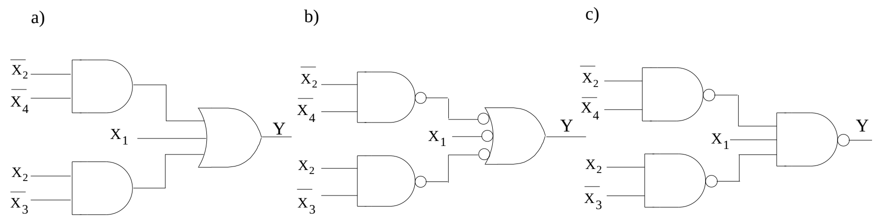


Figure 4.22: Fig21.36.xfig.ps

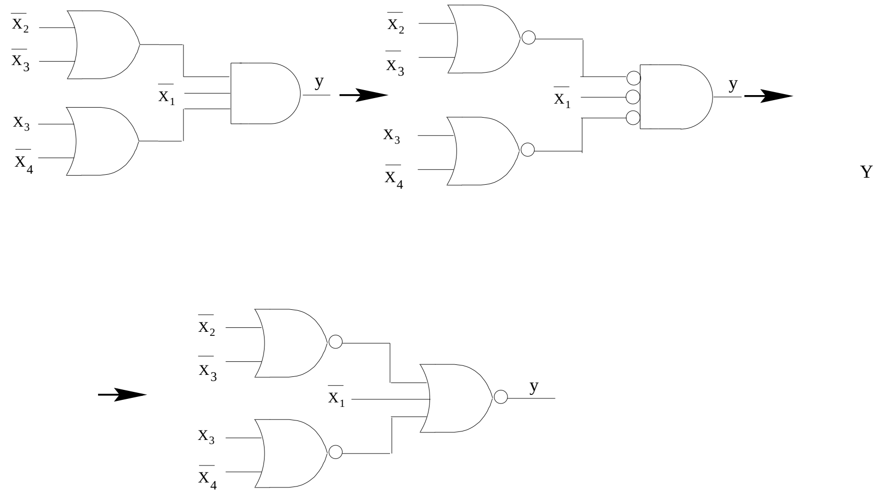


Figure 4.23: Fig21.37.xfig.ps

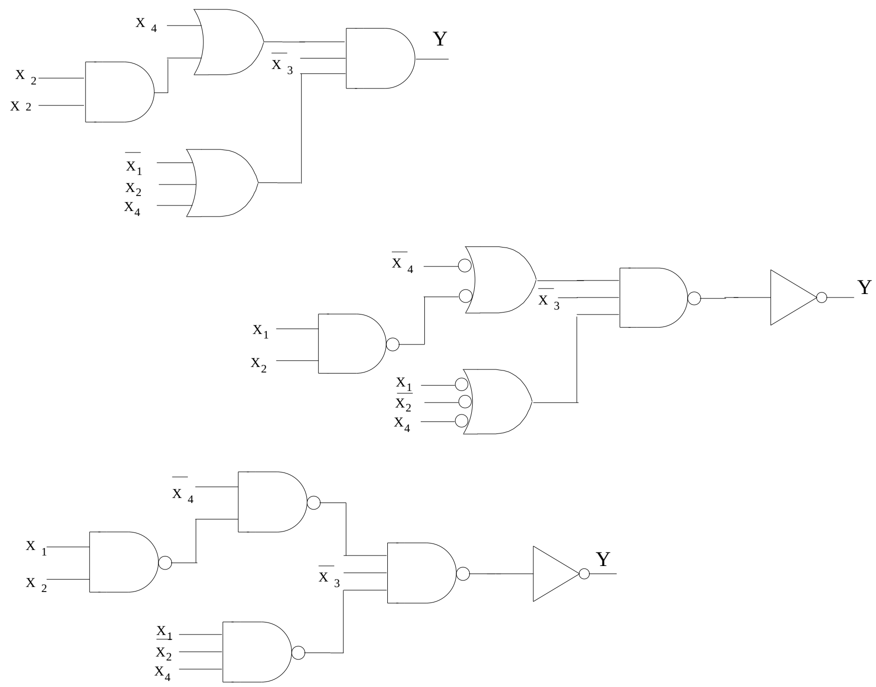


Figure 4.24: Fig21.38.xfig.ps

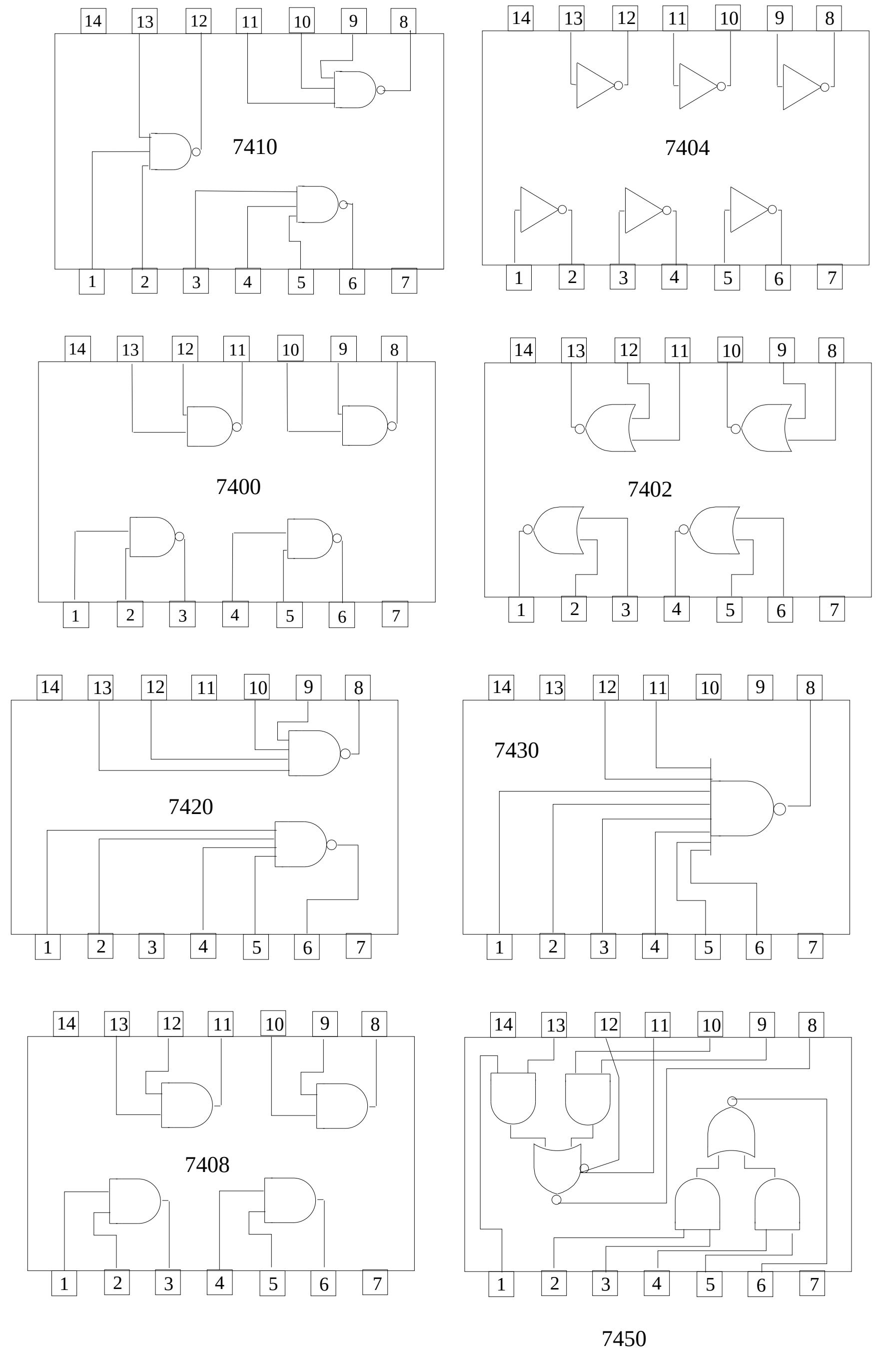


Figure 4.25: Fig21.39.xfig.ps

ab \ c de	000	001	011	010	110	111	101	100
00	1	0	0	-	-	0	0	-
01	0	-	1	1	0	0	1	1
11	0	1	0	1	0	0	-	0
10	0	0	1	1	1	-	0	0

ab \ c de	000	001	011	010	110	111	101	100
00	0	1	1	0	-	1	1	-
01	0	0	0	-	1	1	1	0
11	1	1	1	-	1	1	1	1
10	0	-	1	0	0	1	0	-

Figure 4.26: Fig21p.1.xfig.ps

Chapter 5

FIGURES TO CHAPTERS ABOUT TANT, NAND MULTI-LEVEL SYNTHESIS

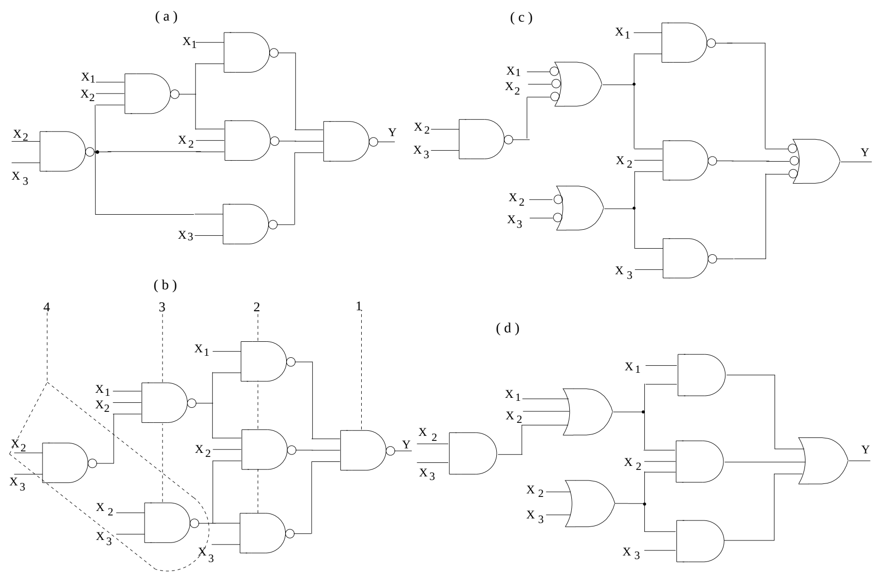


Figure 5.1: Fig31.7.xfig.ps

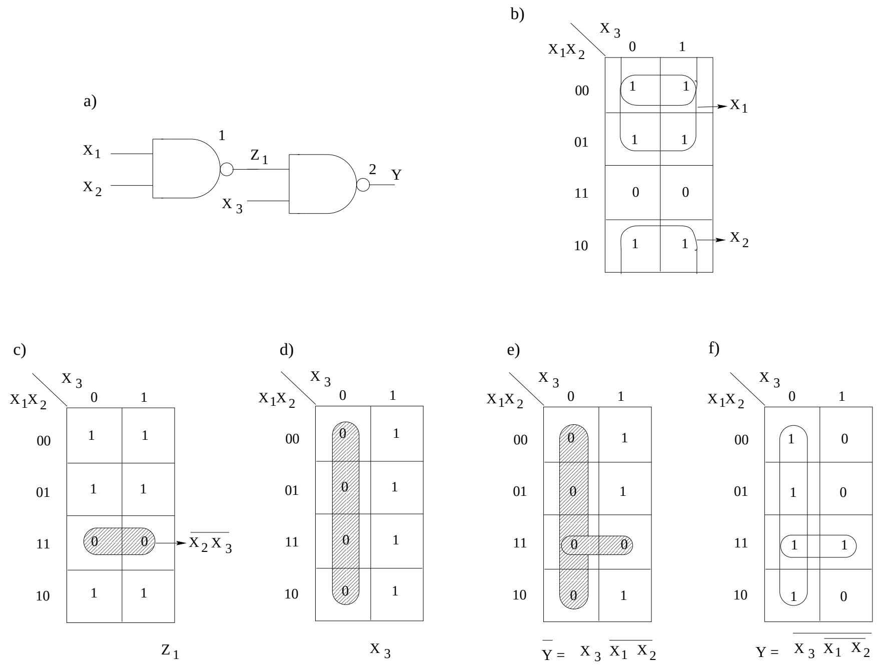


Figure 5.2: Fig31.8.xfig.ps

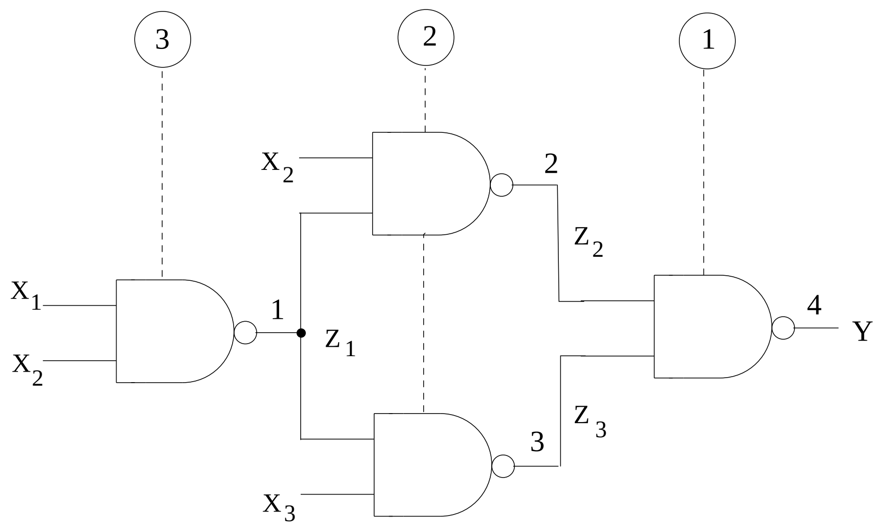


Figure 5.3: Fig31.9.xfig.ps

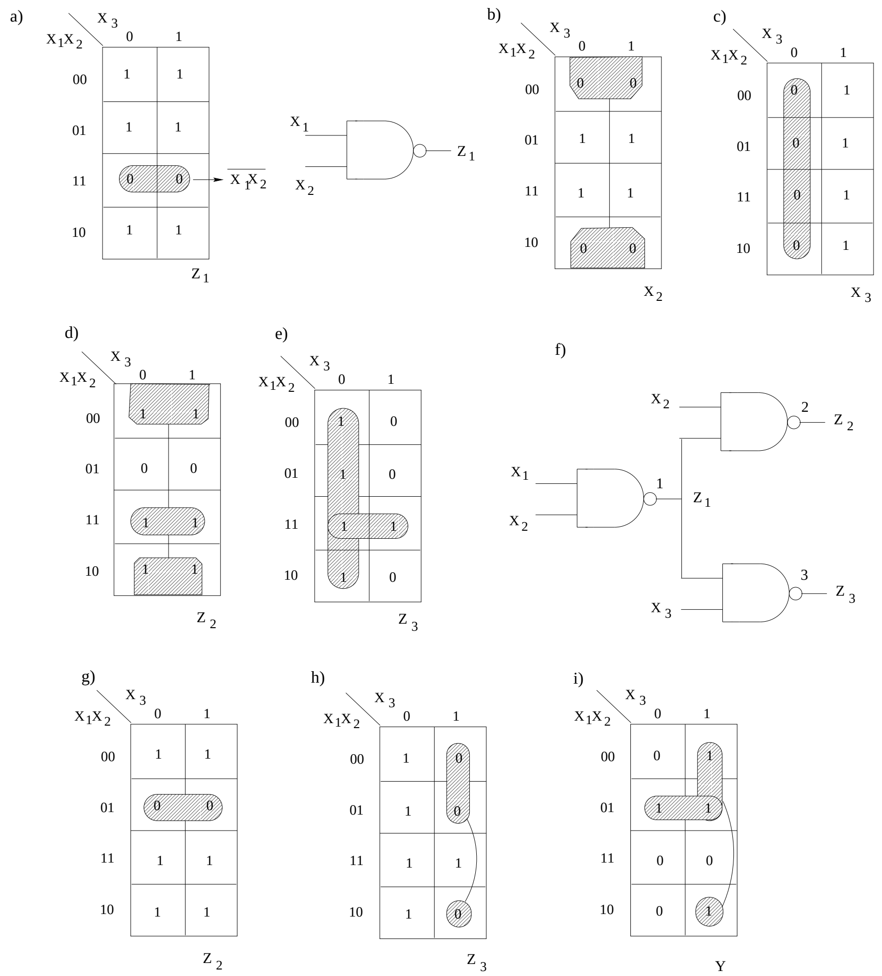


Figure 5.4: Fig31.10.xfig.ps

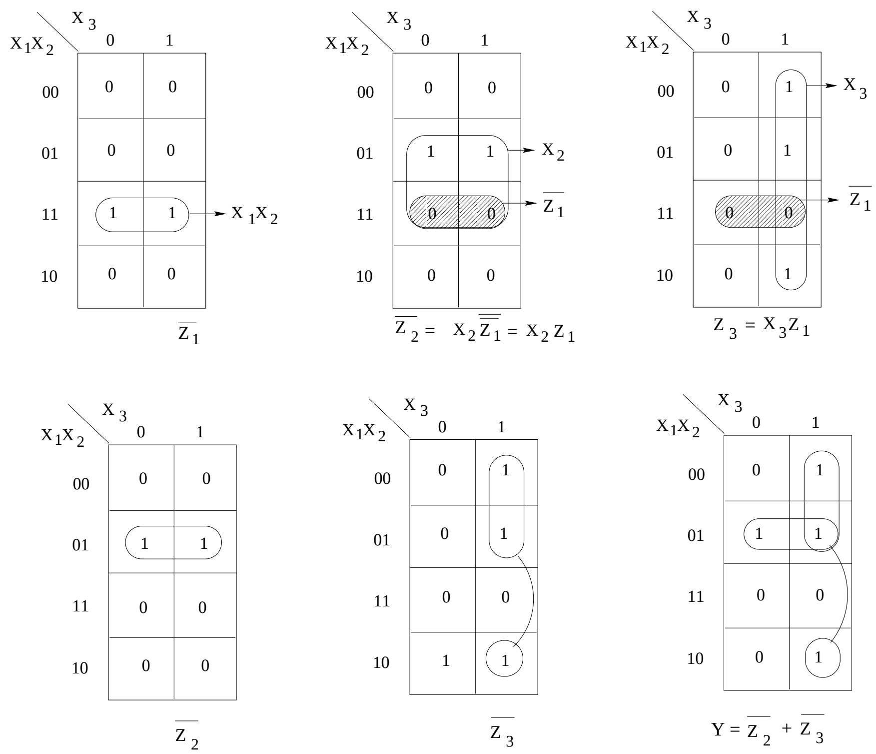


Figure 5.5: Fig31.11.xfig.ps

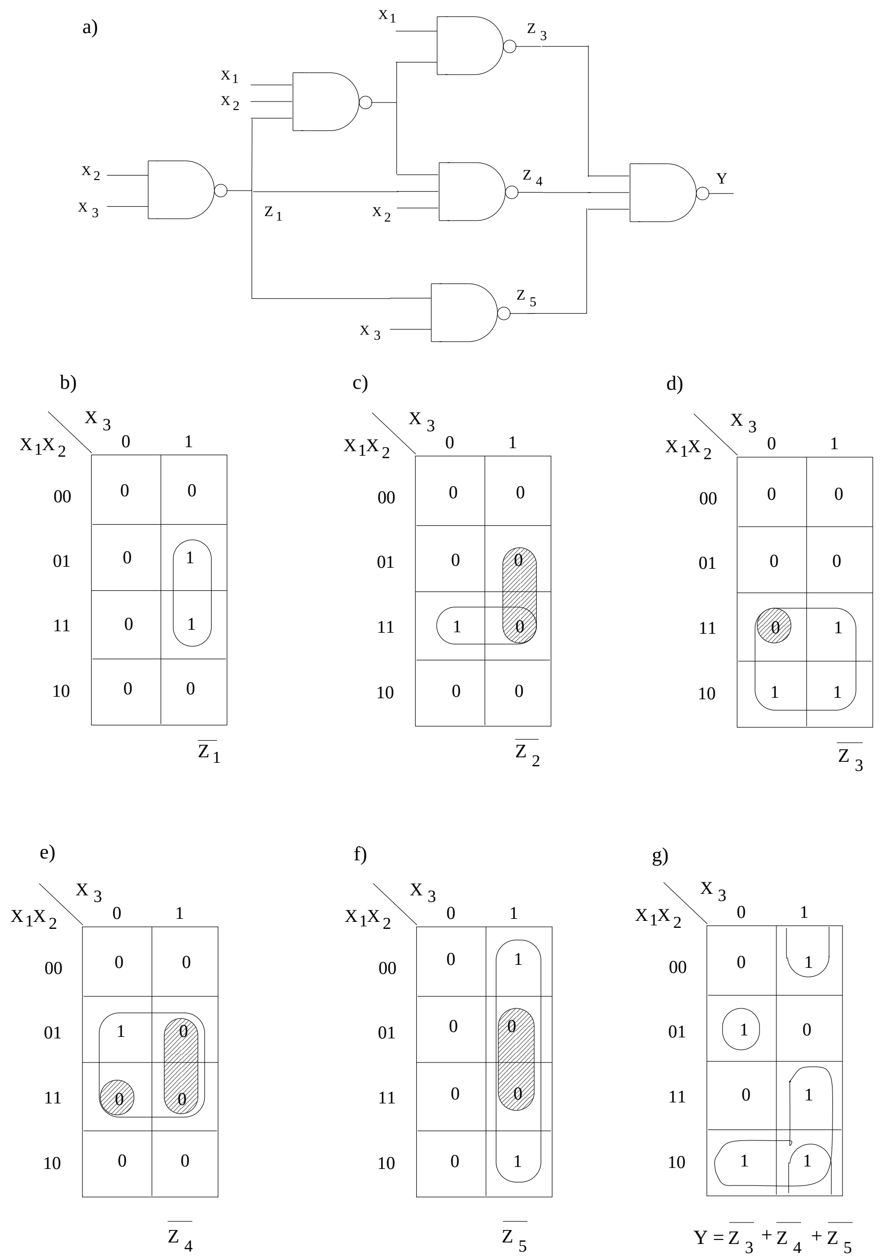


Figure 5.6: Fig31.12.xfig.ps

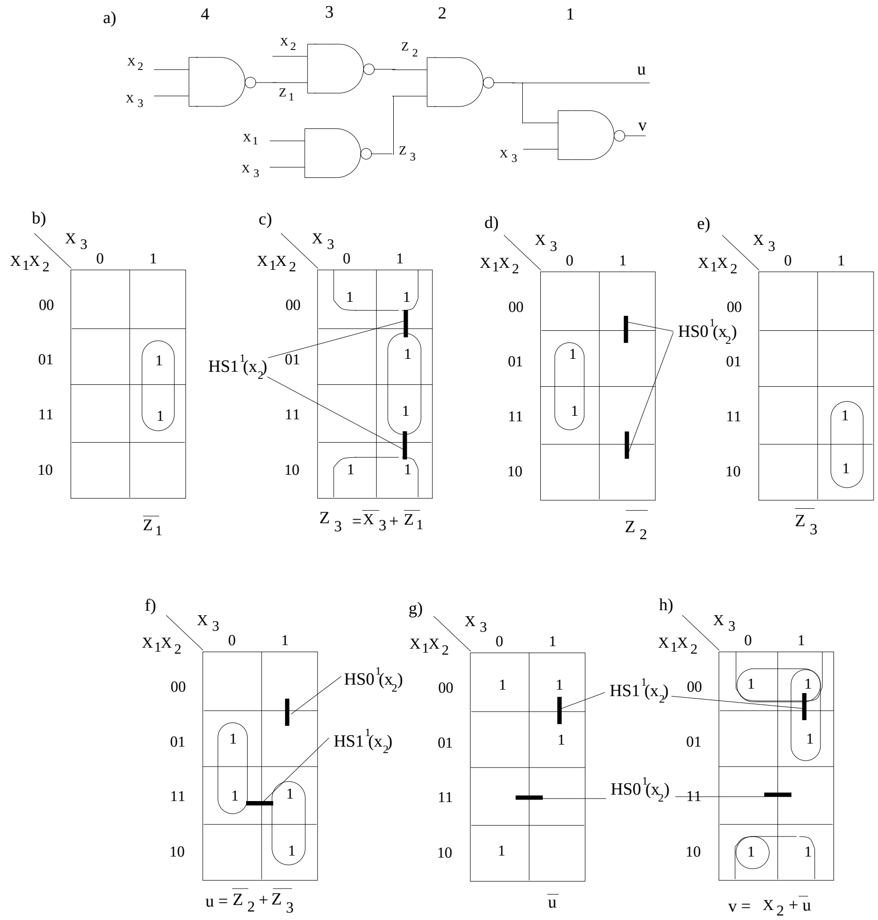


Figure 5.7: Fig31.13.xfig.ps

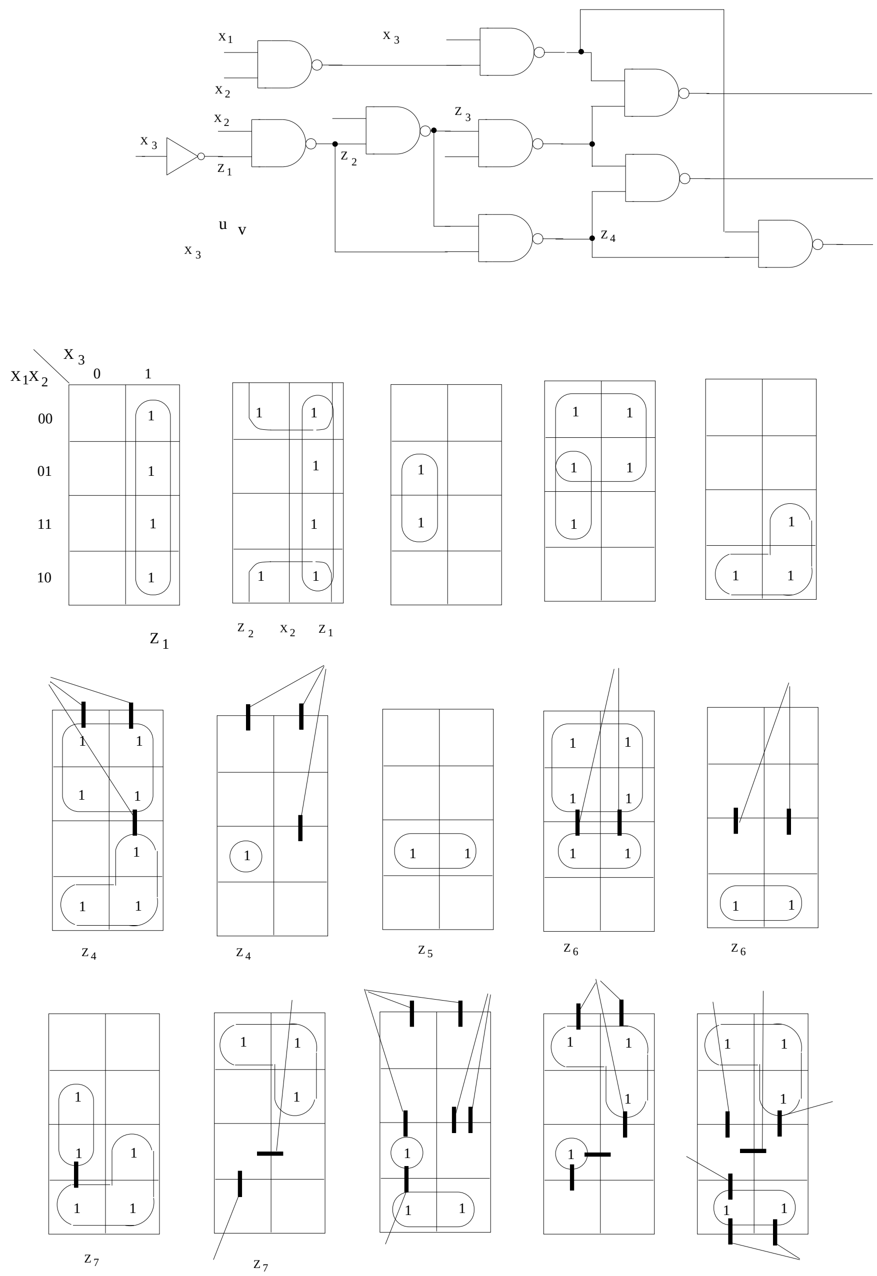


Figure 5.8: Fig31.14.xfig.ps

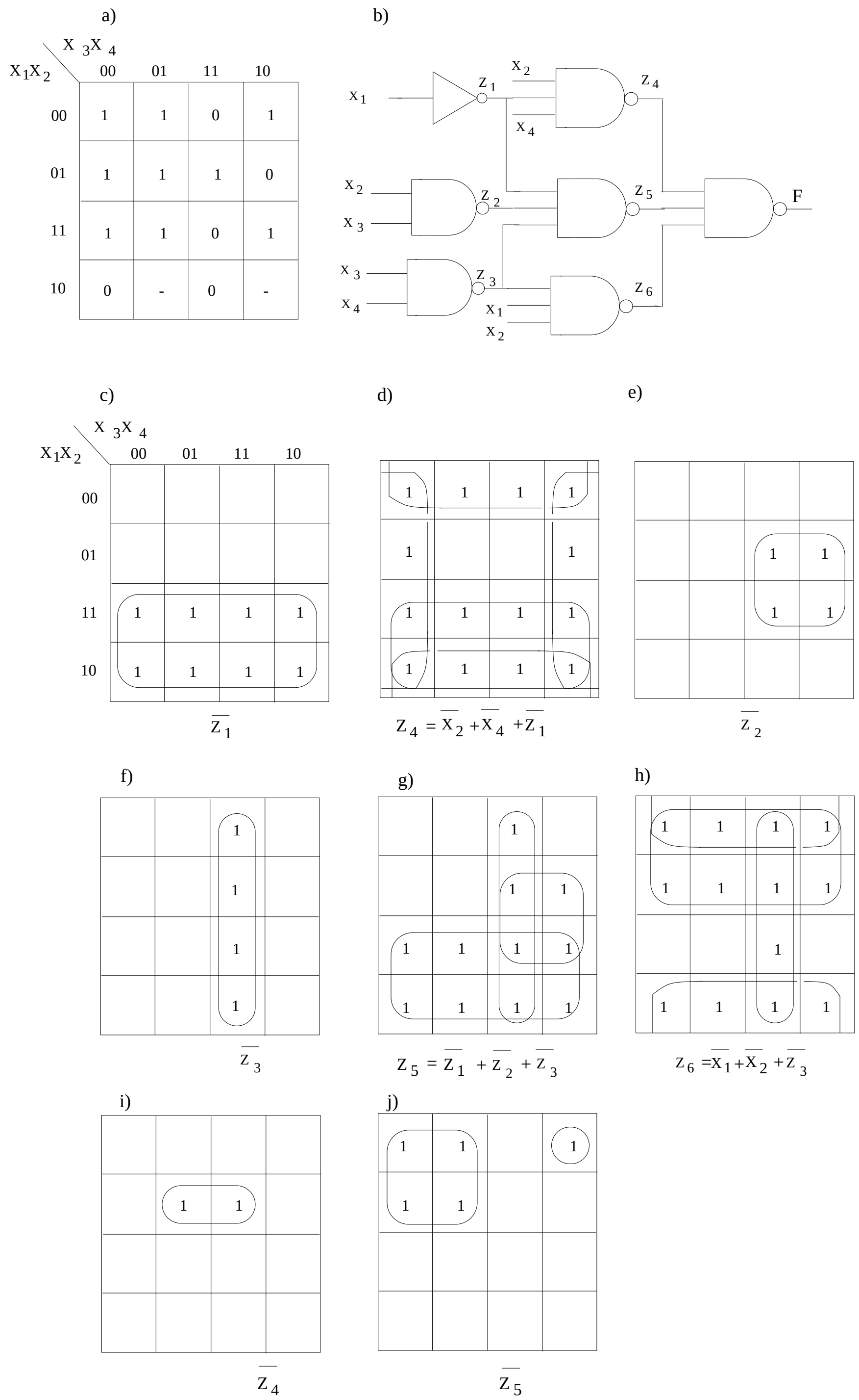


Figure 5.9: Fig31.15.xfig.ps

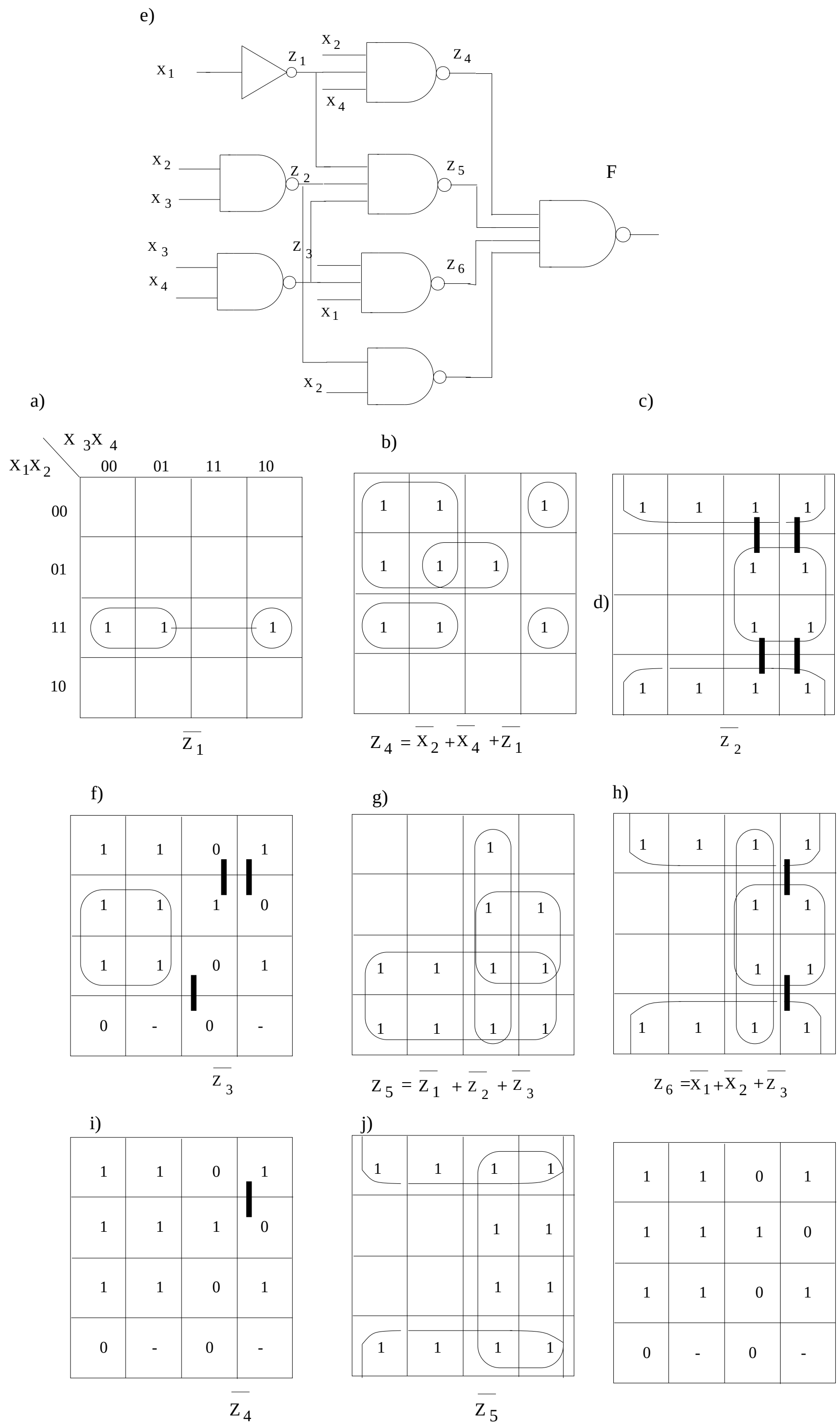


Figure 5.10: Fig31.16.xfig.ps

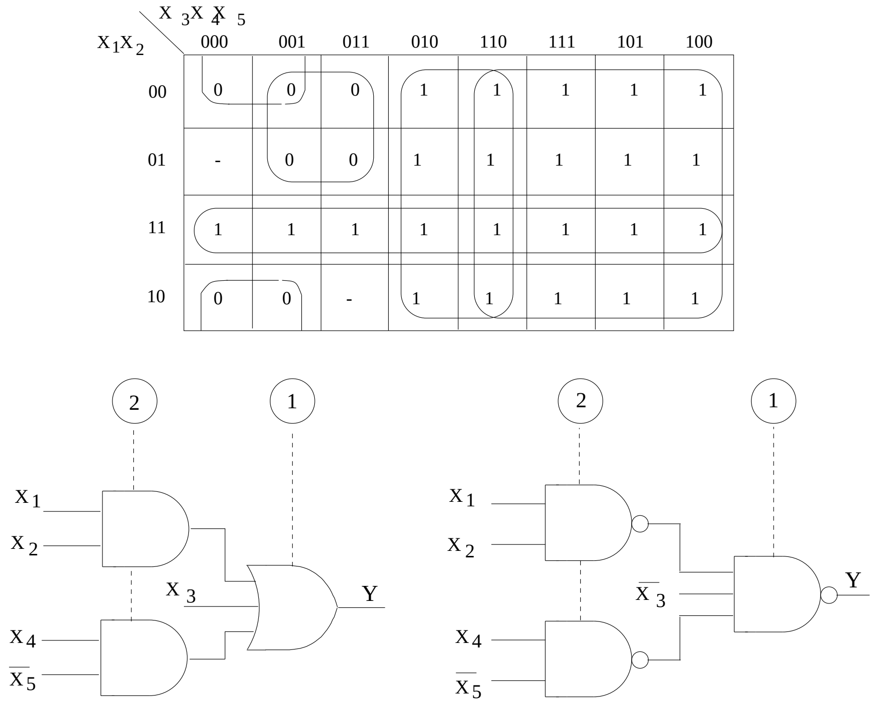


Figure 5.11: Fig31.17.xfig.ps

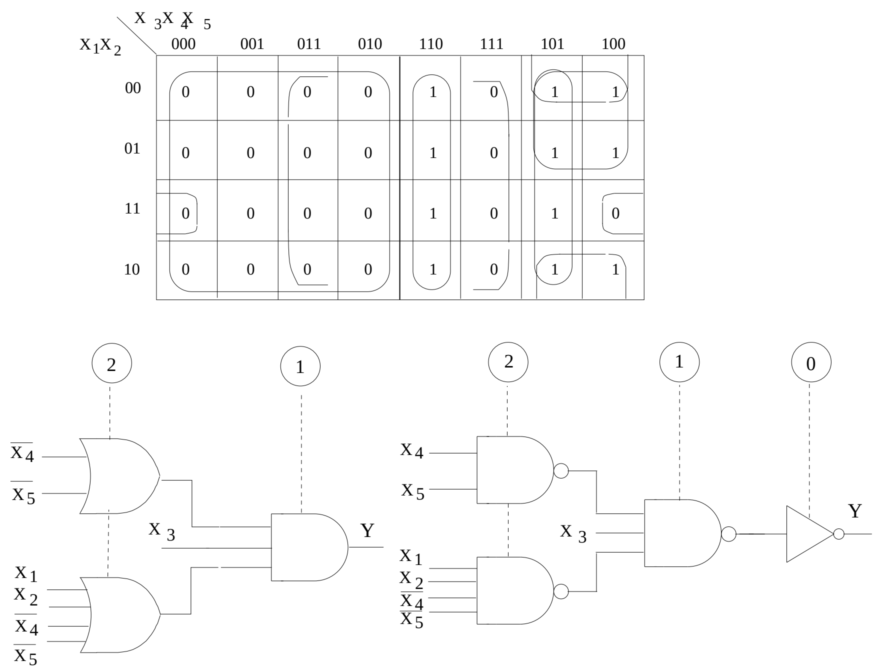


Figure 5.12: Fig31.18.xfig.ps

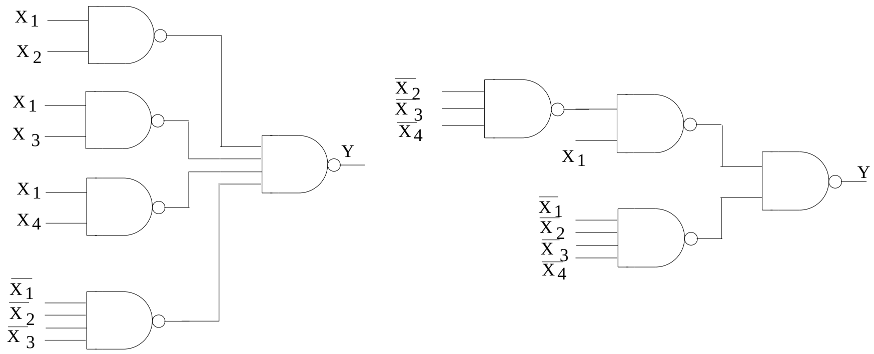


Figure 5.13: Fig31.19.xfig.ps

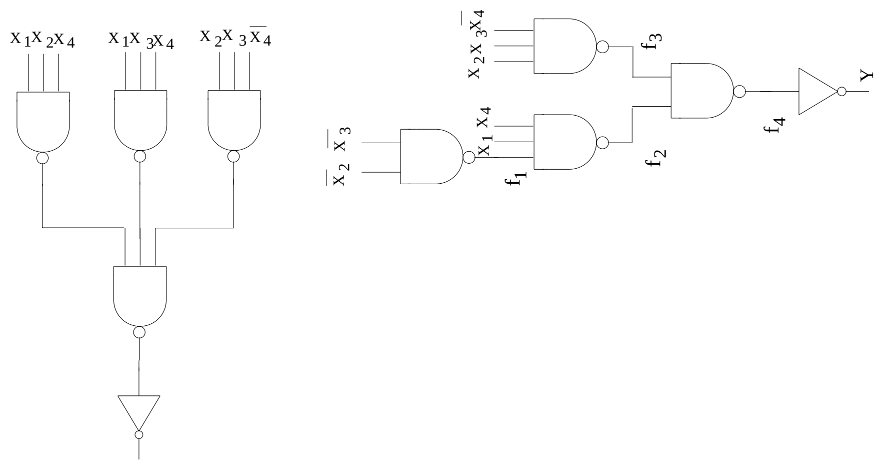


Figure 5.14: Fig31.20.xfig.ps

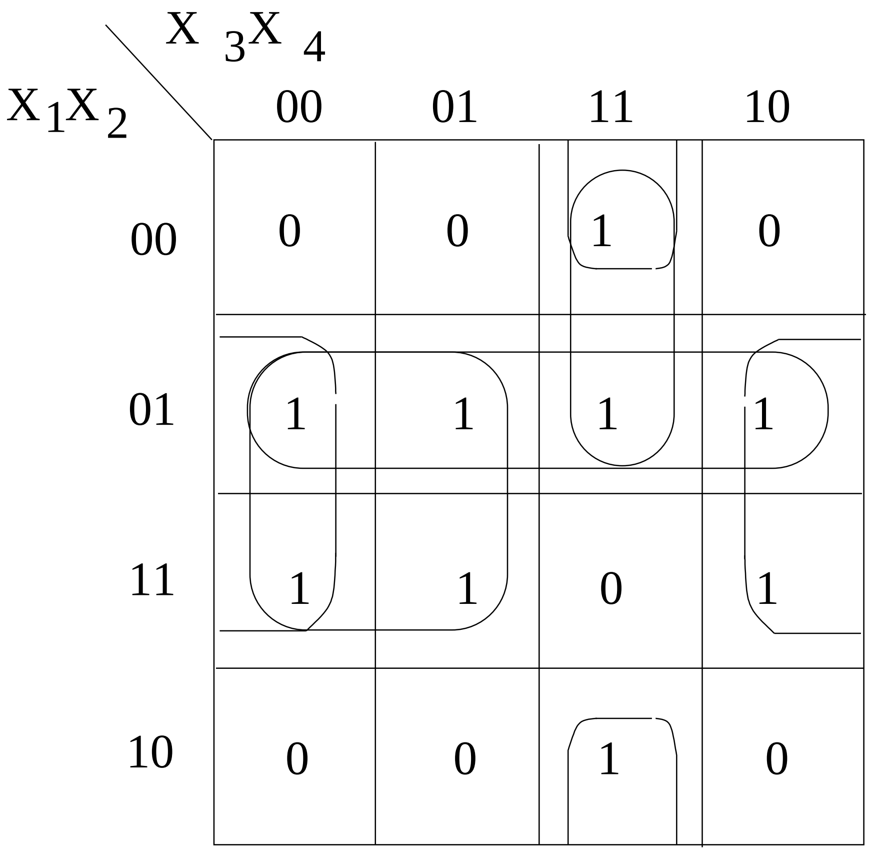


Figure 5.15: Fig31.21.xfig.ps

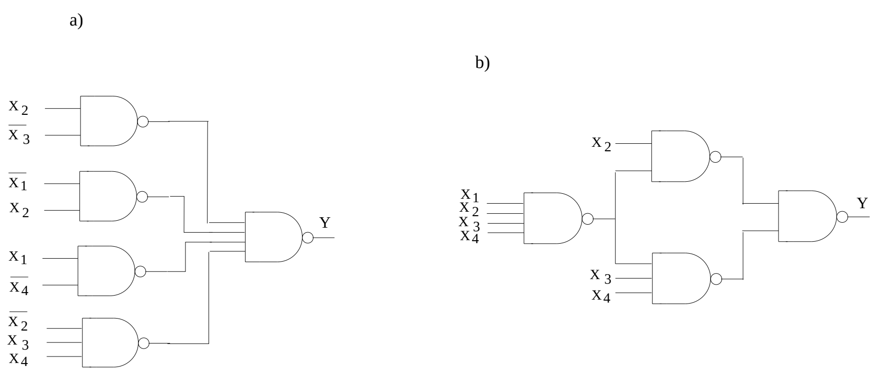


Figure 5.16: Fig31.22.xfig.ps

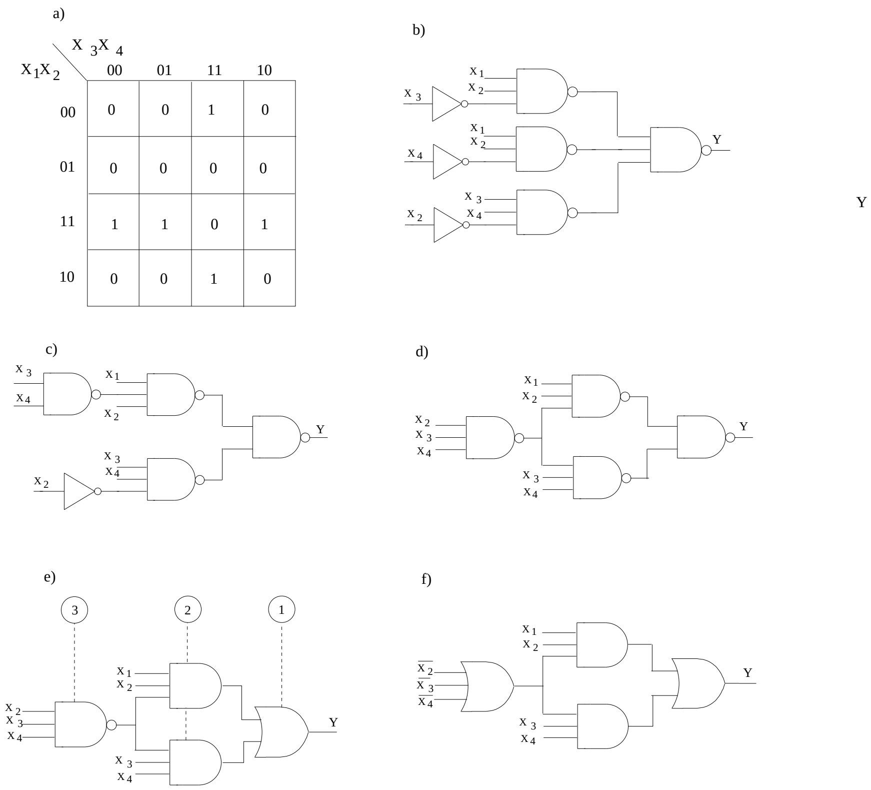


Figure 5.17: Fig31.23.xfig.ps

Figure 5.18: Fig31.24.xfig.ps

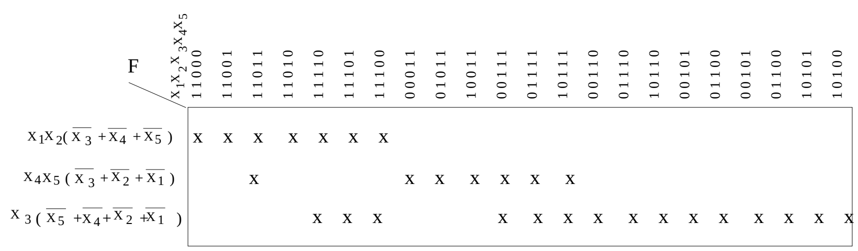


Figure 5.19: Fig31.25.xfig.ps

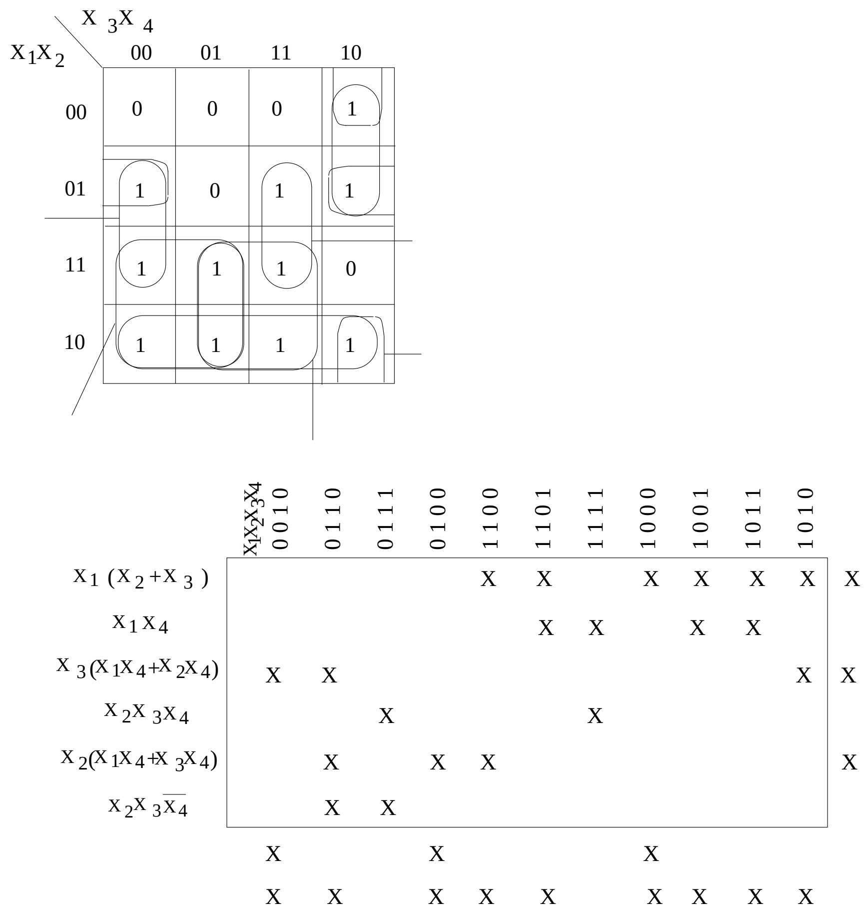


Figure 5.20: Fig31.26.xfig.ps

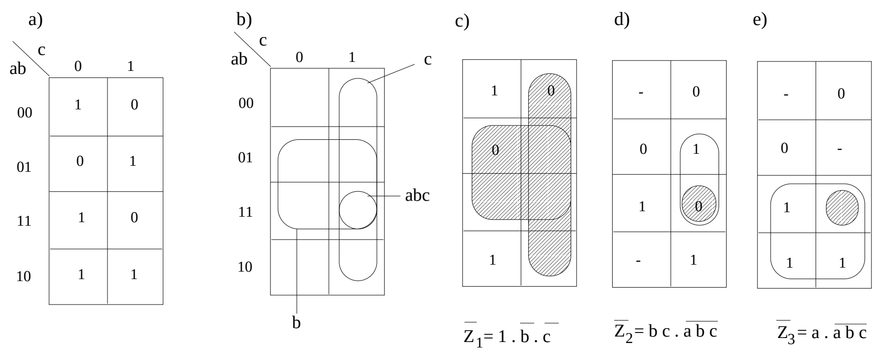


Figure 5.21: Fig31.27.xfig.ps

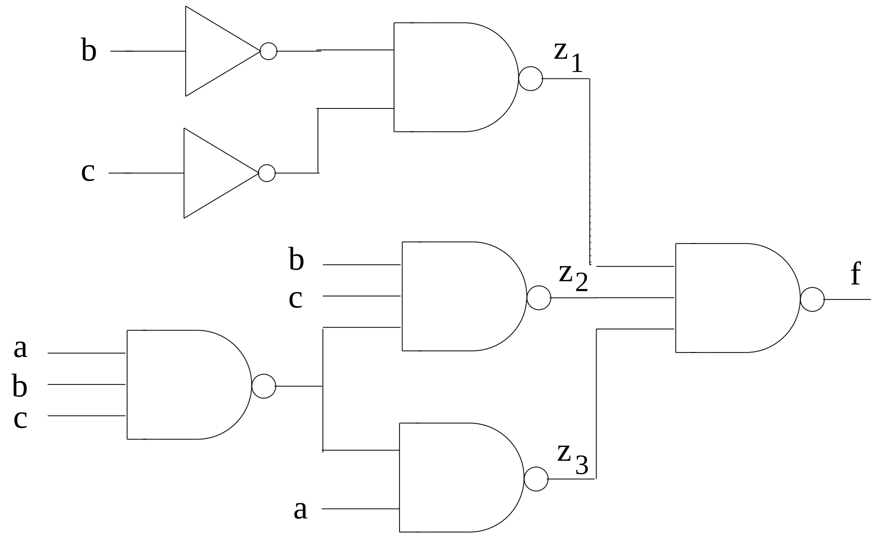


Figure 5.22: Fig31.28.xfig.ps

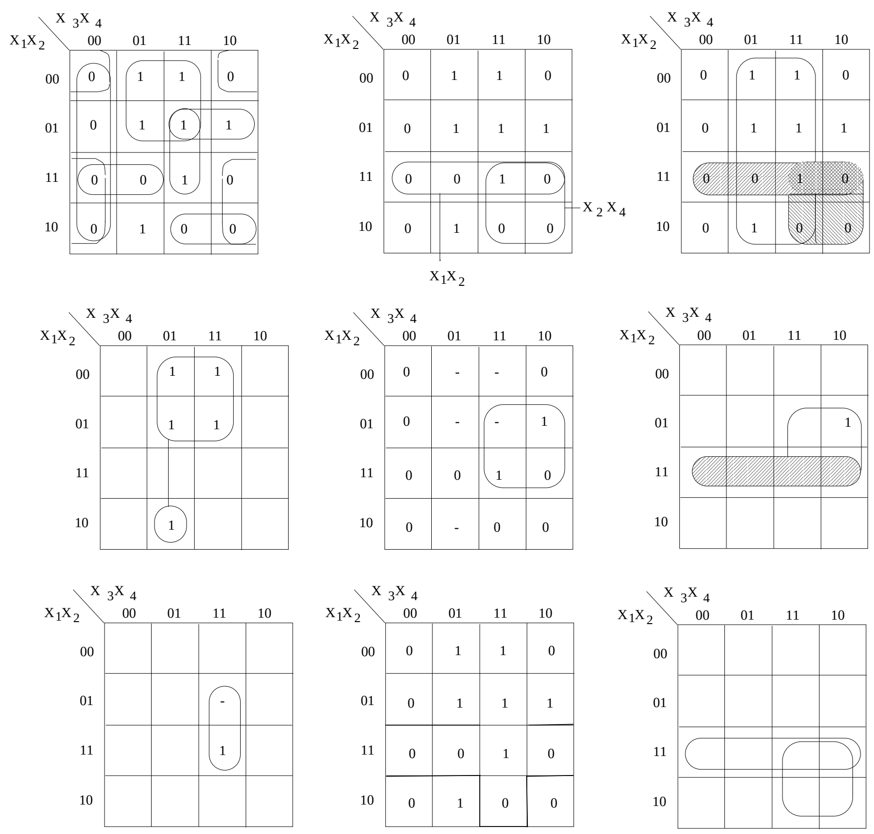


Figure 5.23: Fig31.29.xfig.ps

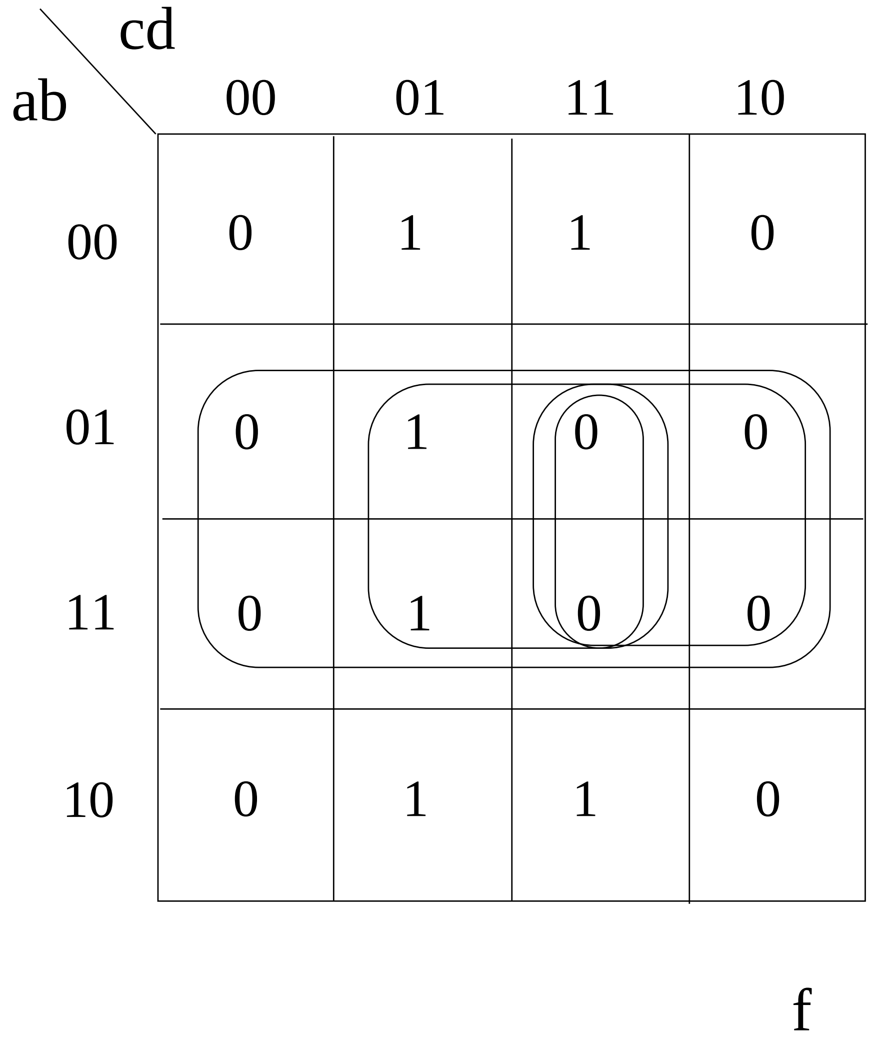


Figure 5.24: Fig31.30.xfig.ps

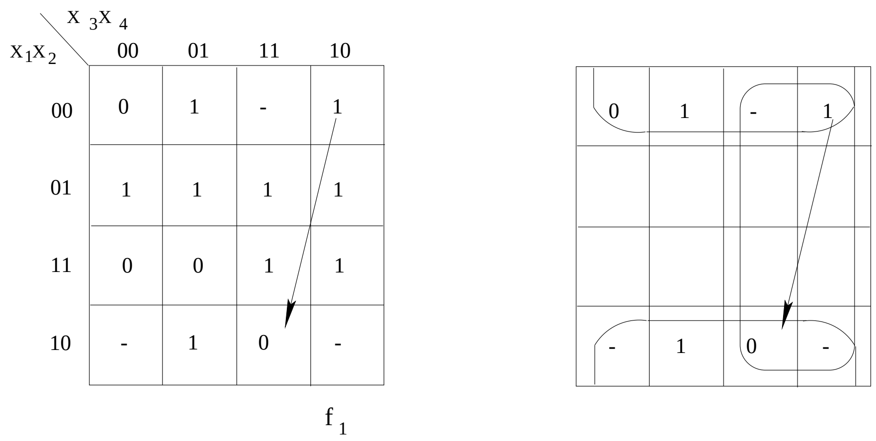


Figure 5.25: Fig31.31.xfig.ps

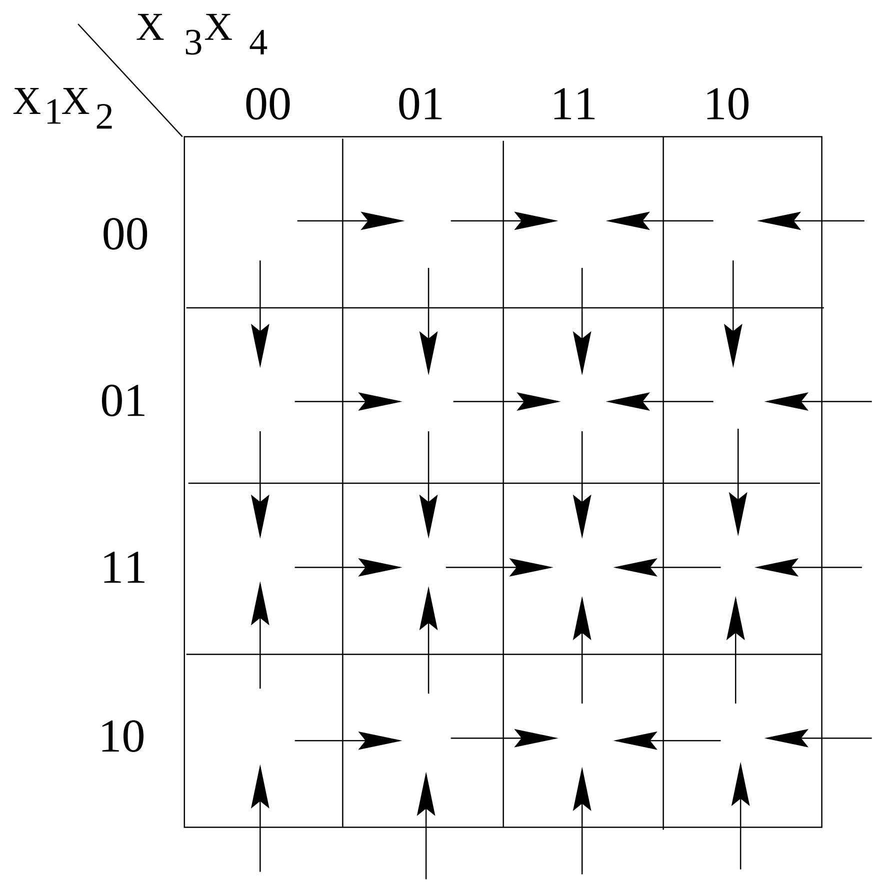


Figure 5.26: Fig31.32.xfig.ps

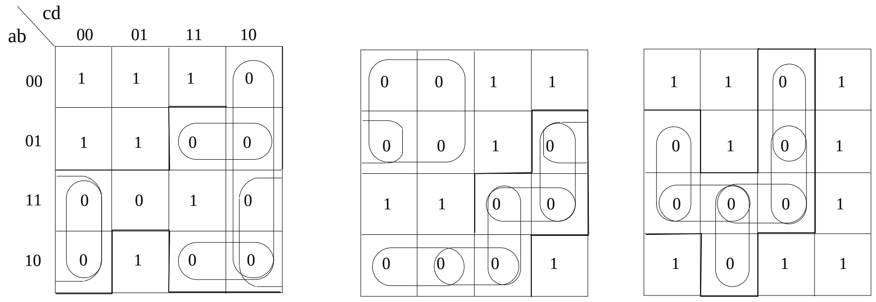


Figure 5.27: Fig31.33.xfig.ps

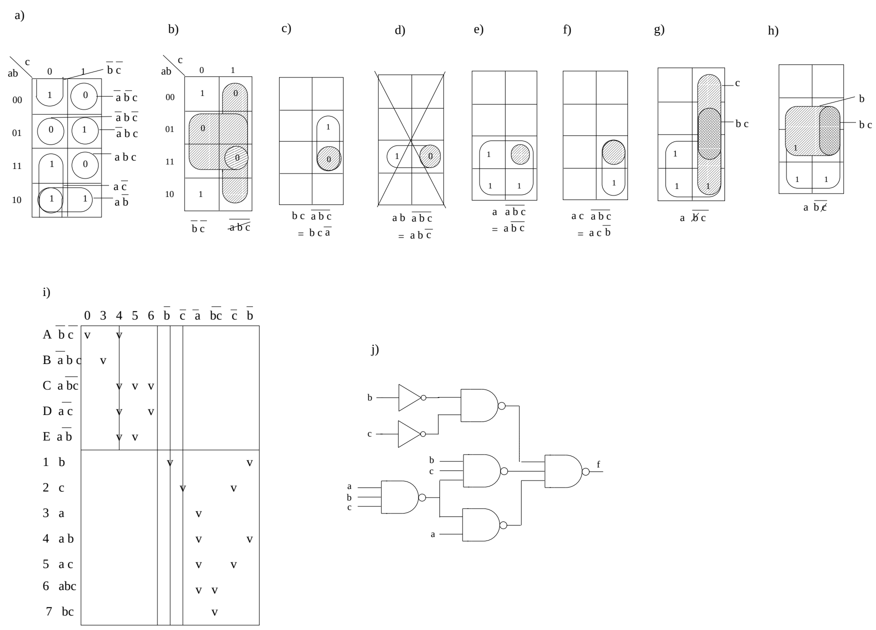


Figure 5.28: Fig31.34.xfig.ps

ab \ c	0	1
00	0	0
01	0	1
11	1	0
10	1	1

Figure 5.29: Fig31.35.xfig.ps

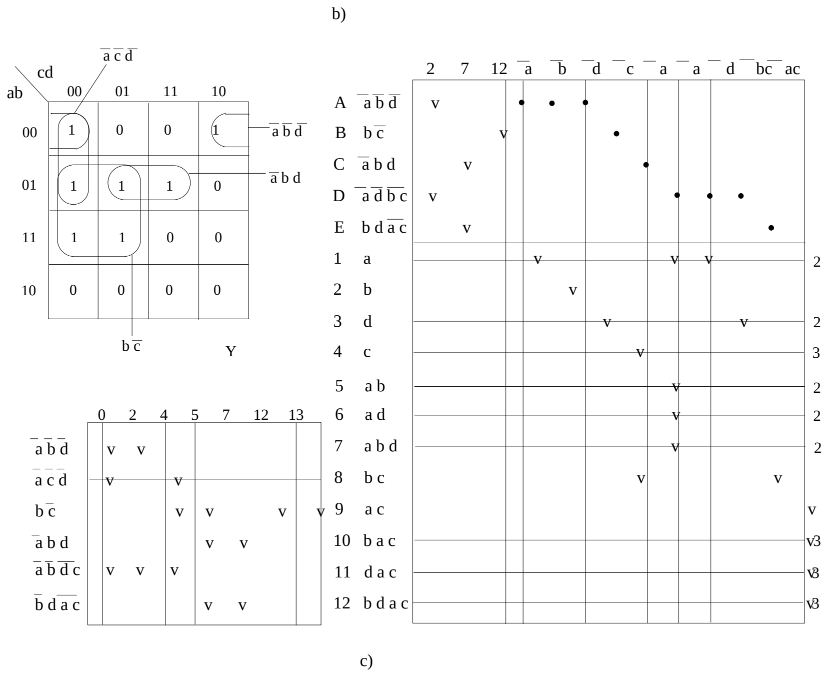


Figure 5.30: Fig31.36.xfig.ps

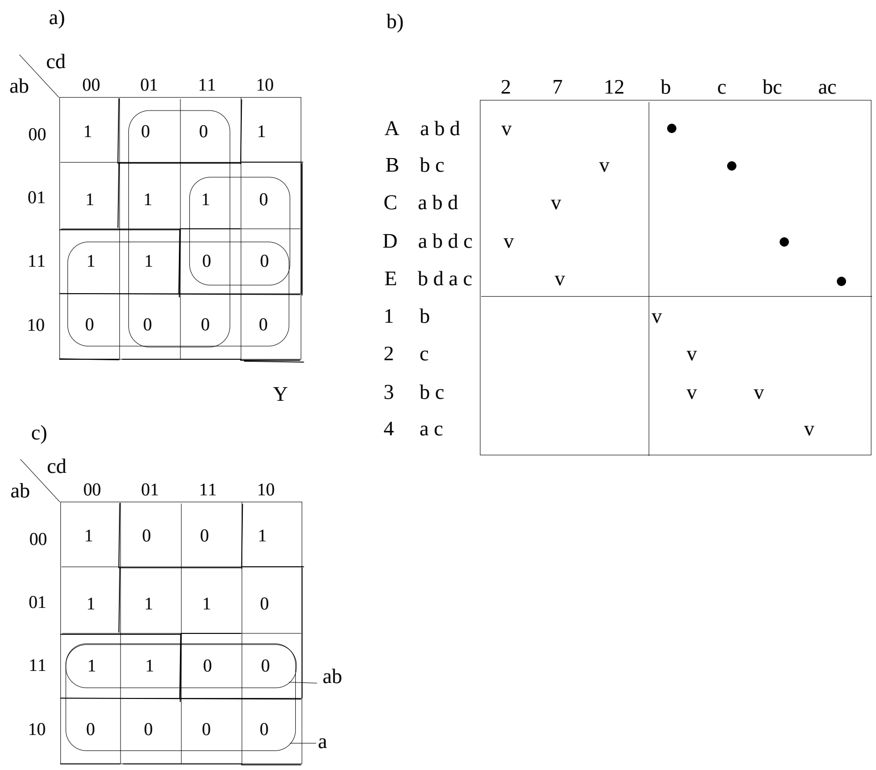


Figure 5.31: Fig31.37.xfig.ps

ab \ c de	000	001	011	010	110	111	101	100
00	0	-	-	-	-	0	1	1
01	0	1	-	-	-	-	1	-
11	0	1	1	-	1	0	-	0
10	0	-	-	-	1	1	-	1

Figure 5.32: Fig31.38.xfig.ps

		01001	11001	11011	11110	10110	10111	00101	00100	01101	10100
A	e cd ac	v	v	v				v		v	
	de c			v							
B	dce			v	v	v					
C	cde				v	v					
	cdea							v		v	
D	c de ab ae					v		v	v	v	v
E	be acd	v	v	v						v	
	bde ac			v							
F	bd ace			v	v						
G	bcd ae				v						
	bc de a										
	bce ad									v	
	bc a									v	
	abe cd		v	v							
	ab de c			v							
H	abd ce			v	v						
I	ab cd e				v						
	abce d										
	aec		v	v							
J	ade bc			v			v				
K	ad bce			v	v	v	v				
L	acd be				v	v	v				
M	acde b						v				
	ac be					v					v

Figure 5.33: Fig31.39.xfig.ps

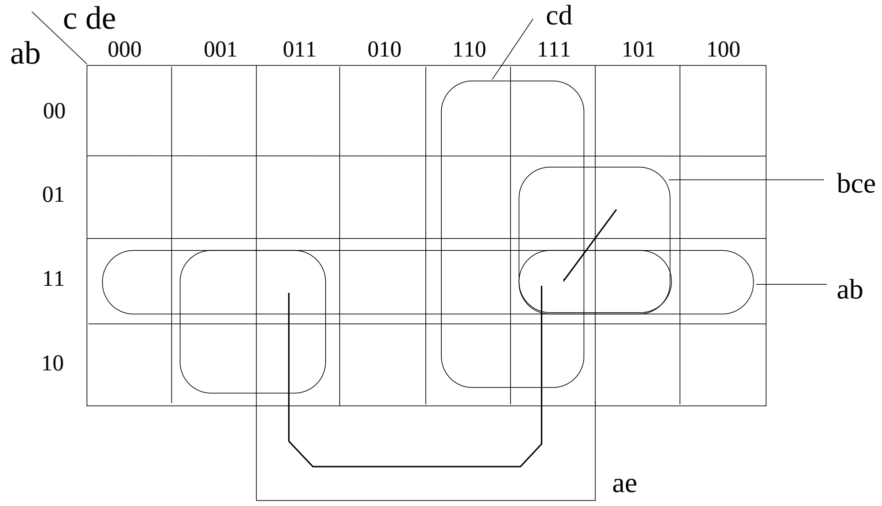


Figure 5.34: Fig31.40.xfig.ps

x_1x_2 \ x_3	0	1
00	1	1
01	0	1
11	1	0
10	0	0

Figure 5.35: Fig31.41.xfig.ps

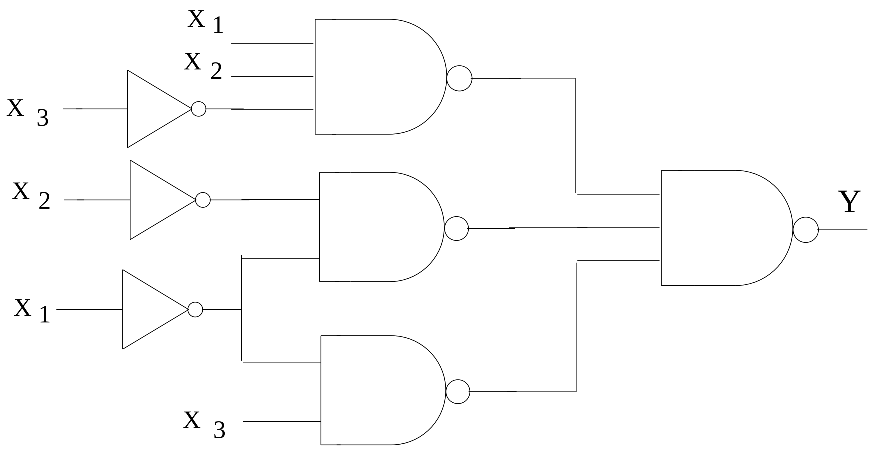


Figure 5.36: Fig31.42.xfig.ps

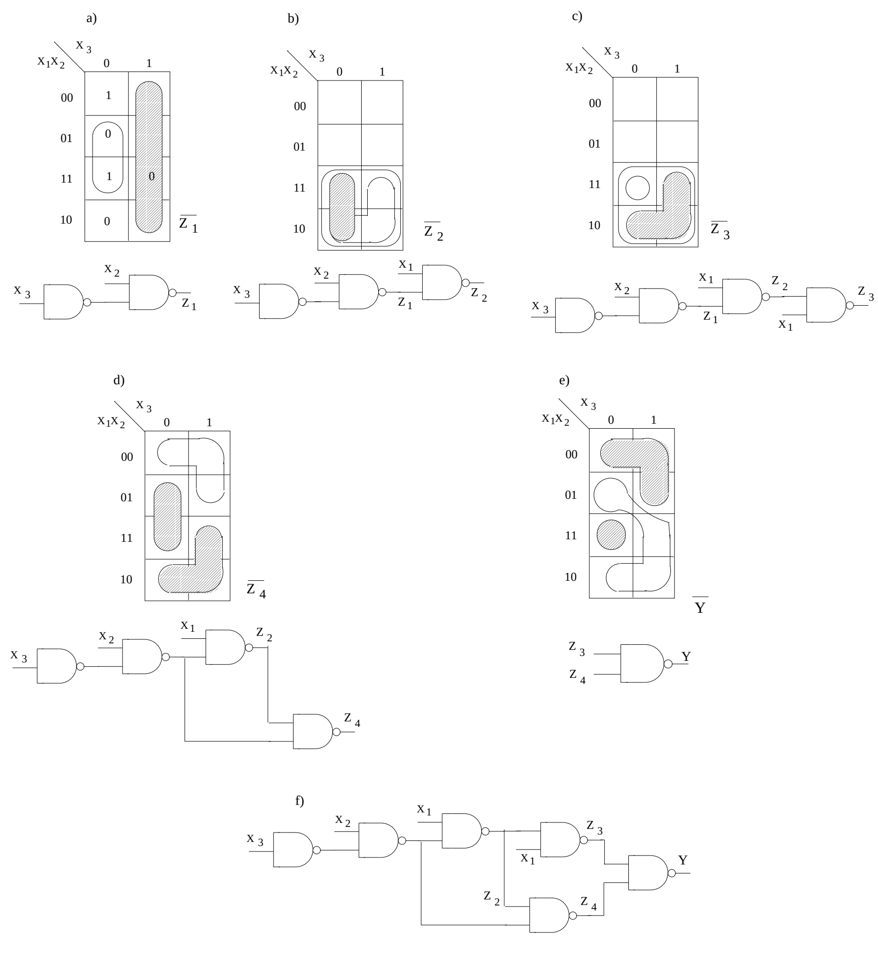


Figure 5.37: Fig31.43.xfig.ps

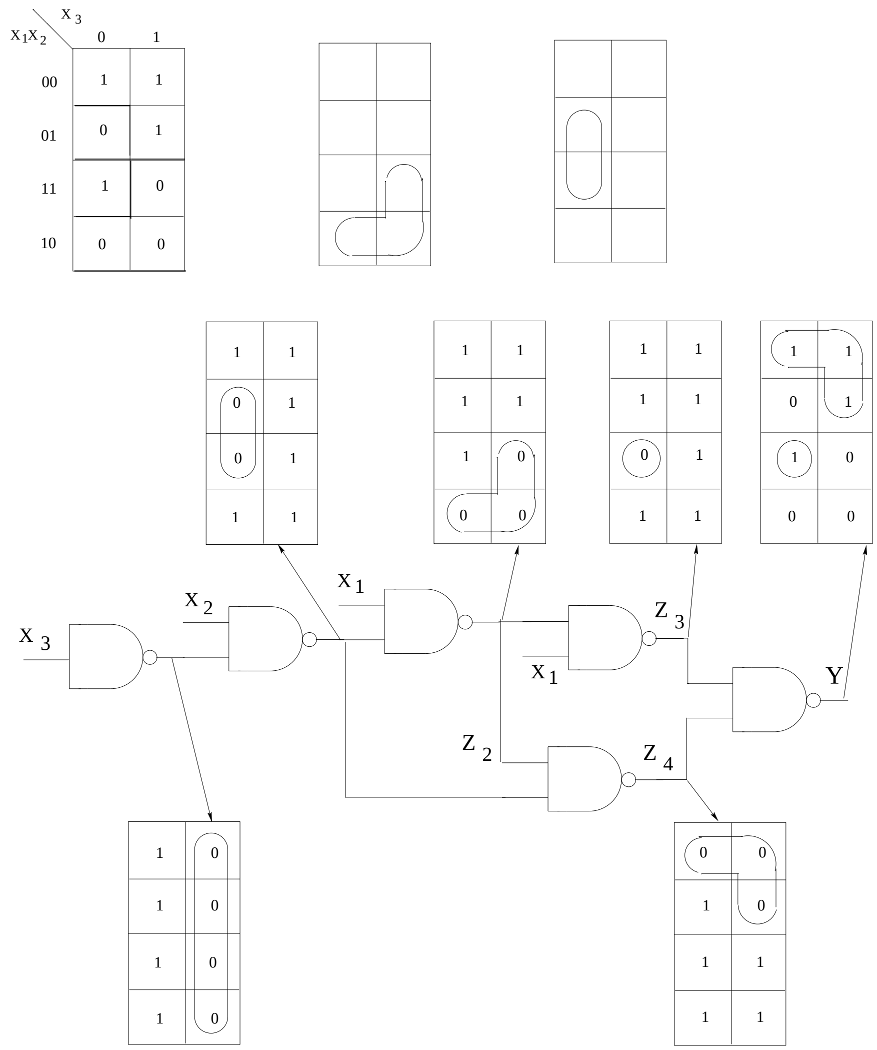


Figure 5.38: Fig31.44.xfig.ps

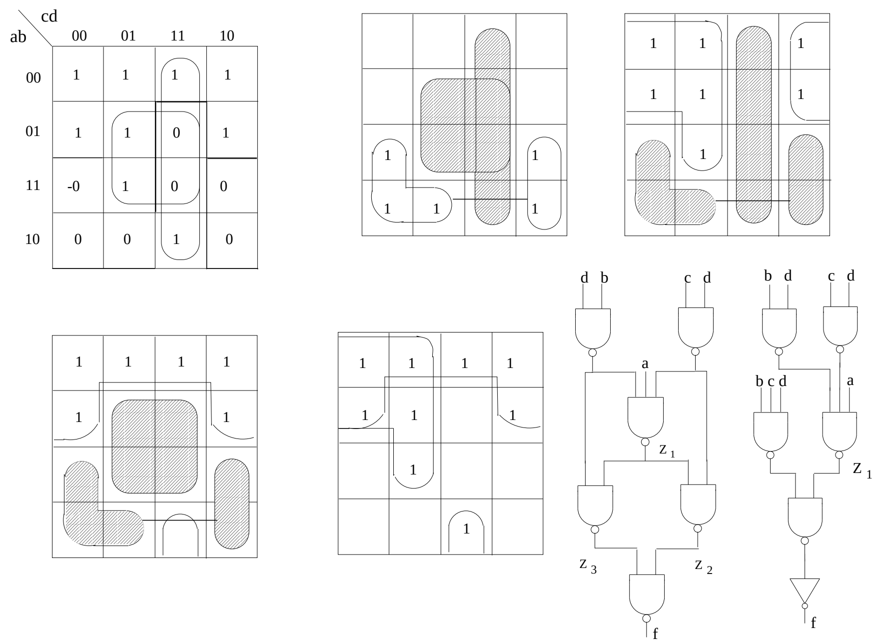


Figure 5.39: Fig31.45.xfig.ps

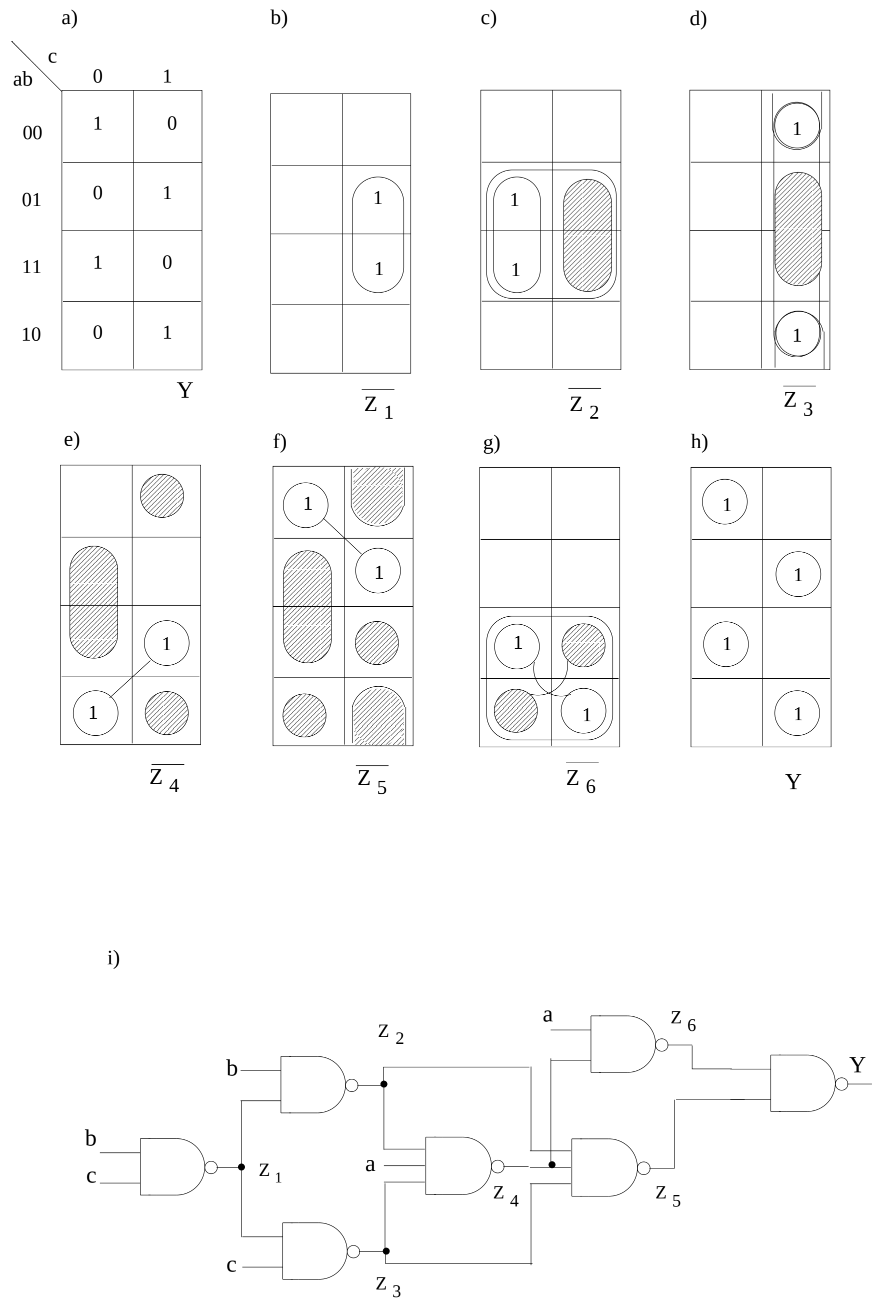


Figure 5.40: Fig31.46.xfig.ps

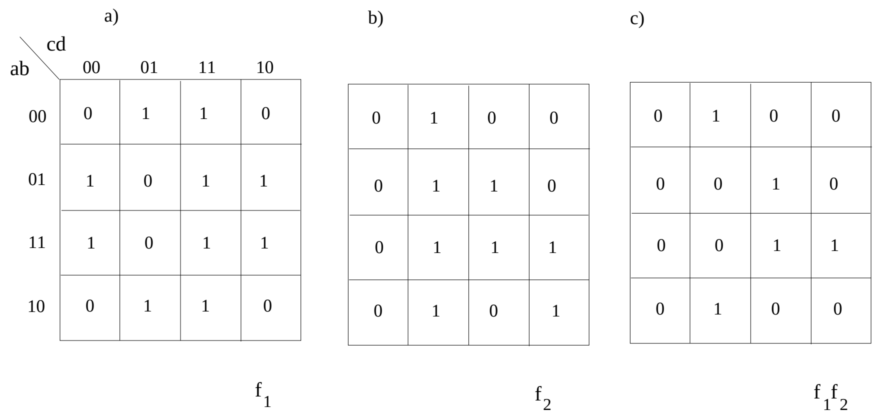


Figure 5.41: Fig31.47.xfig.ps

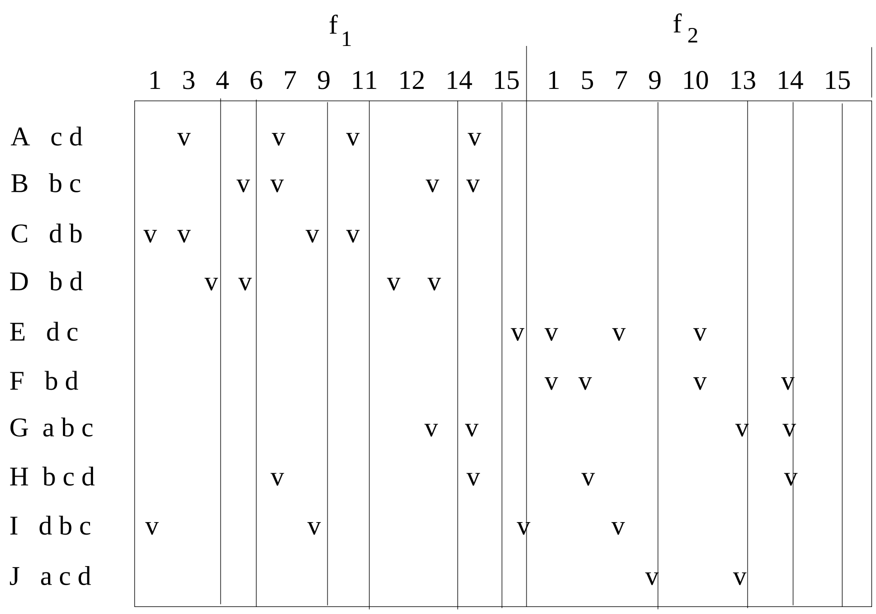


Figure 5.42: Fig31.48.xfig.ps

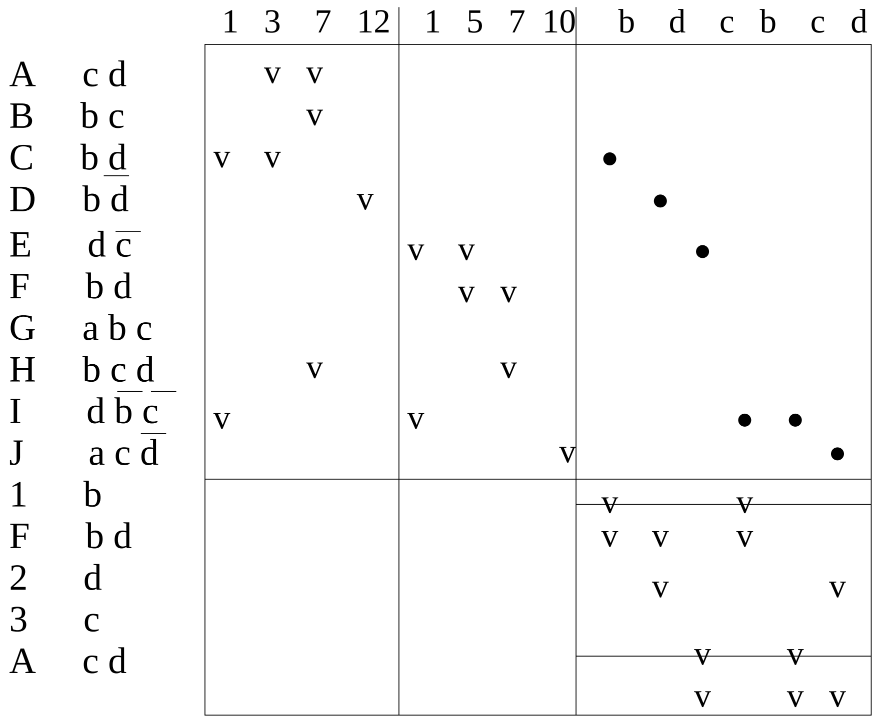


Figure 5.43: Fig31.49.xfig.ps

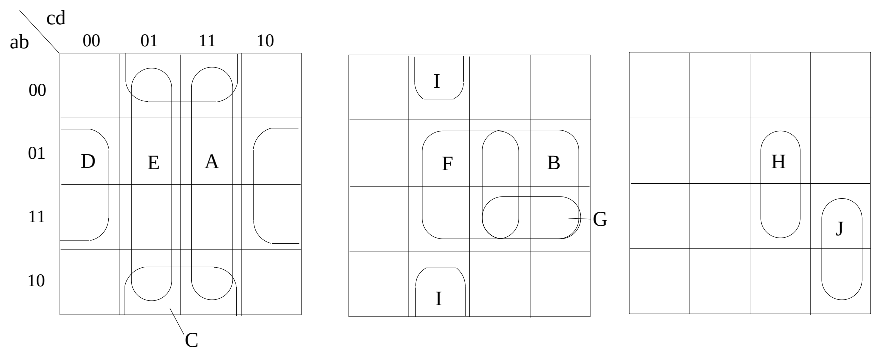


Figure 5.44: Fig31.50.xfig.ps

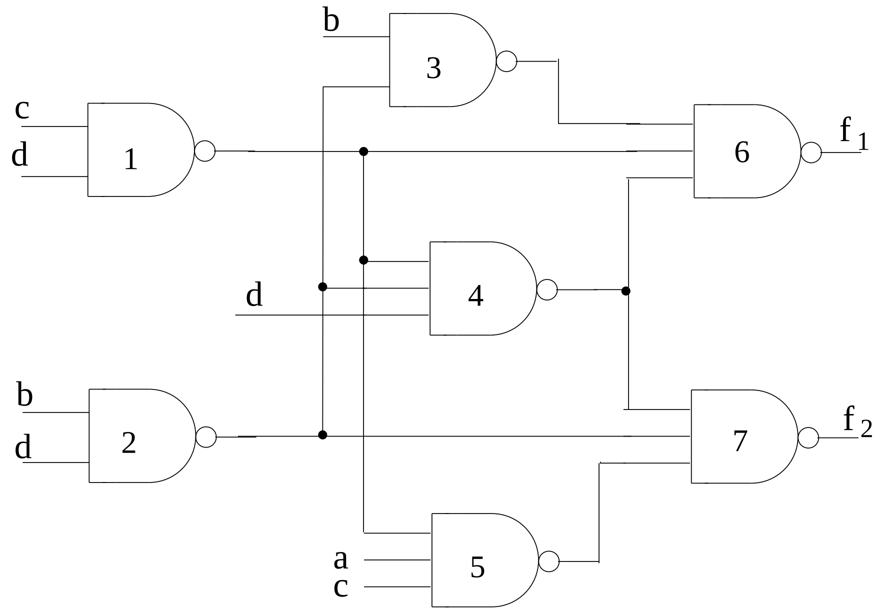


Figure 5.45: Fig31.51.xfig.ps

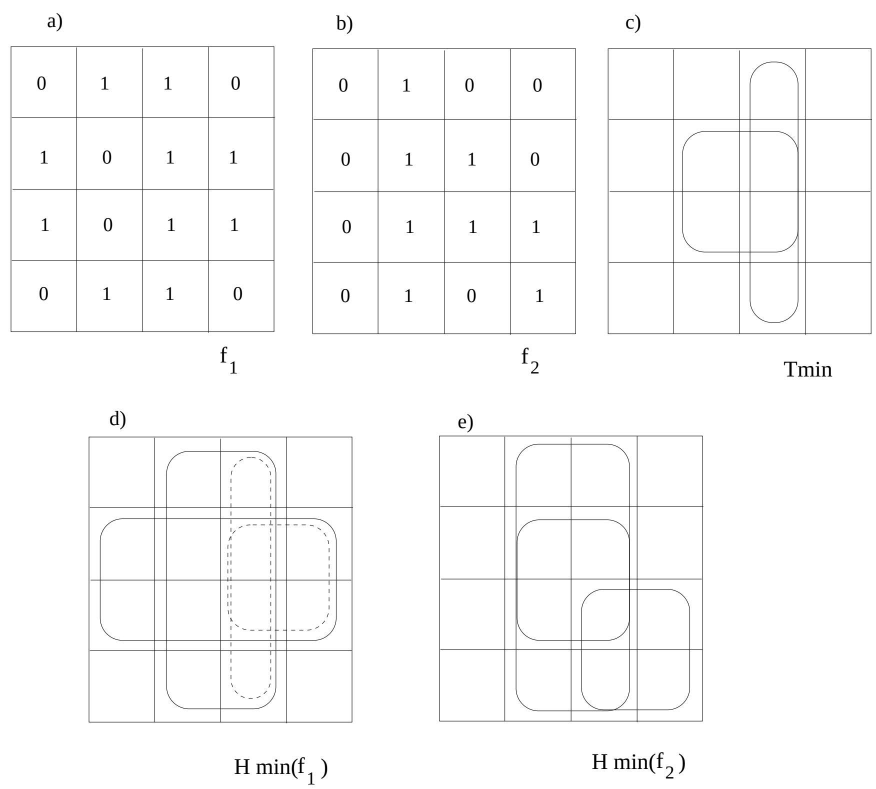


Figure 5.46: Fig31.52.xfig.ps

	a	b	c	d	f_1	f_2	f_3	f_4	f_5
0	0	0	0	0	1	1	0	0	0
1	0	0	0	1	1	0	1	0	0
2	0	0	1	0	0	1	1	0	0
3	0	0	1	1	1	0	0	1	0
4	0	1	0	0	0	1	0	1	0
5	0	1	0	1	0	0	1	1	0
6	0	1	1	0	1	0	0	0	1
7	0	1	1	1	0	1	0	0	1
8	1	0	0	0	0	0	1	0	1
9	1	0	0	1	0	0	0	1	1

Figure 5.47: Fig31.53.xfig.ps

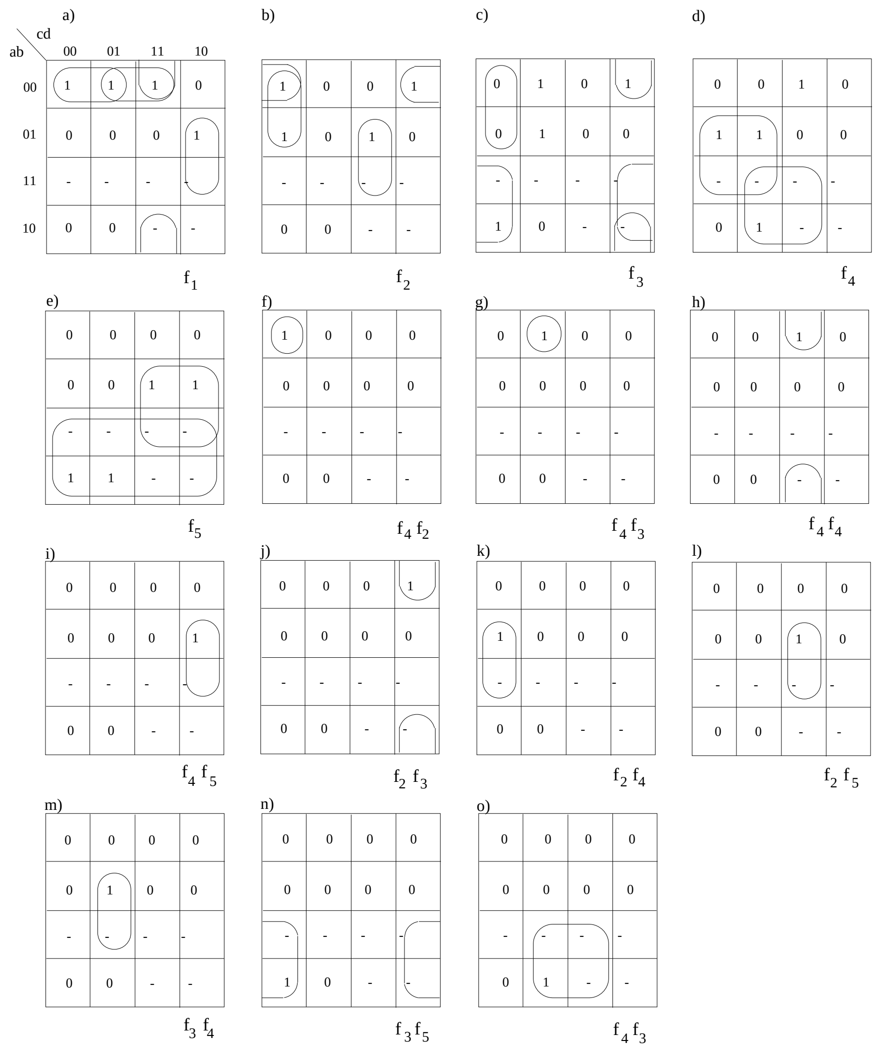


Figure 5.48: Fig31.54.xfig.ps

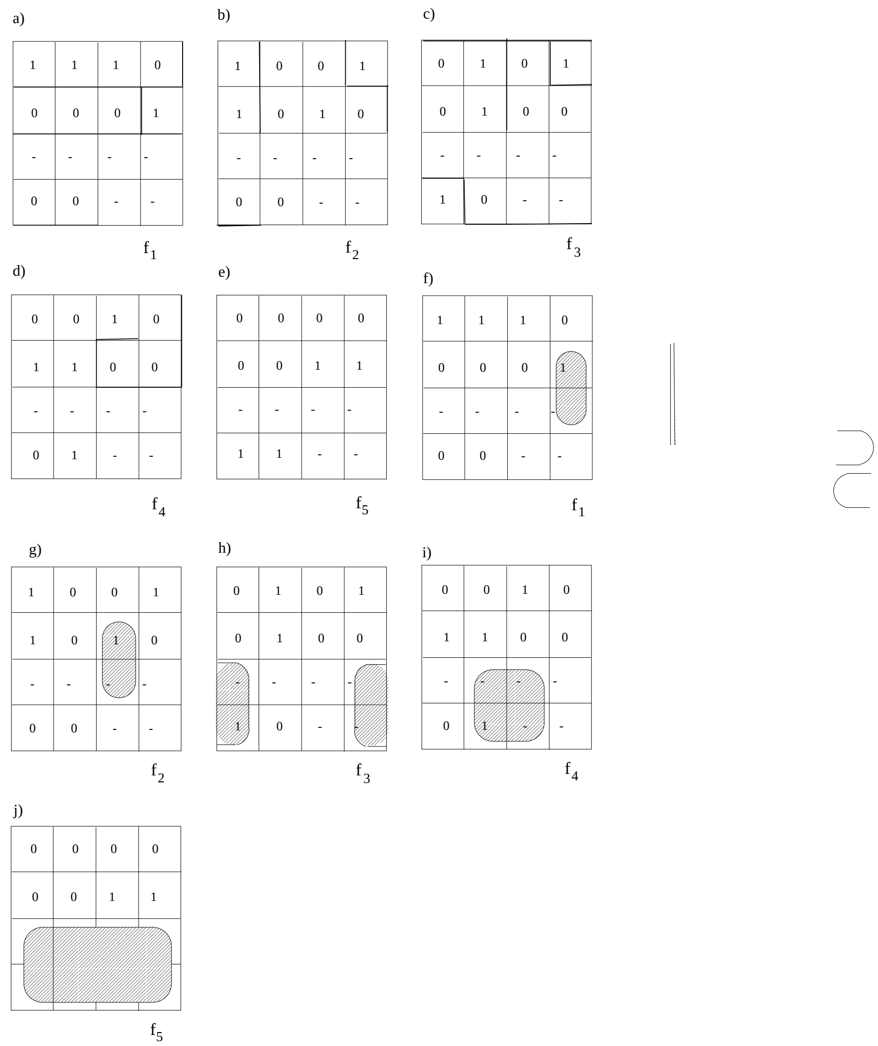


Figure 5.49: Fig31.55.xfig.ps

		f_1			f_2			f_3			f_4		f_5			
		0	1	3	0	2	4	1	2	5	4	5	6	7	bc	
A	a b c	v	v													
B	a b d		v	v												
C	c d b			v												*
D	a b c d	v			v											
E	a b c d		v					v								
F	a b d				v	v										
G	a d b c				v	v	v									
H	a c d							v		v					●	
I	b c d								v							
J	b c										v	v				
K	b c												v	v		
L	b c d						v				v					
O	b c d								v			v				
M	b c d													v		*
N	b c d												v			*
K	b c														v	

Figure 5.50: Fig31.56.xfig.ps

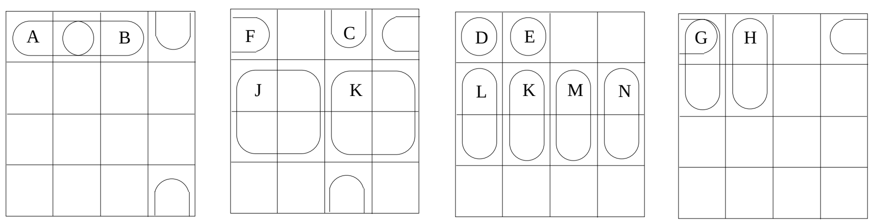


Figure 5.51: Fig31.57.xfig.ps

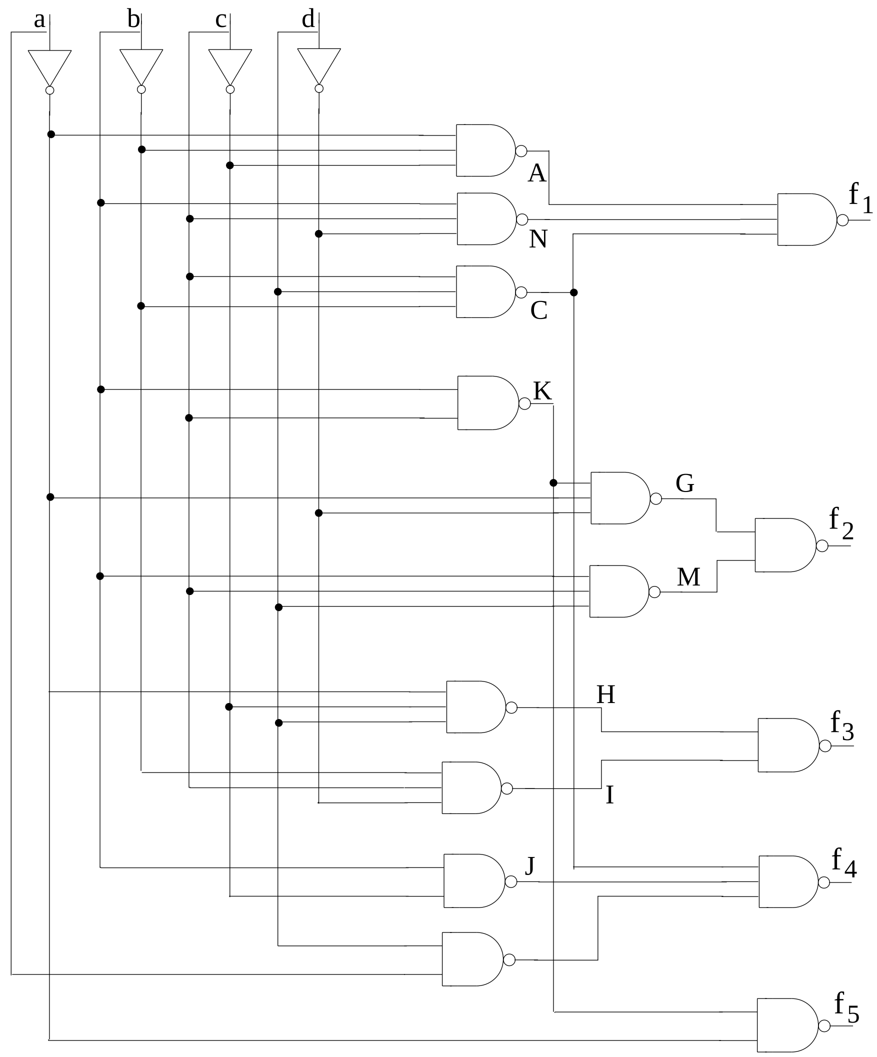


Figure 5.52: Fig31.58.xfig.ps

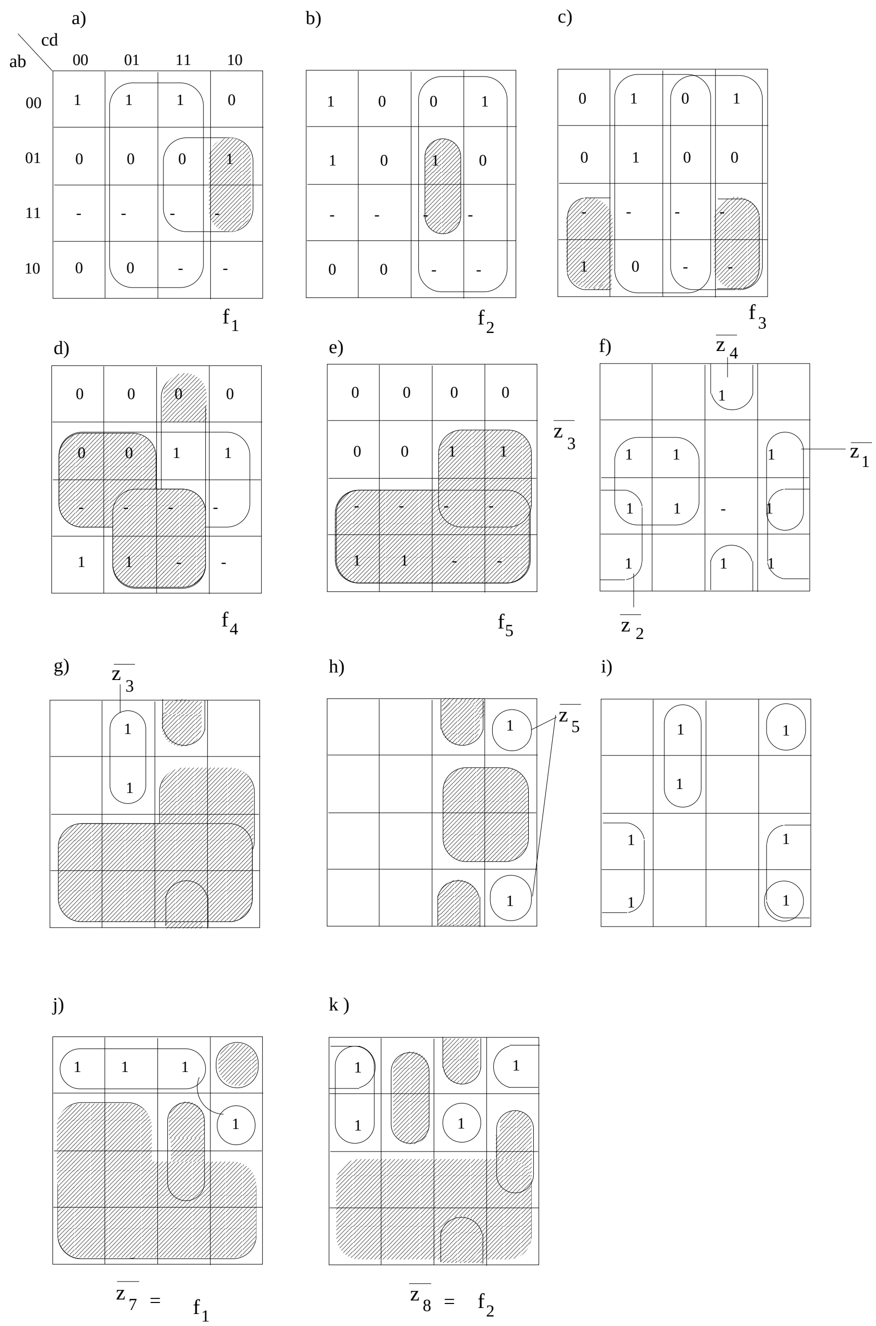


Figure 5.53: Fig31.59.xfig.ps

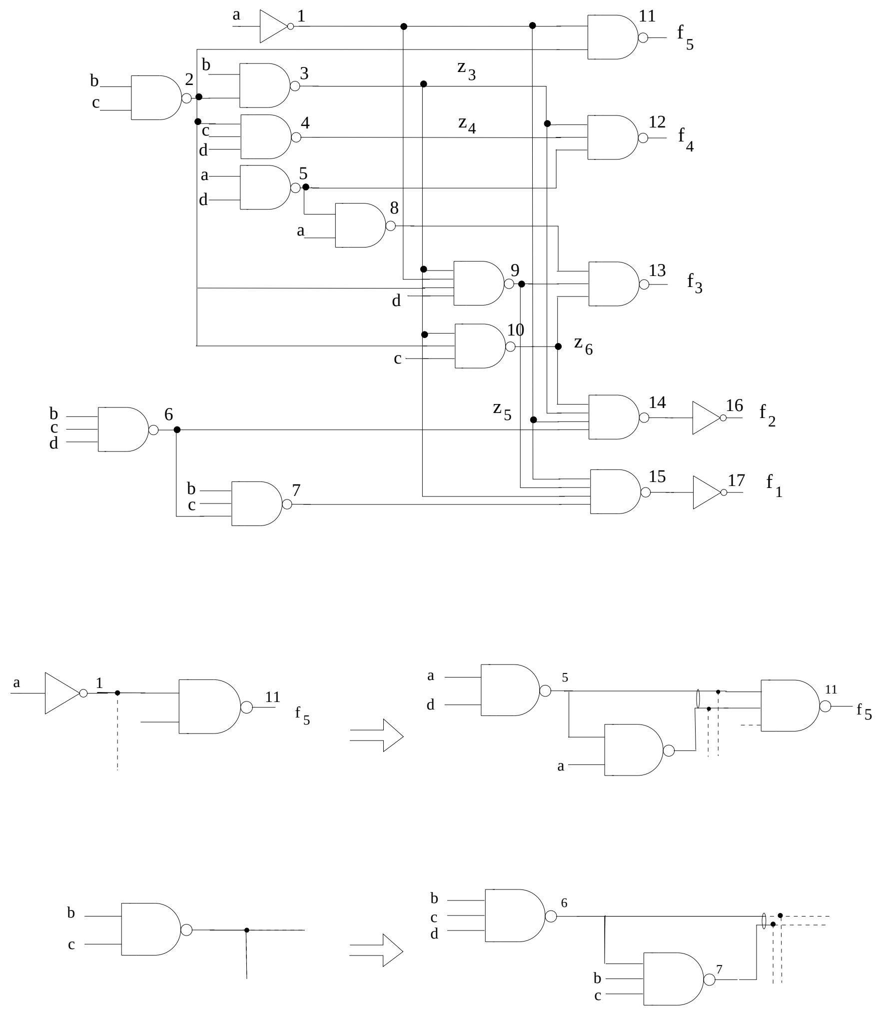


Figure 5.54: Fig31.60.xfig.ps

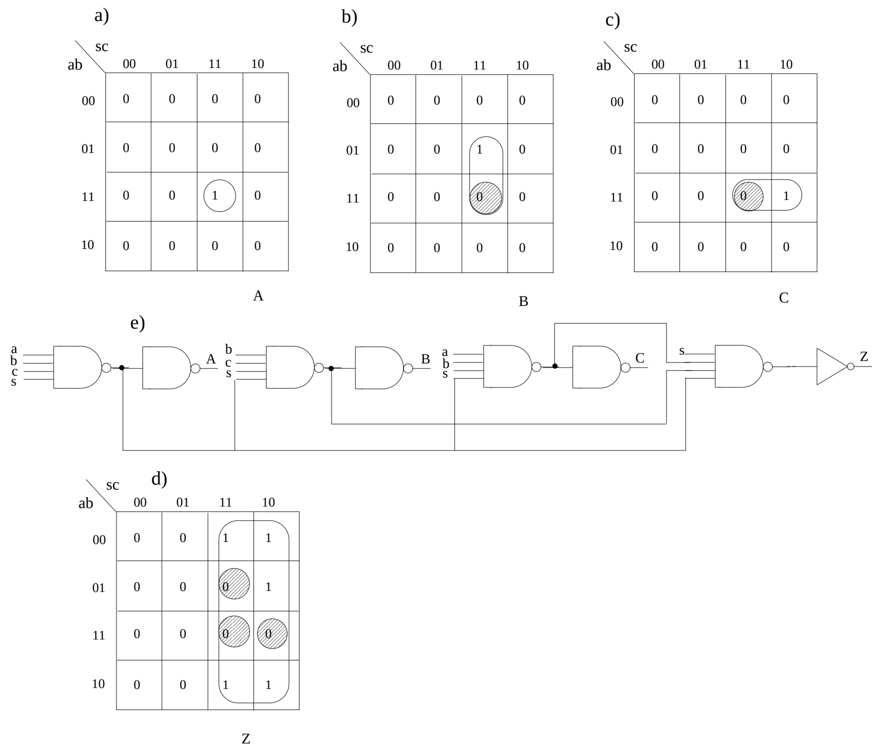


Figure 5.55: Fig31.61.xfig.ps

Chapter 6

Combinational Logic Design

6.1 Boolean Functions Representations in Logic Design and Tools

A logic function can be represented in different ways. Both **behavioral** (also called functional) representations and structural representations are used in logic designs. The representations can be used at all different levels of abstraction; architecture level, register-transfer level, and logic level (see next chapter).

Boolean functions are characterized usually as:

1. **truth tables**, that you know from previous chapters,
2. **arrays of cubes**, that will be discussed in detail in Chapter **??**.
3. **decision diagrams**.
4. **Labeled Rough Partitions**.
5. **Spectral Representations**.
6. **Netlists**.
7. **Expressions**.
8. **Waveforms**.

Representations can be **canonical**, it means unique for a function, or **non-canonical**. Truth tables and Binary Decision Diagrams are examples of canonical representations for a specified order of input variables. Cube representations are not canonical, but can be made canonical under certain assumptions. For instance, the set of all prime implicants of a completely specified function is a canonical representation of this function. Thus, the respective set of cubes is also canonical.

Canonical representations are important because they simplify, among others, the comparison of two functions used by the verification and synthesis algorithms.

Good understanding of cube calculus and decision diagrams is necessary to create and program efficient algorithms for logic design, test generation and formal verification.

The main goal of using a specific representation of a Boolean function is to facilitate certain operations on the function and to compress the information provided by the function to save time and space when processing the function.

6.2 Truth Tables and Karnaugh Maps

As illustrated in Chapter **??**, a truth table for a logic function is a listing of input combinations and their corresponding output values. Truth tables are suitable to present functions with small numbers of inputs, for instance single cells of **iterative circuits**. Truth tables can be easily, and in several different ways, specified in VHDL.

Karnaugh Maps are two-dimensional visual representations of truth tables, in which input variables are partitioned into vertical and horizontal variables, and all value combinations of input variables are expressed in Gray codes to allow the geometrically adjacent cells to become combinable using the law $AB + A\overline{B} = A$. For functions with large numbers of inputs, the corresponding truth tables or Karnaugh maps are too large. Kmaps are not used as data formats in tools and their meaning is nowadays only didactic. Marquand Charts, Harvard Decomposition Charts and many similar representations have been also invented. Most of them are similar to Kmaps, but have binary natural codes for rows and columns, rather than Gray codes. Various types of maps have some specific adavantages, for instance are easier for decomposition or finding unateness or symmetry. In my opinion, it is better if the beginner will stick to Kmaps with which he is already well familiar.

6.3 Multi-Level Design Using Multiplexers

Multi-level circuits using multiplexers can be built as a direct application of Shannon expansion:

$$f = af_a + \overline{a}f_{\overline{a}}$$

where f_a and $f_{\overline{a}}$ are cofactors of function f with respect to variable a.

Variable a is assigned to the multiplexer control input, cofactor f_a to the input 1 and cofactor $f_{\overline{a}}$ to the input 0 of the multiplexer. Each cofactor can be again Shannon expanded giving as a result multi-level multiplexer circuit.

Multiplexers can be used to implement any Boolean function but are particularly suitable for symmetric functions, and other special classes of functions. As an example we will show how the functions of two variables can be implemented by means of a multiplexer. The multiplexer has two data inputs $D_1 D_0$, one select input s and one output z. Now, suppose we want to implement the EXCLUSIVE-OR operation $A \oplus B$, i.e., $z = A\overline{B} + \overline{A}B$. This can be accomplished by connecting variable A to select input s and variables B and $\overline{B}$ to D_0 and D_1, respectively.

In general, to implement an n-variable function, we assign $n-1$ variables to the select inputs—one variable to each such input. The last variable and the constant 0 and 1 are assigned to the data inputs in such a way that together with the select-input variables they will yield the required function. Such an implementation is usually possible when at least one variable is available in both its complemented as well as uncomplemented forms; otherwise a larger multiplexer may be required. Implementations of functions of five or more variables are usually accomplished by means of a multilevel arrangement of several smaller standard multiplexers.

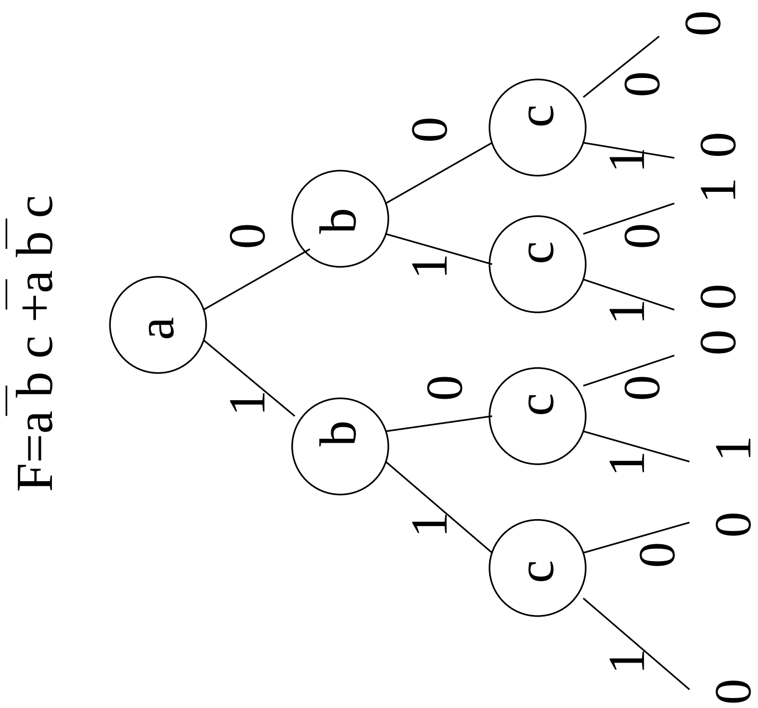


Figure 6.1: BDD Decomposition

A multiplexer can be viewed as an electronic switch that connect one out of 2 inputs to the output. Thus any Boolean function can be realized by multi-level tree or DAG of multiplexers, each using Shannon expansion $F = af_a + \bar{a}f_{\bar{a}}$. The structure of the function which is realized by multiplexers is like BDD, as shown in Fig. **??**. At every level, x_i is used as a selector to multiplexers.

6.4 Fundamental Expansions

Shannon Expansion
Shannon's expansion theorem is one of the fundamental expansion methods which states that any switching function $f(x_1, x_2, \ldots, x_n)$ can be expressed as

$$f(x_1, x_2, \ldots, x_n) = x_1 \cdot f(1, x_2, \ldots, x_n) + \overline{x_1} \cdot f(0, x_2, \ldots, x_n)$$

$$f(x_1, x_2, \ldots, x_n) = [x_1 + f(0, x_2, \ldots, x_n)] \cdot [\overline{x_1} + f(1, x_2, \ldots, x_n)]$$

Shannon expansion can be used to create Binary Decision Diagram and get a set of disjoint prime implicants. It also can be used to implement circuits with multiplexers or AND/OR gates.
Typical applications of Shannon expansion include:

- BDD construction
- create circuit with MUX
- prime implicants generation and functional decomposition.

Davio Expansion
Davio expansion has two forms. One is positive Davio expansion, and the other is negative Davio expansion.

- Positive Davio Expansion
  A Boolean function f can be expressed as:

$$\begin{aligned} f &= af_a \oplus \overline{a}f_{\overline{a}} \\ &= af_a \oplus (1 \oplus a)f_{\overline{a}} \\ &= af_a \oplus f_{\overline{a}} \oplus af_{\overline{a}} \\ &= f_{\overline{a}} \oplus a[f_a \oplus f_{\overline{a}}] \end{aligned}$$

- Negative Davio Expansion
  A Boolean function f can also be expressed as:

$$\begin{aligned} f &= af_a \oplus \overline{a}f_{\overline{a}} \\ &= (1 \oplus \overline{a})f_a \oplus \overline{a}f_{\overline{a}} \\ &= f_a \oplus \overline{a}f_a \oplus \overline{a}f_{\overline{a}} \\ &= f_a \oplus \overline{a}[f_a \oplus f_{\overline{a}}] \end{aligned}$$

Davio expansion can be used to create Functional Decision Diagram and implement circuits with AND/EXOR gates. The ESOP realization may also use Davio expansion.

6.5 Binary Decision Diagrams.

Decision Diagrams represent a function by a **Directed Acyclic Graph** (DAG). In case of the most often used **Binary Decision Diagrams**, BDDs, the nodes of the graph correspond to **Shannon Expansions** (realized by multiplexer gates), controlled by the variable a associated to this node: $F = a \cdot F_a + \overline{a} \cdot F_{\overline{a}}$. **Shared BDDs** are those in which equivalent nodes of several output functions are shared. **Equivalent nodes** g and h are those whose cofactor functions are mutually equal: $g_a = h_a$ and $g_{\overline{a}} = h_{\overline{a}}$.

Ordered BDDs are those in which the order of nodes in every branch from the root is the same. A diagram is obtained from arbitrary function specifications, such as arrays of cubes, factored forms, expressions or netlists, by recursive application of Shannon expansion to the function, next its two cofactors, four cofactors of its two cofactors, and so on, and combining any **isomorphic** (logically equivalent) nodes. The function corresponds to the root of the diagram. The are two terminal nodes of a Binary Decision Diagram, 0 and 1, corresponding to Boolean false and true. If two successor nodes of a node S_i point to the same node, then node S_i can be removed from the DAG. There are other similar reduction transformations in diagrams more general than BDDs. Decision Diagrams with such reductions are called **Reduced Ordered Decision Diagrams.**

In addition, negated (inverted) edges are introduced in BDDs. Such edges describe negation of its argument function. In **Kronecker Decision Diagrams** (KDDs) three types of expansion nodes exist: **Shannon nodes** (realizing function $f = a \cdot f_a + \overline{a} \cdot f_{\overline{a}}$), **Positive Davio nodes** (realizing function $f = a \cdot (f_a \oplus f_{\overline{a}}) \oplus f_{\overline{a}}$), and **Negative Davio nodes** (realizing function $f = \overline{a} \cdot (f_a \oplus f_{\overline{a}}) \oplus f_a$), which correspond to all three possible canonical expansions of Boolean functions. Other known decision diagrams include **Zero-Suppressed Binary Decision Diagrams** (ZSBDDs), **Moment Diagrams** and other, used primarily in verification or technology mapping. **Multi-valued Decision Diagrams** have more than two terminal nodes and multi-valued branchings with more than two successors of a node. These diagrams allow to describe and verify some circuits (such as large multipliers) that are too large to be described by standard BDDs. Some diagrams may be also better for logic synthesis to certain technologies.

There are two types of Decision Diagrams, **canonical diagrams** are used for function representation and tautology checking, thus used in various design and verification programs as internal function representation. ZSBDDs and KDDs are examples of canonical representations. DDs can be also adapted to represent state machines, by describing the (logic) characteristic function of a machine described as a relation. **Non-canonical decision diagrams**, such as for instance free Pseudo-Kronecker Decision Diagrams where any types of Shannon and Davio expansions can be mixed in levels and all orders of variables are allowed in branches, are used in synthesis and technology mapping [**?**, **?**].

6.6 Labeled Rough Partitions

to be added

6.7 Spectral Representations

to be added

6.8 Waveforms

Waveforms are normally used for viewing simulation results and specifying stimulus (input) to the simulator. Recently they are also being increasingly used as one of possible input data design specifications, especially for designing asynchronous circuits and circuits that cooperate with buses. Figure **??** shows the waveforms of a full adder from Table **??**. This full adder has two data inputs A and B, a carry-in input C_{in}, and two outputs Sum and carry-out C_{out}.

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 6.1: Truth table of full adder

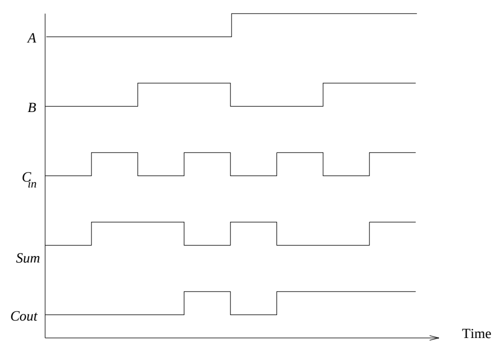


Figure 6.2: Waveforms for the full adder

6.9 Netlists or Logic Gate Networks

Standard design uses basic logic gates: AND, OR, NOT, NAND and NOR. Recently EXOR and XNOR gates become incorporated into tools and designs, and several algorithms for logic design that better take into account such gates together with other gates have been created. For certain designs, such as arithmetic data-path operations, EXOR based logic can decrease area, improve speed and power consumption, and improve significantly the testability. Such circuits are thus used in design for test. Other gate models include designing with EPLDs which realize AND-OR and OR-AND architectures, corresponding to Sum-of-Products and Product-of-Sums expressions, respectively. In **standard cell** technologies a more powerful libraries of cells are used, such as AND-OR-INVERT, or OR-AND-INVERT gates. In FPGAs different combinations of multiplexers are used, cells that use **Positive Davio** (AND-EXOR) and **Negative Davio** (NOT-AND-EXOR) expansion gates, or similar cells with a small number of inputs and outputs. The lookup-table model, used for instance in Xilinx FPGAs, assumes that arbitrary function of some small number of variables (3,4, or 5) and small number of outputs, usually 1 or 2, can be realized in a programmable cell. There are also

universal modules, that allow to realize all functions of certain type; the examples can be the cells used in Actel's or Atmel's FPGAs.

Several design optimization methodologies have been developed for each of these models.

6.10 Boolean Expressions

Boolean expressions use **logic functors** (operators) such as AND, OR, NOR, NOT, NAND, EXOR, MAJORITY and others, as well as literals, to specify the (multi-output) function. In order to specify netlists that correspond to DAGs, **intermediate variables** need to be introduced to the expressions. Every netlist or Decision Diagram can be specified by a set of Boolean Expressions with intermediate variables. Boolean Expressions can use **infix** (or standard), **prefix** (or Polish), or **postfix** (or Reverse Polish) notations. Most modern specification languages use infix notation for operators such as AND or OR. Operator AND can be sometimes omitted, as in standard algebraic notations. In conjunction with operators such as NAND, both infix and prefix notations are used. For instance (NAND a b c) in prefix and (a NAND b NAND c) in infix. Care is recommended when reading and writing such expressions in hardware description languages and input formats to tools. It is always good to use parentheses in case of doubt about operators' precedence. In some languages, arbitrary operators can be defined by users to be used in expressions on equal terms with well-known operators. Expressions can be created for SOP (Sum-of-Products), POS (Product-of-Sums), factorized SOPs and POSs, and other representations as a result of logic synthesis and optimization algorithms that will be described in section on Combinational Logic Design.

6.11 Combinational Logic Design

A combinational logic design involves a design of a combinational circuit. For instance, the design may assume two levels of logic. A Two-Level combinational logic circuit consists of two levels of logic gates. In the sum of products two-level form, the first (from the inputs) level of gates are AND gates and the second level of gates are OR gates. In the product of sums two-level form, the first level of gates are OR gates and the second level of gates are AND gates.

The reason of logic minimization is to improve the performance and decrease the cost by decreasing the area of the silicon, decreasing the number of components, increasing the speed of the circuit, making the circuit more testable, making it to use less power, or achieving any combination of the above design criteria. The optimization problem can be also specified to minimize certain weighted cost function under certain constraints; for instance, to decrease the delay under the constraint of not exceeding certain prespecified silicon area.

There are usually two logic minimization processes, the first is **generic** and **technology-independent**, the next one is technology-dependent minimization, called also **technology mapping**. This second stage may also take into account some topological or geometrical constraints of the device. These are used, for instance, in CPLD **device fitting** or designing for Fine Grain FPGAs, such as those from ATMEL or Motorola.

6.12 Two-Level SOP Combinational Logic

There are two types of programs for Two-Level logic minimization. **Exact programs** minimize the number of product implicants, the number of literals, or some weighted functions of the two. Heuristic or approximate programs attempt to minimize these **cost functions**, but do not give assurance of their minimum values. Usually, the exact programs generate all **prime implicants** or a subset of them, which can be proven to include at least one minimal solution. Prime implicant is a product of literals from which no literal can be removed. These implicants are then the largest products-of-literals groups of true minterms in a Karnaugh map that imply the function. Next, an exact program creates a **covering table** and finds its best covering with prime implicants. Such table has true minterms as columns and prime implicants (or their subset) as rows (or vice versa). If an implicant **covers** (includes) a minterm, it is denoted by an entry "1" at the intersection of the row corresponding to the implicant and the column corresponding to the minterm. The **exact solution** (minimum, product-wise) is the subset of rows that cover (have minterms in) all columns and that has the minimum number of rows. The **minimum solution** (literal-wise) is the subset of rows that cover (have minterms in) all columns and has the minimum total number of literals in product terms (or that minimizes other similar cost function). Some algorithms use the concept of **essential prime implicants.** An essential prime is a prime that includes certain minterm that is covered only by this prime. A **redundant prime** is an implicant that covers only minterms covered by essential primes. Redundant primes can be thus disregarded. A **secondary essential prime** is a prime that becomes essential only after previous removal of some redundant primes.

The essential primes found are taken to the solution and the covered by them minterms are removed (using for instance sharp operator of cube calculus) from the function. This causes some primes to become **redundant** and results next in the origination of the secondary essential primes. This process of finding essential and secondary essential primes is iterated until no further minterms remain in the function - thus the exact solution is found without creating and solving the covering table. Functions with such property are called **non-cyclic functions.** Most of real life functions are either non-cyclic or have large non-cyclic cores, which is the reason of relative efficiency of such algorithms.

Approximate algorithms try to generate primes and find primes cover at the same time, thus they reshape the current cover by replacing some groups of primes with other primes, applying cube operations such as consensus.

Program Espresso from U.C. Berkeley [**?**] is a standard for two-level logic functions. The original program was next extended to handle multi-valued logic to allow PLAs with decoders, function generators, and preprocessing PLAs. Espresso-MV is useful for sequential machine design, especially state assignment and input/output encodings. Its ideas help also to develop new programs for these applications. Although heuristic version of Espresso does not guarantee the exact solution, it is close to minimum on several families of practical functions. Its variant, Espresso-Exact, finds the minimum solution, and program Espresso-Signature can find exact solutions even for functions with extremely large number of prime implicants. This is due to a smart algorithm that can find exact solutions even without enumerating all the prime implicants. There are some families of practical functions for which Espresso does not give good results and which are too large for Espresso-Exact or Espresso-Signature. Programs such as McBoole or other internally designed are combined with Espresso as user-called options in some commercial tools. Two-level minimization is used in many programs for multi-level synthesis, EPLD-based synthesis, and PLD/CPLD device fitting. These algorithms constitute the most advanced part of todays EDA theory and practice.

Topics close to Sum-of-Products minimization are Product-of-Sums design, three-level design (AND-OR-AND or OR-AND-OR), four-level design (AND-OR-AND-OR), etc., and other designs with few levels of AND-OR logic. Algorithms for their solution usually generate some kind of more complex implicants and solve the set covering or graph-coloring problems, either exactly or approximately. Such approaches are used for PLD, FPLA, and Complex PLD minimization.

Many of concepts and techniques used in two-level minimization, for instance essential implicants or covering tables, are also used in multi-level synthesis. Similar techniques are used in sequential synthesis: for example, a **binate covering** problem is used in both three-level design and state minimization, and **clique covering** is used in creating primes and also in several problems of sequential design.

Important stage of design is also finding the **minimum support** of a function, i.e., minimum set of input variables on which the given function depends. This can be used for logic realization with EPLDs (because of limited number of inputs) or in ROM-based function realization.

Many efficient "generic" combinatorial algorithms have been created for logic synthesis programs. They include: unate covering (used in SOP minimization, decomposition and minimum support problems), binate covering (used in state machine minimization and three-level design), **satisfiability** (Is F = 0? If yes, when?), tautology (is F = G?), and **graph coloring** (used in SOP minimization and functional decomposition). They can be all "re-used" for new applications by EDA tool designers.

6.13 Two-Level ESOP Minimization

Another two-level minimization problem is to find for a given function the Exclusive-Sum-of-Products expression with the minimum gate or literal cost. Several algorithms have been created for this task [**?**, **?**]. Tools for circuits that have few, usually three, levels and have levels of gates AND, EXOR and OR, in various orders, have been also recently designed [**?**, **?**].

6.14 Multilevel Factorization

A **multilevel combinational logic circuit** consists of more than two levels of logic gates. There exist several design styles that are used to obtain such circuits. The first method is called **factorization**.

Factorization is used to simplify Boolean expressions and reduces the number of gates needed to realize them. The following identities show factorization of Boolean expressions:

- $AB + AC = A(B + C)$
- $(A + B)(A + C) = A + BC$

An arbitrary combinational circuit can be converted to a logically equivalent circuit satisfying fan-in and fan-out constraints. Factorization is only one of procedures used in the generation of such circuits but it is the simpliest and the most universal one. Consider the function $F = x_1x_2x_3\overline{x_4} + x_1x_2\overline{x_3}x_4$. The two terms have the common factor x_1x_2 and hence F can be rewritten as $x_1x_2(x_3\overline{x_4} + \overline{x_3}x_4)$. The maximum gate fan-in has been reduced from four to two by factoring, while the number of levels of logic has increased from two to three.
The number of inputs of a gate(fan-in) is usually restricted to be within certain maximum value. In order to satisfy these constraints, it is frequently necessary to realize circuits by increasing the number of logic levels.
Another example, function:$F = abd + acd$. Assume the number of inputs of the logic gate is restricted to be 2. One has to realize this function using the method shown in Fig. **??**b, instead of using the method shown in Fig. **??**a. Here a, d are common factor of the two items of the function.

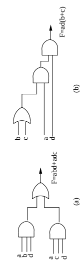


Figure 6.3: Factorization

Some factoring algorithms in addition to the two rules shown above use recursively other transformations as well, such as $a\overline{b} + a\overline{c} = a\ \overline{bc}$, and $ab\overline{cd} = ab\overline{abcd} = ab\overline{acd}$. Efficient factoring algorithms based on **kernels** and **rectangle covering** have been created [**?**]. They are used in many commercial tools, and are still being refined and improved to add special functionalities, for instance, improved testability. They are also being adapted for state assignment or reduced power consumption of the circuit. A high quality multi-level program, SIS, based mostly on factorization, can be obtained from U.C. Berkeley [**?**].
Factorization is used not only in context of circuit minimization, but also to allow representation of large circuits.

6.15 Functional Decomposition

The second group of multi-level synthesis methods are those based on **functional decomposition**. Such methods have originated from early research of Ashenhurst [**?**], Curtis [**?**], Roth and Karp, [**?**]. Functional decomposition methods do not assume any particular type of gates: rather they decompose a larger function to several smaller functions. Both functions can be specified as tables, arrays of cubes, BDDs, netlists, or any other above-mentioned logic representations, both binary and multi-valued. Functional decompositions can be separated to parallel and serial decompositions. **Parallel decomposition** decomposes multi-output function

$[F_1(a,b,c,...,z),\ F_2(a,b,c,...,z), ...F_{n-1}(a,b,c,...,z), F_n(a,b,c,...,z)]$

to several, usually two, multi-output functions, called blocks. For instance,

$[F_1(a,b,c,...,z),\ F_2(a,b,c,...,z),...\ F_{n-1}(a,b,c,...,z), F_n(a,b,c,...,z)]$

is decomposed into

$[F_{i_1}(a,....x), ...,\ F_{i_r}(c,....,z)]$ and $[F_{i_{r+1}}(a,b,....y), ...,\ F_{i_n}(c,...,x)]$,

such that each component function depends now on fewer variables (thus the support set of each is decreased, often dramatically). This problem is similar to the partitioning of a PLA into smaller PLAs.

Serial decomposition is described by a general formula:

$$F(A,B,C) \ = \ H(A \cup C, G(B \cup C)) \tag{6.1}$$

The set of variables $A \cup C$ is called the set of **free variables** (free set), set of variables $B \cup C$ is called the set of **bound variables**, and set of variables C is called the **shared set** of variables. If $C = \emptyset$, the decomposition is called **disjoint** (disjunctive), otherwise it is called **non-disjoint** (non-disjunctive). Function G is multi-output (or multi-valued) in Curtis decomposition, and single-output in classical Ashenhurst decomposition. Function G can be also multi-valued. Every function is non-disjoint decomposable, and many practical functions are also disjoint decomposable. The more the function is unspecified (more don't cares it has), the better are the decompositions and higher the chances of finding disjoint decompositions with small bound sets.

For example $Y = F(x_1, x_2, \ldots, x_n)$, is decomposable if it can be expressed as $F = H(A, C, G(B, C))$ where A, B and C are proper subsets of the set of input variables and H has fewer input variables than F shown in Fig. **??**. One of applications of decomposition is FPGA synthesis, especially for lookup-table based FPGAs such as Xilinx 3000 series of our interest.

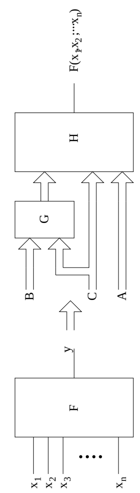


Figure 6.4: Decomposition Diagram

It was shown that practical benchmark functions are well-decomposable with small or empty shared sets, in contrast to randomly generated completely specified functions for which such decompositions do not exist. Most of the decomposition methods decompose recursively every block G and H, until they become non-decomposable. What is defined as "non-decomposable" depends on any particular realization technology, for instance any function with not more than 4 variables is treated as non-decomposable assuming the lookup-table model with 4 input variables in a block. In a different technology, the decomposition is conducted until every block becomes a simple gate realized in this technology, for instance a two-input AND gate, a MUX, or a three-input majority gate. Important problems in decomposition are: finding good bound sets and shared sets, and the **optimal encoding** of multi-valued functions G to binary vectors, in order to simplify concurrently both functions G and H.

Designing combinational logic using ROMs or RAMs is another important area. Because of the limited width of industrial chips, one has to create additional combinational address generator circuits, or other circuits that collaborate with the memory chip. Some of these techniques are quite similiar to decomposition.

Decomposition methods are not yet used in many industrial tools, but their importance is increasing. A universal functional decomposer program can be obtained from Portland State University [**?**].

6.16 Multi-level Synthesis Based on Decision Diagrams

The next group of multi-level synthesis methods is based on various kinds of decision diagrams. In the first phase the decision diagram such as a BDD, KFDD or ZSBDD is created, and its cost, for instance the number of nodes, is minimized. An important design problem is to find a good order of input variables, i.e. one that minimizes the number of nodes. For synthesis applications these diagrams are not necessarily ordered and canonical, because the more general diagrams (with less constraints imposed on them) can correspond to smaller of faster circuits. A universal Decision Diagram package PUMA that includes BDDs, KFDDs and ZSBDDs is available from Freiburg University [?]. Free BDDs, with various orders of variables in branches, or non-canonical Pseudo-Kronecker DDs with mixed types of expansions in levels are for instance used in technology mapping [?]. In the next phase, certain rules are used to simplify the nodes. Among these rules, the propagation of constants is commonly used. For instance, a Shannon node (a MUX) realizing a function $a \cdot 0 + \overline{a} \cdot b$ is simplified to AND gate $\overline{a} \cdot b$. Mux realizing $ab \oplus \overline{a}$ is simplified to OR gate $\overline{a} + b$. All rules are based on simple Boolean tautologies. For instance, a Positive Davio node realizing a function $a \cdot b \oplus a$ is simplified to an AND gate $a \cdot \overline{b}$. The heuristic algorithms that apply these transformations are iterative, recursive or rule-based. They are usually very fast.

For some technologies, the stage of generic multi-level optimization is followed by the technology-related phase (called **technology mapping**), in which techniques such as DAG-covering, or Tree Covering by Dynamic Programming are applied [?]. At this stage, the particular technological constraints of given target technology are taken into account by re-designing for particular cell libraries, fitting to certain fan-in of fan-out constraints, minimizing the number of cells to fit the IC device or to decrease the number of connections to fit the routing channel width.

6.17 Iterative Networks

6.17.1 Introduction to Iterative Networks

An **iterative network** consists of a number of identical cells interconnected in a regular manner. Some operations, like binary addition, are naturally realized with an iterative network because the same operation is performed on each pair of input bits.

The simplest form of iterative network consists of a linear array of combinational cells with signals between cells traveling in only one direction.

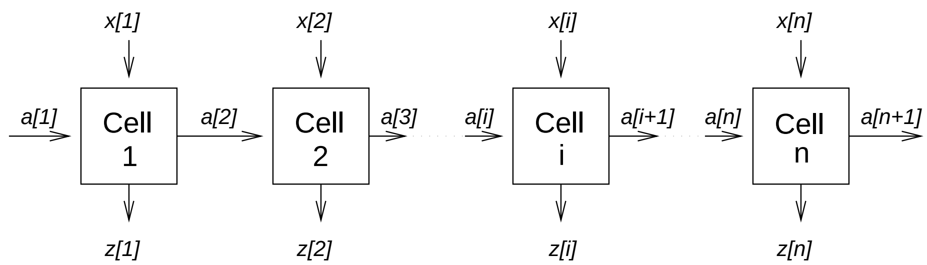


Figure 6.5: A Simple Iterative Network

As shown in Figure **??**, each cell is a combinational network with one or more **primary input(s)** ($x[i]$) and possibly one or more **primary output(s)** ($z[i]$). In addition, each cell has one or more **secondary input(s)** ($a[i]$) and one or more **secondary output(s)** ($a[i+1]$). The $a[i]$ leads carry information about the **"state" of the previous cell**. The primary inputs to the cells ($x[1], x[2], \ldots, x[n]$) are applied **in parallel**, that is, they are applied at the same time. The $a[i]$ signals then propagate down the line of cells. Since the network is combinational, the time required for the network to reach a stable state condition is determined only by the **delay times of the gates in the cells**. As soon as **stable state** is reached, the output may be read.

Therefore, the iterative network can function as a parallel input, parallel-output device, in contrast with the sequential network in which the input and output are provided in a serial manner.

Example 6.1

The **Parity Checker** determines whether the number or 1's in a n-bit word is even or odd. The Figure **??** shows the complete iterative network for $n = 4$. The output of it will be 1 if an odd number of x inputs are 1. The logic of a cell can be described by

$$a[i+1] = a[i] \oplus x[i]$$

Assuming the delay of a cell (an EXOR gate in this example) is t_{cell}. The $a[1]$ input to the first cell must be 0 since no ones are received to the left of the first cell and 0 is an even number. The delay of the output of the last cell $a[5]$ is $4t_{cell}$ in this example.

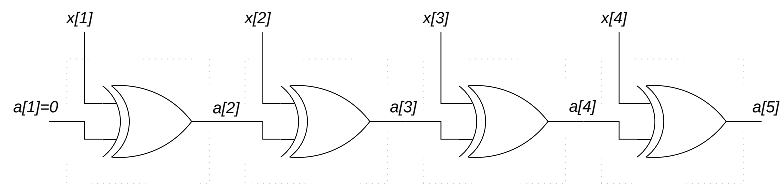


Figure 6.6: Parity Checker

Example 6.2

The **Ripple-carry binary adder** is used to perform addition on two binary numbers. A 4 bits adder can be constructed by cascading 4 full-adder circuit in series as shown in Figure **??**. The logic of a full-adder can be described by

$$c[i+1] = g[i] + p[i]c[i]$$
$$s[i] = p[i] \oplus c[i]$$

where $c[i]$, $c[i+1]$ and s are **carry input**, **carry output** and **sum output** of cell i, respectively. g **(generate)** and p **(propagate)** are two intermediate signals and can be described by

$$g[i] = a[i] \cdot b[i]$$
$$p[i] = a[i] \oplus b[i]$$

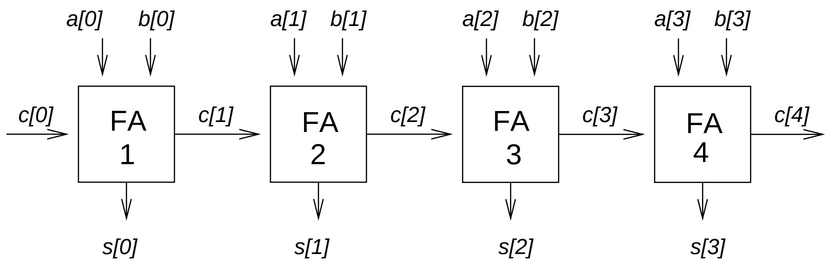


Figure 6.7: Ripple-Carry Binary Adder

Assuming the delay from inputs ($a[i]$ and $b[i]$) to intermediate signals (g and p) is t_1, and from intermediate signals and carry input ($c[i]$) to the outputs ($c[i+1]$ and $s[i]$) is t_2, and the input $c[0]$ is 0 (constant). It can be seen that the carry signal is propagated from left to right along the carry chain, and the carry chain is the worst-case delay path of the adder. Thus, the delay of this adder is $t_1 + 4t_2$.

For the design cases where an iterative network can be used, it offers several advantages over an ordinary combinational network:

- It is easier to design. Design of an iterative circuit can be reduced to the design of one block.

- It is easily expanded to accommodate more inputs simply by adding more cells.

The principal disadvantage of the iterative network is that the signal must be propagated through a large number of cells, so the response time will be longer than in a corresponding combinational network with few levels.

Concluding, an iterative circuit in general is a digital structure composed of a one-, two- or more dimensional structure of identical (or few types of) circuits or **cells**.

Both above examples were one-dimensional, one-directional iterative networks. An iterative circuit may be **sequential in nature,** where each cell is a sequential circuit, e.g., a shift register which consists of a number of cascaded flip-flops, or it may be a combinational network where each cell is a combinational circuit, as, for instance, the two example above.

Observe that every finite output sequence that can be produced **sequentially** by a **sequential machine** can also be produced **spatially** by a **combinational iterative circuit.** The first are calculated **in time**, the second are calculated **in space**.

As a consequence of the iterative structure, iterative circuits are easier to design, construct, and maintain. On the other hand, the time of operation is substantially longer than that of a two-level combinational realization. When realizing combinational circuits, for which speed of operation is not crucial and which can be composed of identical cells, the iterative circuits prove to be very useful and economical. They are particularly effective for functions with special properties, such as the symmetric functions, and when conditions for an output can be specified in terms of the relative positions of blocks of 1's and 0's.

More examples of various iterative circuits will follow.

Example 6.3

Design an iterative circuit to calculate the value of Petrick function for a given decision table and for any subset of table rows.

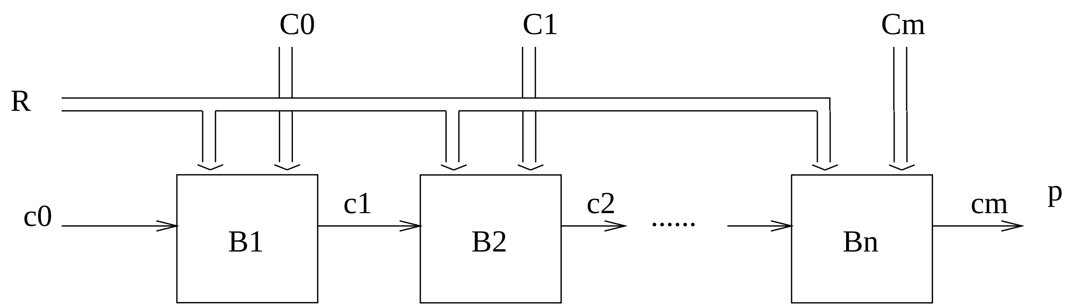


Figure 6.8: Petrick function circuit from blocks

The block schematics of the circuit is shown in Fig. ??. It consists of n blocks, n equal to the number of columns, each block having three inputs: vector R, vector C and carry signal c. Binary vector R of length m, m equal to the number of rows, contains 1s for the rows which are present in the subset for Petrick function calculation. Binary vector C of length m represents a column of the decision table. Petrick function for a decision table of n columns and m rows may be described by the following equation:

$$P = \prod_{j=1}^{n}(\sum_{i=1}^{m} D_{ij})$$

where D_{ij} is the value of the decision table at the crossing of row i and column j. The above equation is a product of sums, each sum is mapped into a block in our circuit. The value of the sum is the output of the block and, as a carry signal, input to the next block. The circuit which performs this operation is shown in Fig. ??. As we see, this modification of iterative circuit has **global inputs** R. Thus, a more general diagram of a one-dimensional, one-directional combinational iterative circuit is shown in Figure ??. The global control input can play the role of both control and data signals.

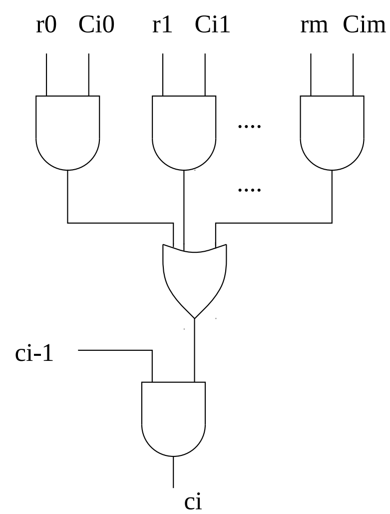


Figure 6.9: Internal structure of a block of Petrick Function Machine

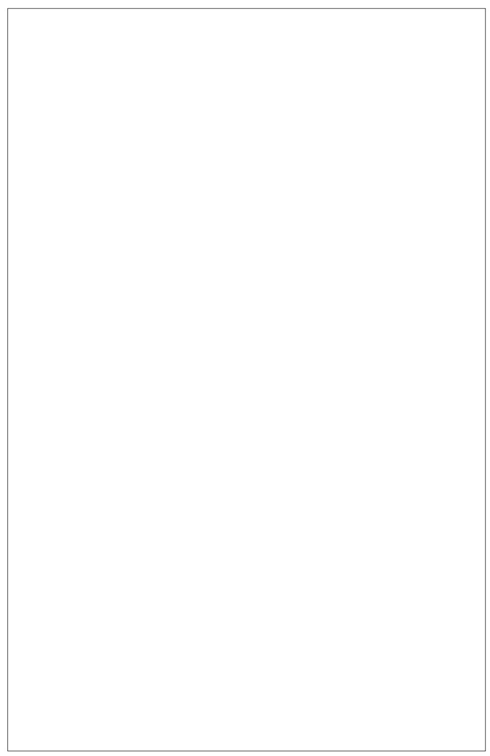


Figure 6.10: An iterative circuit with global inputs

Example 6.4

Problem: Design a circuit which would determine whether the number of ones in the input data vector is greater then three and less then seven.
Solution: We design a block having the following inputs and outputs:

- one input corresponding to one bit of the input data vector.
- three inputs coming from the the left neighbor of the current block and indicating how many ones is in the data vector left to the current block bit; if the number of ones is greater or equal to seven all three inputs will be equal to one.
- three outputs going to the right neighbor and indicating how many ones are in the data vector left to the right block bit; if the number of ones is greater or equal to seven all three inputs will be equal to one.
- the terminal block will determine whether the number of ones in the input data vector is within the correct range, it is an easy task since the number of ones in the data vector is available from the the last block.

Architecture of the circuit is shown in Fig. **??**a, K-map for the i-th block in Fig. **??**b and the architecture of the last block in Fig. **??**c. The last block is described by the following equations:

$$\overline{c_{n0}\ c_{n1}\ c_{n2}} = 1 \qquad for\ number\ of\ ones < 7$$

and

$$c_{n2} = 1 \qquad for\ number\ of\ ones > 3$$

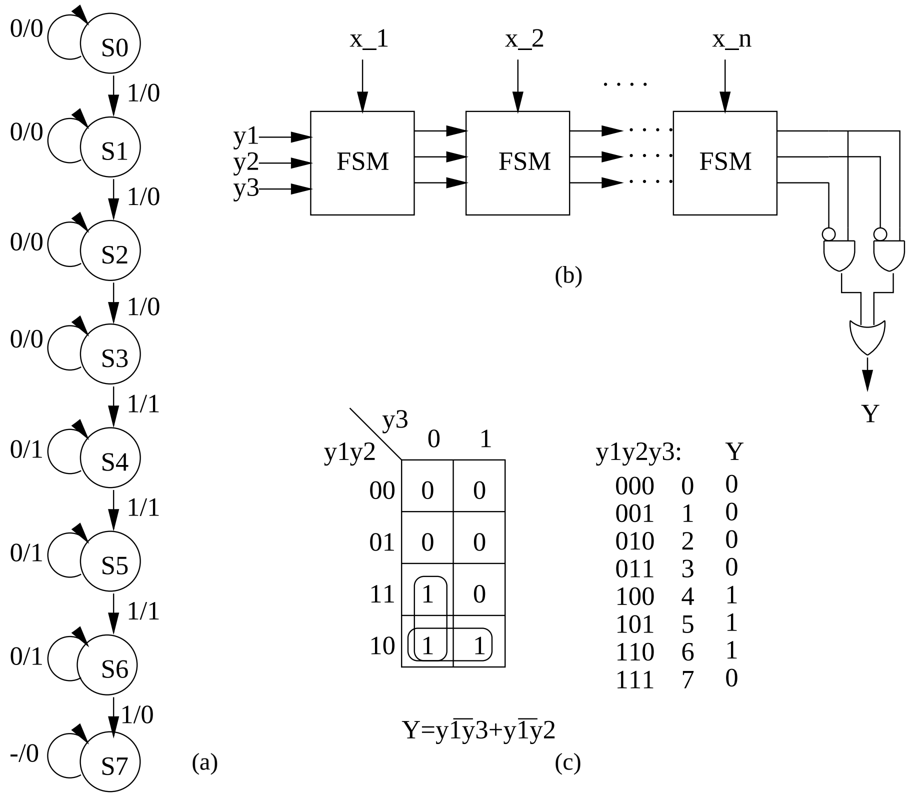


Figure 6.11: Iterative circuits, REMOVE DIAGRAMD FROM FSMS AND KMAP

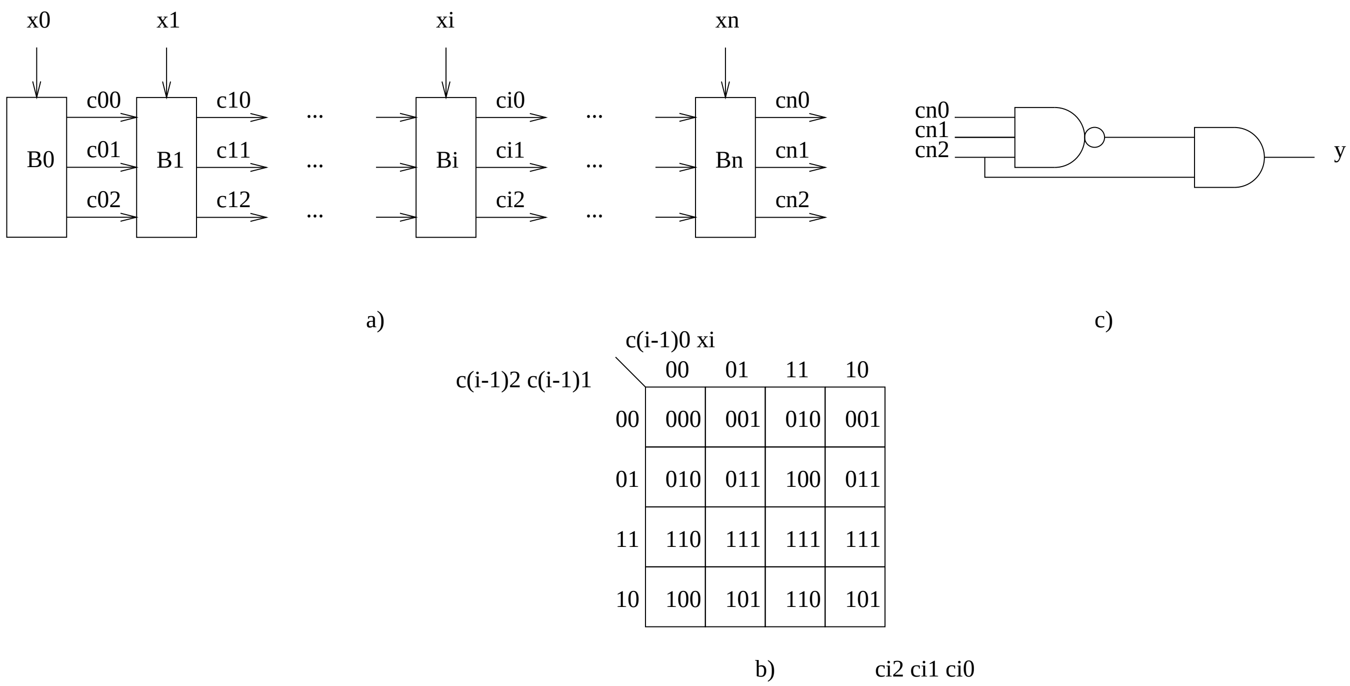


Figure 6.12: Iterative circuit, (a) architecture, (b) block K-map, (c) last block architecture

Example 6.5

Obviously, this iterative circuit should have 8 states to achieve the specified function. Let $S1, \ldots, S7$ be the states of cell. State $S1$ designates the state where none of the cell inputs to preceding cells is equal to 1. Similarly, states $S2$, $S3$, $S4$, $S5$, $S6$, $S7$ and $S8$ designate,respectively, the states where one, two, three, four, five, six, seven or more of the cell inputs to preceding cells are equal to 1. The state diagram is shown in Fig. **??** (a). The cell table (with state assignment) is shown in Fig. **??** (b). In this case, we are interested only in the output from the rightmost cell. Thus the cell has only one input, three input carries and three output carries. The cell outputs are eliminated and the circuit output is taken from the output carries of the last cell. The circuit output network can be derived from the K-map shown in Fig. **??**. Fig. **??** shows the systematical diagram of the circuit.

Figure 6.13: State Diagram

Figure 6.14: Cell Table and Output K-map

Figure 6.15: Systematical Diagram

Now we can write VHDL code to simulate this circuit. In VHDL, a designer can select different design styles. The available design styles are behavioral, structural and functional design.

- Behavioral
  VHDL provides concurrent statements to document parallel operations or to abstractly model an ultimate circuit in a behavioral manner. Within a process, sequential statements specify the step-by-step behavior of the process. The **PROCESS** statement is the primary concurrent statement in behavioral style. VHDL process has a sensitivity list. Processes wait for changes in an incoming signal which is listed in the sensitivity list. Sequential statements define algorithms for the execution within a process. And sequential statements execute in the order in which they appear in the process, as in programming languages. Behavioral model is a high level abstraction of circuits. In a top-down design style, this behavioral type of description is usually the first step; the designer can focus on the "abstract" behavior of a design, rather than the 1 and structural details of a design.

  As an example, the behavior design of the above iterative circuit is given as follows:

```
entity Iterative_circuit is
  port(X1,X2,...,Xn : in BIT;
       Y            : out BIT);
end Iterative_circuit;

architecture behavior of Iterative_circuit is
begin
   process(X1,X2,...,Xn);
   variable count: integer := 0;
   begin
     count the ones in X1,...,Xn;
     if (count is more than 3 but less than 7) then
       Y <= 1;
     else
       Y <= 0;
   end process;
end behavior;
```

- Structural
  The VHDL structural style describes the interconnection of components within an architecture. It is similar to a netlisting language in other CAD systems. In a structural architecture, we declare the components that we are using, then create instances of those components with particular mappings of signal wires to the various pins of the components. Most top-level VHDL designs use the structural style to instance and connect previously compiled designs.

  Let the above iterative circuit be an example, the structural design is as follows:

```
entity Iterative_circuit is
  port(X1,X2,...,Xn : in BIT;
       Y            : out BIT);
end Iterative_circuit;

architecture struct of Iterative_circuit is
signal Y1,Y2,...,Yn: BIT_VECTOR( 1 to 3 );
component CELL
  port (X: in BIT;
        yin: in BIT_VECTOR( 1 to 3 );
        yout: out BIT_VECTOR( 1 to 3 ) );
end component;

component OUTPUT
  port (yin: in BIT_VECTOR( 1 to 3 );
        yout: out BIT );
end component;
```

```
begin
  Cell1: CELL port map(X1,"000",Y1);
  Cell2: CELL port map(X2,Y1,Y2);
  Cell3: CELL port map(X3,Y2,Y3);
  .
  .
  .

  Celln: CELL port map(Xn,Y(n-1),Yn);
  Outnet: OUTPUT port map(Yn,Y);
end struct;
```

- Functional
  In behavioral design descriptions, subprograms provide a convenient way of documenting frequently used functions. A function has sequential statements contained inside, just like a process. A user-defined function needs to be declared if it is called. The function declaration goes in the architecture declaration section. Functions can also be declared in an entity or in a package. When a function is called, values are passed in through parameters prior to its execution. The function executes and returns only one value.

Chapter 7

Basics of Multiple-Valued Cube Calculus

Most of the efficient modern logic synthesis programs use **cube calculus** to represent and process Boolean functions. This representation is used in U.C.Berkeley programs, including the well-known Espresso [**?**], MIS II and SIS, and many others [**?**, **?**, **?**, **?**]. This calculus has been used for Boolean minimizers, tautology and satisfiability checkers, verifiers and other software tools[**?**].

In this chapter, the concept of a **set** is presented first because it is used as a fundament of cube calculus; then the concept of cube and the cube calculus are presented. The last part of this chapter presents positional notation of cubes and how to carry out the cube calculus in positional notation. The notations shown in Table **??** will be used in the entire book to represent different types of objects.

Table 7.1: Notation

Type	**Font**	**Example**
variable	lower case letter	$a,\ b,\ x,\ x_i$
literal	lower case letter	$\overline{a},\ x_i^{0,1,3}$
cube	upper case letter	$A,\ B$
array of cubes	upper case letter with arrow on head	$\vec{A},\ \vec{B}$
set	curly braces	$\{0,1,\ldots,n\}$
the name of set	upper case letter	$P,\ S$

7.1 Sets

A **set** is a collection of objects called **elements** or **members**. As listed in Table **??**, we use curly braces to indicate sets.

For instance, the set of all integers between 0 and 5 is written as:

$$\{0,1,2,3,4,5\}$$

the infinite set of all positive, odd integers can be described by:

$$\{1,3,5,7,\dots\}$$

The membership of a element a in a set A is denoted by $a \in A$ to mean *"a is an element of A"*. A set which has no element is called an **empty set** (denoted by $\emptyset$). The empty set is a subset of all sets. The elements contained in a set are either listed explicitly or described by their properties.

The following relations between two sets are used in cube calculus:

- Two sets A and B are **equal**, or **identical**, if they contain precisely the same elements. It is denoted by $A = B$.
- A set A is said to be a **subset** of B if every element of A is also an element of B. It it denoted by $A \subseteq B$.
- If $A \subseteq B$, and B contains at least one element which is not contained in A, then A is said to be **proper subset** of B. It is denoted by $A \subset B$.

The elements of the sets are taken from **universal set** (U). The following basic **set operations** are used in cube calculus:

- The **complement** of A in universal set U (denoted by $\neg A$) is the set of all elements of U that are not elements of A.
- The **intersection** of A and B (denoted by $A \cap B$) is the set containing the elements that are in both A and B.
- The **union** of A and B (denoted by $A \cup B$) is the set containing the elements that are in either A or B.

Example 7.1

Assuming that the universal set U is $\{0,1,2,3,4,5\}$, a set A is $\{0,1,2,3\}$ and another set B is $\{2,3,4\}$. Then $\neg A = \{4,5\}$, $A \cap B = \{2,3\}$, and $A \cup B = \{0,1,2,3,4\}$.

The universal set U of possible values of a binary variable is $\{0,1\}$. For a binary variable a, literal $\overline{a}$ means that literal is true when variable a is 0, and can be described by $a^{\{0\}}$, where $\{0\}$ is the true set of literal $\overline{a}$. More detailed discussion on sets can be found in [**?**, **?**].

7.2 The Concept of A Cube

The basic concepts of cube calculus are a **cube** and an **array of cubes**. A cube is a **product of literals**. For example, product $\overline{a}bc\overline{d}$ is a cube. An array of cubes is **a sum of cubes**. A logic function can be represented by a cube or an array of cubes. For instance, a simple 2-input binary logic function **AND** can be described by a cube as: $f_{AND}(a,b) = ab$; another simple 2-input binary logic function **XOR** (exclusive OR) can be described by an array of cubes as: $f_{XOR}(a,b) = \overline{a}b + a\overline{b}$.

In a binary logic, a **literal** is a binary variable with negation or without negation (x or $\overline{x}$). In a multi-valued logic a literal ($x_i^{S_i}$) is a variable (x_i) with its set of values (S_i) for which the variable is true.

A multi-valued input, two-valued output, incompletely specified switching function (**multi-valued function** for short) is a mapping:

$$f(x_1, x_2, \cdots, x_n) : P_1 \times P_2 \times \cdots \times P_n \mapsto B \tag{7.1}$$

where x_i is a **multi-valued (p_i-valued) variable**, $P_i = \{0, 1, 2, \cdots, p_i - 1\}$ is the **set of values** that variable x_i may assume, $B = \{0, 1, \mathrm{X}\}$ (X denotes a **don't care value**). Value n denotes the **number of variables (number of positions)**. For any subset $S_i \in P_i$, $x_i^{S_i}$ is a literal of x_i representing the function such that:

$$x_i^{S_i} = \begin{cases} 1 & \text{if } x_i \in S_i \\ 0 & \text{if } x_i \notin S_i \end{cases} \tag{7.2}$$

S_i is called **true values set** (**true set** for short) of literal $x_i^{S_i}$. For example, a four-valued input logic of $x^{\{1,2,3\}} = 1$ if $x \in \{1, 2, 3\}$, which means $x^{\{1,2,3\}} = 1$ if $x = 1$ or $x = 2$ or $x = 3$; otherwise, $x^{\{1,2,3\}} = 0$. We always assume that the set of possible values of a n-valued logic variable is $\{0, 1, 2, \ldots, n - 1\}$.

A product of literals, $x_1^{S_1} x_2^{S_2} \cdots x_n^{S_n}$, is refereed to as a **product term** (also called **product** or **term** for short), and can be represented by a cube. A product term that includes literals for all function variables $x_1, x_2, \ldots, x_n$ is called a **full term**. Any literal of the form $x_i^{P_i}$ is always equal to 1 since the literal is true for all possible values of x_i, thus $x_i^{P_i} x_j^{S_j}$ can be written as $x_j^{S_j}$.

A sum of products is denoted as a **Sum-Of-Products (SOP) Expression** while a product of sums is called a **Product-Of-Sums (POS) Expression**. An EXOR of products will be called a **Exclusive Sum Of Products** expression **ESOP**. A product of EXORs will be called a **Product Of Exclusive Sums** expression (**POES**). SOPE, POSE, ESOP and POES are all represented as an arrays of cubes. Products of SOPEs (PSOPEs) are also used in **Generalized Propositional Formulas** expressions to solve Boolean and Multi-valued equations. They are represented as **arrays of arrays of cubes**.

Definition 7.1 *The* **degree** *of a cube is the number of literals in the cube that are not equal to one (in other word, $P_i \neq S_i$).*

Example 7.2

The degree of binary cube $a^{\{0\}} b^{\{1\}} c^{\{1\}} d^{\{0,1\}}$ is 3 (assuming a,b,c and d are binary variables).

Definition 7.2 *The* **difference** *of two cubes is the number of variables for which the corresponding literals have different true sets.*

Definition 7.3 *The* **distance** *of two cubes is the number of variables for which the corresponding literals have disjoint true sets.*

Example 7.3

Given two cubes $A = a^{\{0,1\}} b^{\{0\}} c^{\{0\}}$, $B = a^{\{1,2\}} b^{\{1\}} c^{\{0\}}$. The difference of cubes A and B is 2 because they have different true sets on variables a and b. The distance of cubes A and B is 1 because they have disjoint true sets on variable b.

7.3 Cube Calculus Operations

The cube calculus operations presented in this boook can be categorized into three groups:

1. **simple combinational operations**,
2. **complex combinational operations**, and
3. **sequential operations**.

Each cube operation has one or two operand cube(s). Cubes A and B are used to represent these arguments and they can be described by:

$$A = x_1^{S_1^A} x_2^{S_2^A} \cdots x_i^{S_i^A} \cdots x_n^{S_n^A} \tag{7.3}$$

$$B = x_1^{S_1^B} x_2^{S_2^B} \cdots x_i^{S_i^B} \cdots x_n^{S_n^B} \tag{7.4}$$

where S_i^A and S_i^B are the true sets of literal $x_i^{S_i^A}$ and $x_i^{S_i^B}$, respectively. Value n is the **number of variables**. In this chapter, S represents the **true set of a literal**, the subscript of S represents the **index of the literal** (index of a variable corresponding to this literal), the superscript of S represents the **operand cube** (A or B).

7.3.1 Simple combinational cube operations

Simple combinational cube operations are defined as **single set operations**. Such a set operation is applied on all pairs of true sets S_i^A and S_i^B of corresponding literals of operand cubes. A simple combinational cube operation produces one resultant cube. The **intersection** and the **supercube** are simple combinational cube operations presented in this section.

Intersection

The intersection of two cubes A and B is the cube that is included in both A and B. The intersection operation of cubes A and B is defined as follows:

$$A \cap B = \begin{cases} x_1^{S_1^A \cap S_1^B} \cdots x_n^{S_n^A \cap S_n^B} & \text{if there is no such } i \text{ that } S_i^A \cap S_i^B = \emptyset \\ \epsilon & \text{otherwise} \end{cases} \tag{7.5}$$

where $S_i^A \cap S_i^B$ is a set intersection operation. $\emptyset$ denotes an empty set, and ϵ denotes a contradiction.

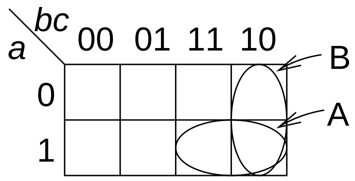
Example 7.4

Assuming two cubes $A = ab$ and $B = b\overline{c}$, where a, b and c are binary variables. The intersection of two cubes A and B is the following:

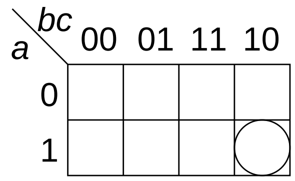
$$A = ab = ab\mathrm{X} = a^{\{1\}}\, b^{\{1\}}\, c^{\{0,1\}}$$

$$B = bc = \mathrm{X}b\overline{c} \;=\; a^{\{0,1\}}\, b^{\{1\}}\, c^{\{0\}}$$

$$A \cap B = a^{\{1\}\cap\{0,1\}}\, b^{\{1\}\cap\{1\}}\, c^{\{0,1\}\cap\{0\}} \;=\; a^{\{1\}}\, b^{\{1\}}\, c^{\{0\}} \;=\; ab\overline{c}$$



(a) Operand cubes



(b) Resultant cube

Figure 7.1: An example of Intersection operation

Example **??** is illustrated in Figure **??** by Karnaugh map. The intersection operation can be used in Ashenhurst/Curtis functional decomposition [**?**, **?**], in Satisfiability problem **??**, and in many other discussed in the forthcoming chapters.

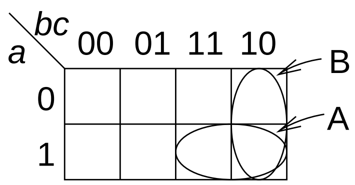
Supercube

The **supercube** of two cubes A and B is the smallest cube that includes cubes A and B. The supercube operation of cubes A and B is defined as follows:

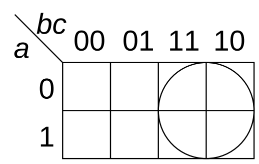
$$A \cup B = x_1^{S_1^A \cup S_1^B} \cdots x_n^{S_n^A \cup S_n^B} \tag{7.6}$$

where $S_i^A \cup S_i^B$ is a set union.
Example 7.5
The supercube on two cubes A and B used in Example **??** follows:

$$A \cup B = a^{\{1\} \cup \{0,1\}}\ b^{\{1\} \cup \{1\}}\ c^{\{0,1\} \cup \{0\}} \ = \ a^{\{0,1\}}\ b^{\{1\}}\ c^{\{0,1\}} \ = \ b$$



(a) Operand cubes



(b) Resultant cube

Figure 7.2: Example of Supercube operation

Example **??** is illustrated in Figure **??** by Karnaugh map. The supercube operation can be used in graph coloring approaches to the logic minimization problem [**?**], see also Chapter **??**.

7.3.2 Complex combinational cube operations

Complex combinational cube operations are defined by two set operations and one set relation. These two set operations are called **before** (bef for short) and **active** (act for short). All variables whose pair of true sets S_i^A and S_i^B satisfy **relation** are said to be **special variables**. The **active set operation** is applied on all pairs of true sets S_i^A and S_i^B of special variables. The **before set operation** is applied on the others. A complex combinational cube operation produces one **resultant cube** like a simple combinational cube operation. **Prime** is an example of a complex combinational cube operation presented in this section.

Prime

The **prime operation** of two cubes A and B is defined as:

$$A\,'B = x_1^{S_1^A} \cdots x_{k-1}^{S_{k-1}^A}\ x_k^{S_k^A \cup S_k^B}\ x_{k+1}^{S_{k+1}^A} \cdots x_{l-1}^{S_{l-1}^A}\ x_l^{S_l^A \cup S_l^B}\ x_{l+1}^{S_{l+1}^A} \cdots x_n^{S_n^A} \tag{7.7}$$

Where the relation for the prime operation is $S_i^A \cap S_i^B \neq \emptyset$. The *active* set operation is $S_i^A \cup S_i^B$, and the *before* set operation is S_i^A. In the above equation, variables x_k and x_l are the special variables.

Example 7.6

Assuming two cubes $A = \overline{x_1}x_2x_3x_4$ and $B = x_1\overline{x_3}$, where x_1, x_2, x_3 and x_4 are binary variables. The prime of $A\,'B$ is defined as follows:

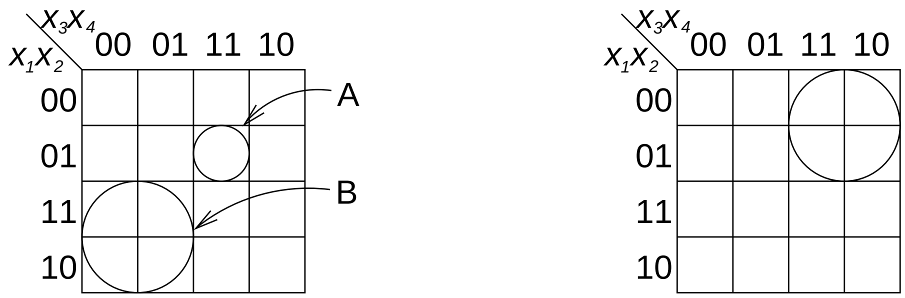
$$\begin{aligned} A &= \overline{x_1}x_2x_3x_4 = x_1^{\{0\}}\ x_2^{\{1\}}\ x_3^{\{1\}}\ x_4^{\{1\}} \\ B &= x_1\overline{x_3} = x_1\mathrm{X}\overline{x_3}\mathrm{X}\ =\ x_1^{\{1\}}\ x_2^{\{0,1\}}\ x_3^{\{0\}}\ x_4^{\{0,1\}} \end{aligned}$$

Because:

$$\begin{aligned} S_2^A \cap S_2^B &= \{1\} \cap \{0,1\} = \{1\} \neq \emptyset \\ S_4^A \cap S_4^B &= \{1\} \cap \{0,1\} = \{1\} \neq \emptyset \end{aligned}$$

variable x_2 and x_4 are special variables. Therefore,

$$A\,'B = x_1^{S_1^A}\ x_2^{S_2^A \cup S_2^B}\ x_3^{S_3^A}\ x_4^{S_4^A \cup S_4^B} = x_1^{\{0\}} x_2^{\{0,1\}} x_3^{\{1\}} x_4^{\{0,1\}} = \overline{x_1}x_3$$



(a) Operand cubes (b) Resultant cube

Figure 7.3: Example of Prime operation

Example **??** is illustrated in Figure **??** by Karnaugh map. The prime operation is used in the ESOP minimization program EXORCISM, developed at PSU [**?**, **?**], see also Chapter **??**.

Consensus

The **consensus operation** on cubes A and B is defined as follows:

$$A\ *\ B = \begin{cases} A \cap B & \text{when } distance(A,B) = 0 \\ \emptyset & \text{when } distance(A,B) > 1 \\ A *_{basic} B & \text{when } distance(A,B) = 1 \end{cases} \tag{7.8}$$

where $A\ *_{basic}\ B$ is defined as follows:

$$A\ *_{basic}\ B = x_1^{S_1^A \cap S_1^B} \cdots x_{k-1}^{S_{k-1}^A \cap S_{k-1}^B}\ x_k^{S_k^A \cup S_k^B}\ x_{k+1}^{S_{k+1}^A \cap S_{k+1}^B} \cdots x_n^{S_n^A \cap S_n^B} \tag{7.9}$$

where $S_k^A \cap S_k^B = \emptyset$. For basic consensus operation, the *before* set operation is $S_i^A \cap S_i^B$, the *active* set operation is $S_i^A \cup S_i^B$, and the *relation* is always true.

Example 7.7

Assuming two cubes $A = x_1 x_2 \overline{x_3}$ and $B = x_1 \overline{x_2}$, where x_1, x_2, x_3 and x_4 are binary variables. Because the distance of cubes A and B is 1, then cubes A and B have consensus as follows:

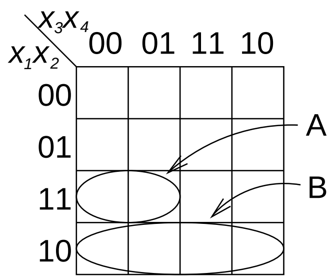
$$\begin{aligned} A &= x_1 x_2 \overline{x_3} = x_1^{\{1\}}\ x_2^{\{1\}}\ x_3^{\{0\}}\ x_4^{\{0,1\}} \\ B &= x_1 \overline{x_2} = x_1 \overline{x_2} \mathrm{XX} = x_1^{\{1\}}\ x_2^{\{0\}}\ x_3^{\{0,1\}}\ x_4^{\{0,1\}} \end{aligned}$$

Because:

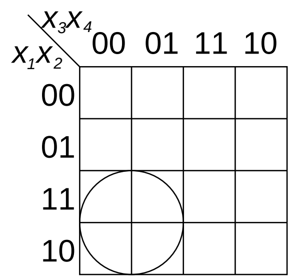
$$S_2^A \cap S_2^B = \{1\} \cap \{0\} = \emptyset$$

variable x_2 is a special variable. Therefore,

$$A\ *\ B = x_1^{S_1^A \cap S_1^B}\ x_2^{S_2^A \cup S_2^B}\ x_3^{S_3^A \cap S_3^B}\ x_4^{S_4^A \cap S_4^B} = x_1^{\{1\} \cap \{1\}}\ x_2^{\{0\} \cup \{1\}}\ x_3^{\{0\} \cap \{0,1\}}\ x_4^{\{0,1\} \cap \{0,1\}} = x_1 \overline{x_3}$$



(a) Operand cubes



(b) Resultant cube

Figure 7.4: Example of Prime operation

Example **??** is illustrated in Figure **??** by Karnaugh map. The consensus operation is used for finding prime implicants, and finding prime implicants is a basic step of the well-known **Quine-McCluskey algorithm** that is used for two-level logic minimization. Consensus is also used in its variants [**?**], as well as many other basic algorithms for two-level, three-level and multi-level logic minimization and machine learning [**?**]. A dual operation to consensus, **resolution**, is a base of automatic theorem-proving, [**?**].

Cofactor

The **cofactor operation** of two cubes A and B is defined as:

$$A \mid B = \begin{cases} A\,|_{basic}\,B & \text{when } A \cap B \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \tag{7.10}$$

where $A\,|_{basic}\,B$ is defined as follows:

$$A\,|_{basic}\,B = x_1^{S_1^A \cap S_1^B} \cdots x_{k-1}^{S_{k-1}^A \cap S_{k-1}^B}\ x_k^U\ x_{k+1}^{S_{k+1}^A \cap S_{k+1}^B} \cdots x_{l-1}^{S_{l-1}^A \cap S_{l-1}^B}\ x_l^U\ x_{l+1}^{S_{l+1}^A \cap S_{l+1}^B} \cdots x_n^{S_n^A \cap S_n^B} \tag{7.11}$$

where the relation for cofactor operation is $S_i^A \supseteq S_i^B$.

The result of the *active* set operation is always U (universal set), and the *before* set operation is $S_i^A \cap S_i^B$. In the above equation, variables x_k and x_l are special variables.

Example 7.8

Assuming two cubes $A = x_1x_2x_3$ and $B = x_1$, where x_1, x_2, x_3 and x_4 are binary variables. The cofactor of $A|B$ is defined as follows:

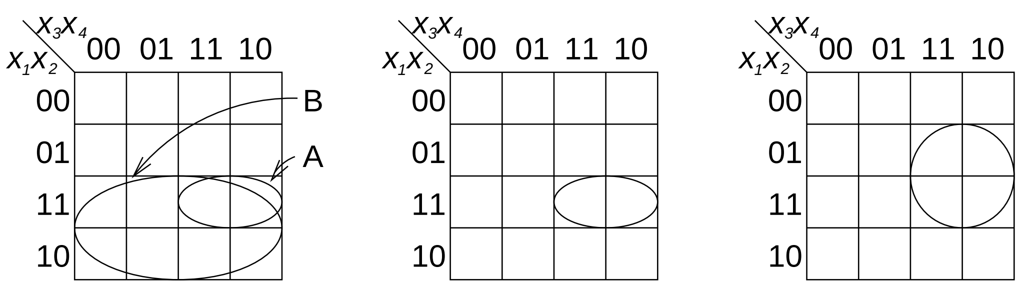
$$\begin{aligned} A &= x_1x_2x_3 = x_1x_2x_3\mathrm{X} = x_1^{\{1\}}x_2^{\{1\}}x_3^{\{1\}}x_4^{\{0,1\}} \\ B &= x_1 = x_1\mathrm{XXX} = x_1^{\{1\}}x_2^{\{0,1\}}x_3^{\{0,1\}}x_4^{\{0,1\}} \end{aligned}$$

Because:

$$\begin{aligned} S_1^A &= \{1\}\,, S_1^B = \{1\}\ \rightarrow\ S_1^A \supseteq S_1^B \\ S_2^A &= \{1\}\,, S_2^B = \{0,1\}\ \rightarrow\ S_2^A \not\supseteq S_2^B \\ S_3^A &= \{1\}\,, S_3^B = \{0,1\}\ \rightarrow\ S_3^A \not\supseteq S_3^B \\ S_4^A &= \{0,1\}\,, S_4^B = \{0,1\}\ \rightarrow\ S_4^A \supseteq S_4^B \end{aligned}$$

variable x_1 and x_4 are special variables. Therefore,

$$A|B \;=\; x_1^U\ x_2^{S_2^A \cap S_2^B}\ x_3^{S_3^A \cap S_3^B}\ x_4^U \;=\; x_1^{\{0,1\}}\ x_2^{\{1\} \cap \{0,1\}}\ x_3^{\{1\} \cap \{0,1\}}\ x_4^{\{0,1\}} = x_2x_3$$



(a) Operand cubes (b) Intersection (c) Result cube

Figure 7.5: Example of Cofactor operation

This example is illustrated in Figure **??** by a Karnaugh map. In the Karnaugh map, first we calculate the intersection of cubes A and B, the intersection result is shown in Figure

??(b), then we remove variable x_1 which means in the result cube, variable x_1 can be either 1 or 0 (**don't care**); the result cube is shown in Figure **??**(c). If there is no intersection of two operand cubes, then the cofactor is an empty cube. The cofactor operation is an important operation used in functional decomposition [**?**, **?**], creation of decision diagrams, and many other applications.

7.3.3 Sequential cube operations

All sequential cube operations are defined as a single formula that has three set operations and one set relation. These three set operations are called *before* (bef for short), *active* (*act* for short) and *after*(aft for short), respectively. All variables whose pair of true sets S_i^A and S_i^B satisfy *relation* are said to be *special variables*.

The sequential cube operations produces m resultant cubes, where m is the number of special variables for a given operation. The sequential cube operations can be generally described by the following fundamental equation:

$$A(op)B = \Big\{ x_1^{aft(S_1^A,S_1^B)} \cdots x_{i-1}^{aft(S_{i-1}^A,S_{i-1}^B)} x_i^{act(S_i^A,S_i^B)} x_{i+1}^{bef(S_{i+1}^A,S_{i+1}^B)} \cdots x_n^{bef(S_n^A,S_n^B)} \;\Big|\; \text{for such } i = 1, \ldots, n, \text{ that } relation(S_i^A, S_i^B) \text{ is true} \Big\} \qquad (7.12)$$

where x_i is a special variable, bef, act and aft are set operations.

Every special variable produces a resultant cube according to Equation **??**. This equation is a general pattern of all cube calculus operations and it was mentioned in the introduction.

Crosslink

The **crosslink operation** on cubes A and B produces an array of cubes defined as:

$$A \uplus B = \Big\{ x_1^{S_1^B} \cdots x_{i-1}^{S_{i-1}^B} x_i^{S_i^A \cup S_i^B} x_{i+1}^{S_{i+1}^A} \cdots x_n^{S_n^A} \;\Big|\; \text{for such } i = 1, \ldots, n, \text{ that } S_i^A \cap S_i^B = \emptyset \Big\} \qquad (7.13)$$

For crosslink operation, the *before* set operation is $bef(S_i^A, S_i^B) = S_i^A$, the *active* set operation is $act(S_i^A, S_i^B) = S_i^A \cup S_i^B$, the *after* set operation is $aft(S_i^A, S_i^B) = S_i^B$, and the *relation* is $S_i^A \cap S_i^B = \emptyset$. The crosslink operation can only be applied to two cubes when the two operand cubes are of the same degree, and `X` (don't care) must be in same position(s).

Example 7.9

Assuming variables x_1, x_2, x_3 and x_4 are binary, two cubes $A = \overline{x_1}\,\overline{x_3}$ and $B = x_1 x_3$, the crosslink operation $A \uplus B$ follows:

$$\begin{aligned} A &= \overline{x_1}\,\overline{x_3} = \overline{x_1}\mathrm{X}\overline{x_3}\mathrm{X} = x_1^{\{0\}}\, x_2^{\{0,1\}}\, x_3^{\{0\}}\, x_4^{\{0,1\}} \\ B &= x_1 x_3 = x_1\mathrm{X}\overline{x_3}\mathrm{X} = x_1^{\{1\}} x_2^{\{0,1\}} x_3^{\{1\}} x_4^{\{0,1\}} \end{aligned}$$

Because:

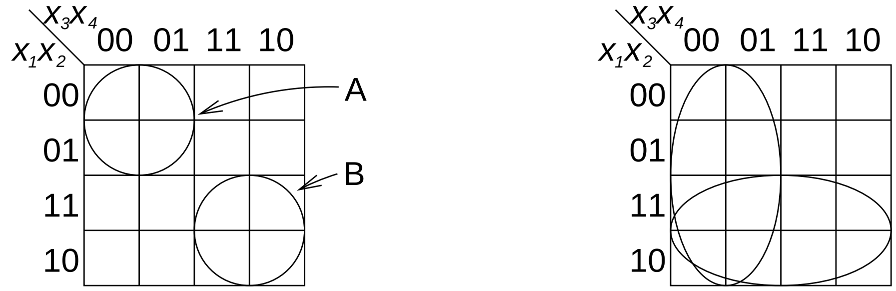
$$\begin{aligned} S_1^A \cap S_1^B &= \{0\} \cap \{1\} = \emptyset \\ S_3^A \cap S_3^B &= \{0\} \cap \{1\} = \emptyset \end{aligned}$$

the variable x_1 and x_3 are special variables. And two resultant cubes are:

$$
\begin{aligned}
x_1^{S_1^A \cup S_1^B} x_2^{S_2^A} x_3^{S_3^A} x_4^{S_4^A} &= x_1^{\{0\}\cup\{1\}} x_2^{\{0,1\}} x_3^{\{0\}} x_4^{\{0,1\}} = x_1^{\{0,1\}} x_2^{\{0,1\}} x_3^{\{0\}} x_4^{\{0,1\}} = \overline{x_3} \\
x_1^{S_1^B} x_2^{S_2^B} x_3^{S_3^A \cup S_3^B} x_4^{S_4^A} &= x_1^{\{1\}} x_2^{\{0,1\}} x_3^{\{0\}\cup\{1\}} x_4^{\{0,1\}} = x_1^{\{1\}} x_2^{\{0,1\}} x_3^{\{0,1\}} x_4^{\{0,1\}} = x_1
\end{aligned}
$$

Therefore,

$$A \uplus B = \overline{x_1}\,\overline{x_3} \uplus x_1 x_3 = \overline{x_3} + x_1$$

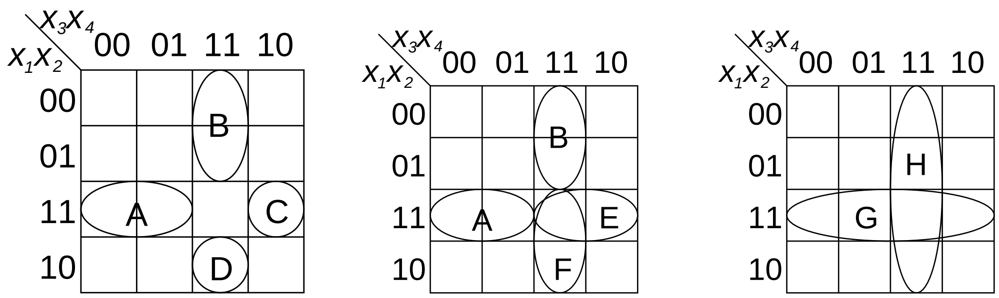


(a) Operand cubes (b) Resultant cube

Figure 7.6: Example of Crosslink operation

The Example **??** is illustrated in Figure **??** by Karnaugh map. The crosslink operation can be used in the minimization of logic functions in some canonical forms based on EXOR logic, for instance, in the Generalized Reed Muller form [**?**], as well as the general-purpose AND/EXOR expression (non-canonical), called ESOPs. The function $f = \overline{x_1}\,\overline{x_3} + x_1 x_3$ can be realized using EXOR gates as:

$$f = \overline{x_1}\,\overline{x_3} + x_1 x_3 = \overline{x_3} \oplus x_1$$



(a) Operand cubes (b) intermediate result (c) Resultant cubes

Figure 7.7: An example of using Complex Crosslink operations to minimize function $\overline{x_1} x_3 x_4 \oplus x_1 x_2 \overline{x_3} \oplus x_1 \overline{x_2} x_3 x_4 \oplus x_1 x_2 x_3 \overline{x_4} \Longrightarrow x_1 x_2 \oplus x_3 x_4$

Another more complex example is shown in Figure **??**. As shown in Figure **??**(a), we have a function $f = A \oplus B \oplus C \oplus D$, where A, B, C and D are four cubes. First, we calculate $C \uplus D$ and obtain cubes E and F, and the function becomes $f = A \oplus B \oplus E \oplus F$ as shown in Figure **??**(b). Second we calculate $A \uplus E$ and $B \uplus F$, and obtain cubes G and H; therefore, the function is simplified as $f = G \oplus H$.

The crosslink operation is used in ESOP minimization program EXORCISM, developed at PSU by an undergraduate student Martin Helliwell in 1988/89 [?, ?]. A more powerful cube operation, called **exorlink**, and a new ESOP minimization programs EXORCISM-MV-2 and EXORCISM-MV-3 based on variants of *exorlink* operation were developed also at PSU [?, ?, ?], see Chapter **??**.

Sharp

The **(non-disjoint) sharp operation** on cubes A and B is defined as follows:

$$A \# B = \begin{cases} A & \text{when } A \cap B = \emptyset \\ \emptyset & \text{when } A \subseteq B \\ A \#_{basic} B & \text{otherwise} \end{cases} \tag{7.14}$$

where $A \#_{basic} B$ is defined as follows:

$$A \#_{basic} B = \Big\{ x_1^{S_1^A} \cdots x_{i-1}^{S_{A,i-1}} \; x_i^{S_i^A \cap (\neg S_i^B)} \; x_{i+1}^{S_{A,i+1}} \cdots x_n^{S_n^A} \;\Big|\; \text{for such } i = 1, \ldots, n, \text{ that } \neg(S_i^A \subseteq S_i^B) \Big\} \tag{7.15}$$

For sharp operation, the *before* set operation is S_i^A, the *active* set operation is $S_{A_i} \cap (\neg S_i^B)$, the *after* set operation is S_i^A, and the *relation* is $\neg(S_i^A \subseteq S_i^B)$.

Example 7.10

Assuming variables x_1, x_2, x_3 and x_4 are binary, two cubes $A = \overline{x_3}$ and $B = x_2 x_4$, the sharp operation $A \# B$ follows:

$$\begin{aligned} A &= \overline{x_3} = \mathrm{XX}\overline{x_3}\mathrm{X} = x_1^{\{0,1\}} \, x_2^{\{0,1\}} \, x_3^{\{0\}} \, x_4^{\{0,1\}} \\ B &= x_2 x_4 = \mathrm{X}x_2\mathrm{X}x_4 = x_1^{\{0,1\}} \, x_2^{\{1\}} \, x_3^{\{0,1\}} \, x_4^{\{1\}} \end{aligned}$$

Because:

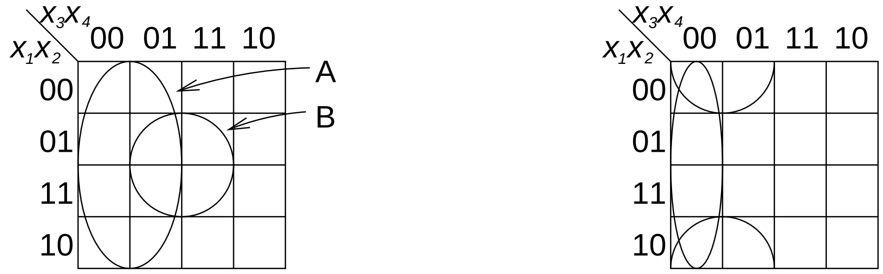
$$\begin{aligned} \neg(S_2^A \subseteq S_2^B) &= \neg(\{0,1\} \subseteq \{1\}) = true \\ \neg(S_{A,4} \subseteq S_{B,4}) &= \neg(\{0,1\} \subseteq \{1\}) = true \end{aligned}$$

variables x_2 and x_4 are special variables. Thus, 2 resultant cubes are:

$$\begin{aligned} x_1^{S_1^A} \, x_2^{S_2^A \cap (\neg S_2^B)} \, x_3^{S_3^A} \, x_4^{S_4^A} &= x_1^{\{0,1\}} \, x_2^{\{0,1\} \cap (\neg \{1\})} \, x_3^{\{0\}} \, x_4^{\{0,1\}} = x_1^{\{0,1\}} \, x_2^{\{0\}} \, x_3^{\{0\}} \, x_4^{\{0,1\}} = \overline{x_2} \, \overline{x_3} \\ x_1^{S_1^A} \, x_2^{S_2^A} \, x_3^{S_3^A} \, x_4^{S_4^A \cap (\neg S_4^B)} &= x_1^{\{0,1\}} \, x_2^{\{0,1\}} \, x_3^{\{0\}} \, x_4^{\{0,1\} \cap (\neg \{1\})} = x_1^{\{0,1\}} \, x_2^{\{0,1\}} \, x_3^{\{0\}} \, x_4^{\{0\}} = \overline{x_3} \, \overline{x_4} \end{aligned}$$

Therefore,

$$A \# B = \overline{x_3} \# x_2 x_4 = \overline{x_2} \, \overline{x_3} + \overline{x_3} \, \overline{x_4}$$



(a) Operand cubes (b) Resultant cube $A \# B$

Figure 7.8: Example of Sharp operation

Remember, universal set U of possible values of a binary variable is $\{0{,}1\}$, therefore, $\neg\{1\} = \{0\}$. This example is also illustrated in Figure **??** by a Karnaugh map. The sharp operation can be used in the tautology problem [**?**], TANT network minimization [**?**], 2-level minimization, and many other.

Disjoint sharp

The **disjoint sharp operation** on cubes A and B is defined as follows:

$$A \;\#d\; B = \begin{cases} A & \text{when } A \cap B = \emptyset \\ \emptyset & \text{when } A \subseteq B \\ A\#d_{basic}B & \text{otherwise} \end{cases} \tag{7.16}$$

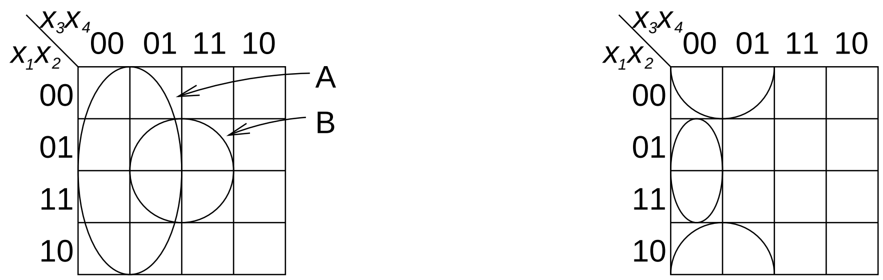
where $A \;\#d_{basic}\; B$ is defined as follows:

$$A \;\#d_{basic}\; B \;=\; \Big\{ x_1^{S_1^A \cap S_1^B} \cdots x_{i-1}^{S_{i-1}^A \cap S_{i-1}^B} \; x_i^{S_i^A \cap (\neg S_i^B)} \; x_{i+1}^{S_{i+1}^A} \cdots x_n^{S_n^A} \;\Big|\; \text{for such } i = 1, \ldots, n, \text{ that } \neg(S_i^A \subseteq S_i^B) \Big\} \tag{7.17}$$

For disjoint sharp operation, the *before* set operation is S_i^A, the *active* set operation is $S_{A_i} \cap (\neg S_i^B)$, the *after* set operation is $S_i^A \cap S_i^B$, and the *relation* is $\neg(S_i^A \subseteq S_i^B)$.

Example 7.11

The disjoint sharp operation $A \;\#d\; B$, where A and B are used in Example 2.5, is calculated as follows:



(a) Operand cubes (b) Resultant cube $A \#d\, B$

Figure 7.9: Example of Disjoint Sharp operation

Since the *relation* of disjoint sharp is the same as sharp, therefore variables x_2 and x_4 are still special variables. Thus, two resultant cubes are:

$$\begin{aligned}
x_1^{S_1^A \cap S_1^B}\ x_2^{S_2^A \cap (\neg S_2^B)}\ x_3^{S_3^A}\ x_4^{S_4^A} &= x_1^{\{0,1\}\cap\{0,1\}}\ x_2^{\{0,1\}\cap(\neg\{1\})}\ x_3^{\{0\}}\ x_4^{\{0,1\}} \\
&= x_1^{\{0,1\}}\ x_2^{\{0\}}\ x_3^{\{0\}}\ x_4^{\{0,1\}} \ = \ \overline{x_2}\ \overline{x_3} \\
x_1^{S_1^A \cap S_1^B}\ x_2^{S_2^A \cap S_2^B}\ x_3^{S_3^A \cap S_3^B}\ x_4^{S_4^A \cap (\neg S_4^B)} &= x_1^{\{0,1\}\cap\{0,1\}}\ x_2^{\{0,1\}\cap\{1\}}\ x_3^{\{0\}\cap\{0,1\}}\ x_4^{\{0,1\}\cap(\neg\{1\})} \\
&= x_1^{\{0,1\}}\ x_2^{\{1\}}\ x_3^{\{0\}}\ x_4^{\{0\}} \ = \ x_2\overline{x_3}\ \overline{x_4}
\end{aligned}$$

Therefore,

$$A \mathbin{\#d} B = \overline{x_3} \mathbin{\#d} x_2x_4 \ = \ \overline{x_2}\ \overline{x_3} + x_2\overline{x_3}\ \overline{x_4}$$

The Example **??** is also illustrated in Figure **??** by a Karnaugh map. The disjoint sharp operation can be used in tautology problem [?] and in conversions between SOP and ESOP representations, **??**.

7.3.4 Summary of cube calculus operations

From the above formulas (**??** to **??**), it is can be seen that sequential cube operations are the most complex operations in three groups of cube operations. Every sequential cube operation is defined by three set operations and one set relation. For the consistency of description, all cube operations in these three groups can be generally described by three set operations and one set relation. For simple combinational cube operations, only one set operation is used (called *before*); For complex combinational cube operations, two set operations and one set relation are used.

All cube operations (some of them are basic operations) described in this chapter are summarized in Table **??**. Every row describes one cube operation. For each operation, its name, notation, set relation and three set operations (called **output functions** in the Table) are listed from left to right, respectively.

Table 7.2: Cube Calculus Operations

Operation	**Notation**	**Relation**	**Output Functions**		
			before	*active*	*after*
crosslink	$A \uplus B$	$S_i^A \cap S_i^B = \emptyset$	S_i^A	$S_i^A \cup S_i^B$	S_i^B
sharp	$A \#_{basic} B$	$\neg(S_i^A \subseteq S_i^B)$	S_i^A	$S_i^A \cap (\neg S_i^B)$	S_i^A
disjoint sharp	$A \#d_{basic} B$	$\neg(S_i^A \subseteq S_i^B)$	S_i^A	$S_i^A \cap (\neg S_i^B)$	$S_i^A \cap S_i^B$
consensus	$A *_{basic} B$	1	$S_i^A \cap S_i^B$	$S_i^A \cup S_i^B$	$S_i^A \cap S_i^B$
intersection	$A \cap B$	1	$S_i^A \cap S_i^B$	–	–
supercube	$A \cup B$	1	$S_i^A \cup S_i^B$	–	–
prime	$A\,'B$	$S_i^A \cap S_i^B \neq \emptyset$	S_i^A	$S_i^A \cup S_i^B$	–
cofactor	$A\,\vert_{basic} B$	$S_i^A \supseteq S_i^B$	$S_i^A \cap S_i^B$	U	–

7.4 Positional Notation and Cube Operations in Positional Notation

From the above section, it can be seen that all cube operations are broken down into several set relations and set operations, and it is easy to carry out these set relations and set operations by hand. Now, the problem is how to represent sets in some way that they can be processed most efficiently by computers. Our answer to this problem is the **positional notation**.

7.4.1 Positional notation

In **Positional notation**, every possible value of a variable (binary or multi-valued) is represented by one bit, 0 or 1. Thus, a p-valued variable is represented by a string of p-bit; The i-th possible value is represented by the i-th bit. If the literal of this variable is true for a specific possible value (say the i-th possible value), the corresponding bit (the i-th bit) is set to 1, otherwise, it is set to 0.

Table 7.3: Positional Notation for binary literals

Binary literals	**Positional Notation**
$\overline{x}$	$\overline{x} = x^0 = x^{10} \rightarrow 10$
x	$x = x^1 = x^{01} \rightarrow 01$
X (don't care)	$\mathrm{X} = x^{0,1} = x^{11} \rightarrow 11$
ϵ (contradiction)	$\epsilon = x^{\emptyset} = x^{00} \rightarrow 00$

For example, a four-valued variable x is represented by a string of 4 bits. Literal $x^{\{0,2\}}$ is represented by 1010 because the first and third possible values let the literal be true.

The positional notation for binary literals is shown in table **??**. The *don't care* means the variable can be either 0 or 1, so both bits are set to 1. The *contradiction* means that the literal is not true for any possible value of variable, so both bits are set to 0. The last two cases, *don't care* and *contradiction*, can be extended to multi-valued variables. For p-valued variable, the string of 1's (the number of 1's is p) presents a *don't care*, and the string of 0's (the number of 0 is p) presents a *contradiction*.

7.4.2 Set operations in positional notation

As listed in Table **??**, all set operations used in cube operations are based on three basic set operations: intersection, union and complement. These three set operations can be executed using bitwise operations in positional notation:

- The set intersection operation can be executed using **bitwise AND** on two strings of bits that represent two true sets of literals in positional notation.

Example 7.12

Assume two literals $x^{\{0,1,2\}}$ and $x^{\{0,2,3\}}$, where x is a 4-valued variable. Thus two true sets of these two literals are {0,1,2} and {0,2,3}, respectively. The intersection of these two true sets is $\{0,1,2\} \cap \{0,2,3\} = \{0,2\}$. In positional notation, set {0,1,2} is represented by 1110, and set {0,2,3} is represented by 1011. The bitwise AND of 1110 and 1011 is 1010, which means set {0,2}, and this is just what we want. Therefore, the set intersection operation is executed by bitwise AND in positional notation.

- The set union operation can be executed using **bitwise OR** on two string of bits that represent two true sets of literals in positional notation.

Example 7.13

Assume two literals $x^{\{0,2\}}$ and $x^{\{3\}}$, where x is a 4-valued variable. Thus two true sets of these two literals are {0,2} and {3}, respectively. The union of these two true sets is $\{0,2\} \cup \{3\} = \{0,2,3\}$. In positional notation, set {0,2} is represented by 1010, and set {3} is represented by 0001. The bitwise OR of 1010 and 0001 is 1011, which means set {0,2,3}, and this is just what we want. Therefore, the set union operation is executed by bitwise OR in positional notation.

- The set complement operation can be executed using *bitwise NOT* on the string of bits that represents the true set of literal in positional notation.

Example 7.14

Assume a literal $x^{\{0,2\}}$, where x is a 4-valued variables. Thus the true sets of the literal is {0,2}. The complement of the true set is $\neg\{0,2\} = \{1,3\}$ (the $U = \{0,1,2,3\}$ for 4-valued variable). In positional notation, set {0,2} is represented by 1010. The bitwise NOT of 1010 is 0101, which means set {1,3}, and this is just what we want. Therefore, the set complement operation is executed by bitwise NOT in positional notation.

All other set operations can be done by combining these three basic set operations.

Example 7.15

Assume two literals $S^A = x^{\{0,2\}}$ and $S^B = x^{\{2,3\}}$, where x is a 4-valued variables. Thus two true sets of these two literals are $S^A = \{0,2\}$ and $S^B = \{2,3\}$, respectively. The set operation is:

$$S^A \cap (\neg S^B) = 1010\ AND\ (NOT\ 0011) = 1010\ AND\ 1100 = 1000$$

where AND and OR are bitwise operations. The result 1000 represents set {0}, which is correct result. This kind of set operation is called **set difference**, and is used in sharp and disjoint sharp cube operations.

7.4.3 Set relations in positional notation

The result of set relation is **true** or **false** and can be represented by one bit, 1 presents *true* and 0 presents *false*. The set relation can not be done by bitwise function because it is the function of all bits of two operand sets in positional notation.

Set relation is broken down into two parts in positional notation, **partial relation** and **relation type**. The *partial relation* determines whether or not a pair of the same possible value of two literals satisfy the *relation* "locally". The *relation type* determines the method of combining *partial relations*.

Assuming there are two literals x^A and x^B, where x is p-valued variable x, A is positional notation of true set of literal x^A, and $A = [a_0, a_2, \ldots, a_{p-1}]$, where a_i presents the $(i+1)$-th possible value of the literal (Note: the possible value starts with 0, ends with $p-1$), and $a_i \in \{0, 1\}$. Similarly, $B = [b_0, b_1, \ldots, b_{p-1}]$, where $b_i \in \{0, 1\}$.

For the crosslink operation, the set relation is $S^A \cap S^B = \emptyset$. Thus partial relation is $a_i \cdot b_i = 0$, or $\overline{a_i} + \overline{b_i} = 1$ (from De Morgan's theorem). If and only if all pairs of possible values satisfy this partial relation, then the set relation is satisfied. This can be written as:

$$relation(A, B) = (\overline{a_0} + \overline{b_0}) \cdot (\overline{a_1} + \overline{b_1}) \cdots (\overline{a_{p-1}} + \overline{b_{p-1}})$$

Therefore, the *partial relation* is $\overline{a_i} + \overline{b_i} = 1$, and the *relation type* of crosslink operation is AND type because AND function is used to combining all *partial relations*.

An example of OR type relation is the one used in the sharp operation, where the *relation* is $\neg(S^A \subseteq S^B)$. Thus partial relation is $\neg(A_i \subseteq B_i)$, where A_i is the subset of the true set S^A. If the set S^A includes the possible value $i-1$, then the set A_i has one element that is the possible value $i-1$ and is represented by $a_i = 1$; otherwise, the set A_i is an empty set and is represented by $a_i = 0$. It can be seen that a_i is the i-th bit of the bit string that represents the set S^A in positional notation. The same thing is with B_i and b_i.

Table 7.4: The partial relation of sharp operation

a_i	b_i	$A_i \subseteq B_i$	$\neg(A_i \subseteq B_i)$	$a_i \cdot \overline{b_i}$
0	0	1	0	0
0	1	1	0	0
1	0	0	1	1
1	1	1	0	0

Table **??** shows how to find the *partial relation* function for the sharp operation. The first column shows two bits a_i and b_i. The next two columns show the value of the negated *partial relation* and the *partial relation* itself, the last column shows the bitwise function used to determine the *partial relation*. In the first two rows, $a_i = 0$ means that the set A_i is an empty set, and it is subset of all sets, thus $A_i \subseteq B_i$ are 1's (true). In the third row, $a_i = 1$ means that the set A_i includes one element that is possible value $i-1$, and $b_i = 0$ means that the set B_i is an empty set, thus $A_i \subseteq B_i$ is 0 (false). In the fourth row, $a_i = b_i = 1$ means that the sets A_i and B_i include one same element that is possible value $i-1$, thus sets A_i and

B_i are equal, and $A_i \subseteq B_i$ is 1 (true). Therefore, the set relation of sharp operation can be determined by:

$$relation(A,B) = (a_0 \cdot \overline{b_0}) + (a_1 \cdot \overline{b_1}) + \ldots + (a_{p-1} \cdot \overline{b_{p-1}})$$

where the *partial relation* function is $a_i \cdot \overline{b_i}$, and the *relation type* is OR.

7.4.4 Summary of cube operations in positional notation

From the above discussion, *before*, *active*, *after* set operations and *partial (set) relation* can be defined by bitwise functions on bits. Therefore, all cube operations (some of them are basic operations) can be completely specified by 4 bitwise functions and *relation type*. Table **??** summarizes all cube operations described in this chapter in bitwise functions and relation type.

Table 7.5: Cube Calculus Operations in bitwise function and relation type

Operation	**Notation**	**Relation**		**Output Function**		
		partial relation	*relation type*	*before*	*active*	*after*
crosslink	$A \uplus B$	$\overline{a_i} + \overline{b_i}$	and	a_i	$a_i + b_i$	b_i
sharp	$A \#_{basic} B$	$a_i \cdot \overline{b_i}$	or	a_i	$a_i \cdot \overline{b_i}$	a_i
disjoint sharp	$A \#d_{basic} B$	$a_i \cdot \overline{b_i}$	or	a_i	$a_i \cdot \overline{b_i}$	$a_i \cdot b_i$
consensus	$A *_{basic} B$	1	and	$a_i \cdot b_i$	$a_i + b_i$	$a_i \cdot b_i$
intersection	$A \cap B$	–	–	$a_i \cdot b_i$	–	–
super cube	$A \cup B$	–	–	$a_i + b_i$	–	–
prime	$A'B$	$a_i \cdot b_i$	or	a_i	$a_i + b_i$	–
cofactor	$A\|_{basic}B$	$a_i + \overline{b_i}$	and	$a_i \cdot b_i$	1	–

The following examples show the entire procedure to carry out cube calculus operations in positional notation.

Example 7.16

Variables x_1, x_2 and x_3 all have 3 possible values, thus the sets of possible values are $P_1 = P_2 = P_3 = \{0,1,2\}$. A cube of $x_1^{\{0,1\}}\ x_2^{\{2\}}\ x_3^{\{0,1,2\}}$ is denoted as $x_1^{110}\ x_2^{010}\ x_3^{111}$ in positional notation; and it is written as [110-010-111] for simplifying.

Example 7.17

Assuming cubes $A = x_1x_2$ and $B = x_2x_3\overline{x_4}$, where variables x_1, x_2, x_3 and x_4 are binary, the intersection of cubes A and B follows:

$$\begin{aligned} A \cap B &= x_1x_2 \cap x_2x_3\overline{x_4} \\ &= x_1^{\{1\}}\ x_2^{\{1\}}\ x_3^{\{0,1\}}\ x_4^{\{0,1\}} \cap x_1^{\{0,1\}}\ x_2^{\{1\}}\ x_3^{\{1\}}\ x_4^{\{0\}} \\ &= x_1^{[01]}\ x_2^{[01]}\ x_3^{[11]}\ x_4^{[11]} \cap x_1^{[11]}\ x_2^{[01]}\ x_3^{[01]}\ x_4^{[10]} \end{aligned}$$

$$
\begin{aligned}
&= x_1^{[01]\cdot[11]}\, x_2^{[01]\cdot[01]}\, x_3^{[11]\cdot[01]}\, x_4^{[11]\cdot[10]} \\
&= x_1^{[01]}\, x_2^{[01]}\, x_3^{[01]}\, x_4^{[10]} \\
&= x_1 x_2 x_3 \overline{x_4}
\end{aligned}
$$

where '$\cdot$' is a bitwise AND operation.

When two opposite literals are multiplied, if the contradiction is generated from the bitwise operation, then there is no resultant cube, which means that there is no intersection between two operand cubes.

Example 7.18

Assuming cubes $A = ab$ and $B = a\overline{b}$, where a and b are binary variables. Then the intersection of cubes A and B is:

$$ab \cdot a\overline{b} = [01 - 01] \cdot [01 - 10] = [01 - \underbrace{00}_{\epsilon}]$$

where 00 is a contradiction symbol for a binary variable, and the contradiction is denoted by ϵ (see Table **??**).

Example 7.19

Redo the crosslink operation shown in Example 2.8 in positional notation as follows:

$$
\begin{aligned}
A &= \overline{x_1}\;\overline{x_3} = x_1^{\{0\}}\, x_2^{\{0,1\}}\, x_3^{\{0\}}\, x_4^{\{0,1\}} = [\overset{a_{1,0}}{1}\overset{a_{1,1}}{0} - \overset{a_{2,0}}{1}\overset{a_{2,1}}{1} - \overset{a_{3,0}}{1}\overset{a_{3,1}}{0} - \overset{a_{4,0}}{1}\overset{a_{4,1}}{1}] \\
B &= x_1 x_3 = x_1^{\{1\}}\, x_2^{\{0,1\}}\, x_3^{\{1\}}\, x_4^{\{0,1\}} = [\overset{b_{1,0}}{0}\overset{b_{1,1}}{1} - \overset{b_{2,0}}{1}\overset{b_{2,1}}{1} - \overset{b_{3,0}}{0}\overset{b_{3,1}}{1} - \overset{b_{4,0}}{1}\overset{b_{4,1}}{1}]
\end{aligned}
$$

where the header of bit is the name of the bit. The subscript of bit name has two parts separated by comma (','), the first part represents the index of the variable, and the second part represents the possible value. This notation will also be used in the next chapters. Because:

$$
\begin{aligned}
relation(A_1, B_1) &= (\overline{a_{1,0}} + \overline{b_{1,0}}) \cdot (\overline{a_{1,1}} + \overline{b_{1,1}}) = (\overline{1} + \overline{0}) \cdot (\overline{0} + \overline{1}) = 1 \\
relation(A_2, B_2) &= (\overline{a_{2,0}} + \overline{b_{2,0}}) \cdot (\overline{a_{2,1}} + \overline{b_{2,1}}) = (\overline{1} + \overline{1}) \cdot (\overline{1} + \overline{1}) = 0 \\
relation(A_3, B_3) &= (\overline{a_{3,0}} + \overline{b_{3,0}}) \cdot (\overline{a_{3,1}} + \overline{b_{3,1}}) = (\overline{1} + \overline{0}) \cdot (\overline{0} + \overline{1}) = 1 \\
relation(A_4, B_4) &= (\overline{a_{4,0}} + \overline{b_{4,0}}) \cdot (\overline{a_{4,1}} + \overline{b_{4,1}}) = (\overline{1} + \overline{1}) \cdot (\overline{1} + \overline{1}) = 0
\end{aligned}
$$

the variables x_1 and x_3 are special variables. The two resultant cubes are:

$$
\begin{aligned}
x_1^{A_1+B_1} x_2^{A_2} x_3^{A_3} x_4^{A_4} &= x_1^{[10]+[01]}\, x_2^{[11]}\, x_3^{[10]}\, x_4^{[11]} = x_1^{[11]}\, x_2^{[11]}\, x_3^{[10]}\, x_4^{[11]} = \overline{x_3} \\
x_1^{B_1}\, x_2^{B_2}\, x_3^{A_3+B_3}\, x_4^{A_4} &= x_1^{[01]}\, x_2^{[11]}\, x_3^{[10]+[01]}\, x_4^{[11]} = x_1^{[01]}\, x_2^{[11]}\, x_3^{[11]}\, x_4^{[11]} = x_1
\end{aligned}
$$

where '+' is the bitwise OR operation.

Chapter 8

ESOP Minimization For Incompletely Specified Multi-Output Functions

8.1 Introduction

The chapter presents an improvement to Exorlink operations, and a new approach to their use for Exclusive Sum of Products (ESOP) minimization. A new approach based on look-ahead search strategies is also introduced. Our program, EXORCISM-MV-3, is the successor to EXORCISM-MV-2, and can minimize multi-valued input, binary-output, incompletely specified, multi-output functions with different and arbitrary numbers of values in every variable. This makes it also useful in many applications, such as Machine Learning, and optimization of PLAs with input decoders. We observed that the new program is extremely fast; the speed-ups of up to 141 times were observed on large functions.

Although a classic in EXOR Logic, the ESOP minimization problem is far from being solved, and numerous results [?, ?, ?] point to various possible improvements of the existing ESOP minimizers. For instance, GRM or FPRM minimizers find, for some benchmark functions, the solutions that are better than the best known solutions of ESOP minimizers. Thus, despite many important works in recent years [?, ?, ?, ?, ?, ?, ?], the ESOP minimization problem still remains an open research area.

The work presented here is an extension and improvement to the approach from [?], implemented in program EXORCISM-MV-2. Here the application of the Exorlink operation was further improved by use of look-ahead ideas. As EXORCISM-MV-2, our new program, EXORCISM-MV-3, is able to find high quality solutions to incomplete multi-output functions with arbitrary number of values for each variable (which allows to run it with input decoders having more than two binary inputs).

Its main advantage is, however, that EXORCISM-MV-3 is very fast. In many practical applications such as: the minimization of regular arrays [?], the minimization of Generalized Reed Muller Forms [?], the minimization of arithmetic functions and functions specified with initial AND/EXOR descriptions [?], the multi-method logical approaches to Machine Learning [?], the array realizations of linearly independent logic [?], the multi-level realizations of linearly independent logic [?, ?], and the lattice realizations of functions [?, ?, ?] the ESOP minimization program must be very fast, because it is used in a loop of another minimization program. EXORCISM-MV-3 satisfies this requirement.

8.2 Background on Multiple-Valued Exorlink Operations

We review here the basic definitions of Exorlink for multiple-valued functions [**?**].

Definition 8.1 *A multiple-valued input, two-valued output, incompletely specified switching function f (*multiple-valued function, for short*) is a mapping $f(X_1, X_2, ..., X_n)$: $P_1 \times P_2 \times ...P_n \rightarrow B$, where X_i is a* multiple-valued variable, $P_i = \{0, 1, ..., p_i - 1\}$ *is a* set of admissible values *that this variable may assume, and $B = \{0,\ 1,\ x\}$ (x denotes a* don't care value*).*

Definition 8.2 *For any subset $S_i \subseteq P_i$, $X_i^{S_i}$ is a* literal *of X_i representing the function such that*

$$X_i^{S_i} = \begin{cases} 1 & \textit{if } X_i \in S_i \\ 0 & \textit{if } X_i \notin S_i. \end{cases}$$

Definition 8.3 *A product of literals, $X_1^{S_1} X_2^{S_2} ... X_n^{S_n}$, is referred to as a* product term *(denoted by PT, and also called* term *or* product *for short). A* minterm *is a product term that there exists only one value in each S_i for $i = 1, 2, ..., n$.*

Definition 8.4 *The* Exclusive-OR *(EXOR for short) of two products is assigned the value 1 if and only if the two products have different values. An EXOR of products is called an* Exclusive Sum of Products Expression *(* ESOP *for short). It is also called a* Multiple-Valued Input Exclusive Sum of Products Expression *(* MIESOP *for short) if one wants to emphasize that the input variables are multiple valued.*

Example 8.1

In binary logic, given three terms $T_1 = x\bar{y}\bar{z}$, $T_2 = \bar{x}y\bar{z}$, and $T_3 = \bar{x}\bar{y}z$. $T_1 \oplus T_2 \oplus T_3$ is a ESOP. Since the binary logic is a special case of *multiple-valued*, the above three product terms can be written in *multiple-valued* form as $T_1 = X^{\{1\}}Y^{\{0\}}Z^{\{0\}}$, $T_2 = X^{\{0\}}Y^{\{1\}}Z^{\{0\}}$, $T_3 = X^{\{0\}}Y^{\{0\}}Z^{\{1\}}$.

Example 8.2

In 4-valued logic, given three terms $T_1 = X^{\{1,2\}}Y^{\{2,3\}}$, $T_2 = X^{\{2,3\}}Y^{\{1,2\}}$, and $T_3 = X^{\{0,1\}}Y^{\{1,3\}}$. $T_1 \oplus T_2 \oplus T_3$ is a MIESOP. We can also call it ESOP.

In cube notation, a term is represented by a cube, and each literal in the term is represented by a vector:

$$c_1^0 c_1^1 ... c_1^{(p_1-1)} - c_2^0 c_2^1 ... c_2^{(p_2-1)} - ... - c_n^0 c_n^1 ... c_n^{(p_n-1)}$$

where

$$c_i^j = \begin{cases} 1 & \text{if } j \in S_i \\ 0 & \text{if } j \notin S_i. \end{cases}$$

For example, $X^{\{0\}}$ is denoted by 100...000, $X^{\{1\}}$ is denoted by 010...000, $X^{\{0,2\}}$ is denoted by 101...000, and $X^{\{0,1,...n\}}$ is denoted by 111...111, which represents the Boolean universe. A cube is a null cube, if one or more variables contain all 0s.

Example 8.3
The ESOPs in *example 1a* and *1b* can be written in the following cube notations respectively:

$$[01 - 10 - 10] \oplus [01 - 10 - 10] \oplus [10 - 10 - 01].$$

In Example 1b, the ESOP can be written in cube notation as follows:

$$[0110 - 0011] \oplus [0011 - 0110] \oplus [1100 - 0101].$$

Definition 8.5 *The* distance of two terms *is the number of variables for which the corresponding literals have different sets of values.*

Example 8.4
The distance of $T_1 = abd$ and $T_2 = a\bar{b}c$ is 3, because three literals have different sets of values:
for a: $\{01\} = \{01\}$, for b: $\{01\} \neq \{10\}$, for c: $\{11\} \neq \{01\}$, for d: $\{01\} \neq \{11\}$.

Example 8.5
Given three terms $T_1 = X^{\{0\}}Y^{\{1\}}$, $T_2 = X^{\{1\}}Y^{\{0,2\}}$, and $T_3 = X^{\{1\}}Y^{\{0,1\}}$. The distance of T_1 and T_2 is 2, because two literals have different sets of values:
for X: $\{0\} \neq \{1\}$, for Y: $\{1\} \neq \{0,2\}$.
The distance of T_2 and T_3 is 1, because only one literal has different sets of values:
for X: $\{1\} = \{1\}$, for Y: $\{0,2\} \neq \{0,1\}$.
We write *distance*$(T_i, T_j) = d$ to indicate that the distance of two terms T_i and T_j is d.
The objective of logic minimization is to find a realization that reduces certain cost function. Our primary goal of ESOP synthesis is to *minimize the number of terms* in the ESOP expression. For the expression with the minimum number of terms our secondary goal is to minimize the total number of inputs to the AND and EXOR gates. Follow the notation in [?], we use C_T to indicate the number of terms and C_L to indicate the number of literals.
Let $T_S = X_1^{S_1}...X_n^{S_n}$ and $T_R = X_1^{R_1}...X_n^{R_n}$ be two terms. The *exorlink* [?] of terms T_S and T_R is defined by the following formula:

$$T_S \otimes T_R = \bigoplus \left\{ X_1^{S_1}...X_{i-1}^{S_{i-1}} X_i^{(S_i \oplus R_i)} X_{i+1}^{R_{i+1}}...X_n^{R_n} \mid \text{for such } i = 1, ..., n, \text{that } S_i \neq R_i \right\}.$$

Here $\otimes$ denotes the exorlink operation and $\oplus$ denotes the EXOR operation.

Definition 8.6 *Given terms T_S and T_R, if the distance of two terms is d, then $T_S \otimes T_R$ is a* distance d exorlink.

It was proved in [?] that the exorlink can be applied to any two cubes in an array, without regard to their distance. According to the above formula, we can observe that distance d exorlink generates d resultant terms.

In the remainder of this section, distance 0, distance 1, distance 2, and distance 3 exorlink operations will be discussed.

Distance 0 Exorlink.
If the distance of two terms is 0, exorlink of the terms generates no resultant terms. These two terms are then removed from the ESOP description.

Distance 1 Exorlink.
Given two terms T_S and T_R, let X^{S_i} and X^{R_i} be a pair of literals in terms T_S and T_R, respectively, such that $X^{S_i} \neq X^{R_i}$, and the other pairs of literals in the two terms are equal. Therefore, these two terms will be called *"Distance 1 Exorlinkable."* The Distance 1 Exorlink operation of the terms generates a single resultant term.

Example 8.6
Let $T_S = X^{\{1,2,3\}}Y^{\{2,3\}}$ and $T_R = X^{\{0,1\}}Y^{\{2,3\}}$.

$$T_S \otimes T_R = X^{\{1,2,3\}\oplus\{0,1\}}Y^{\{2,3\}} = X^{\{0,2,3\}}Y^{\{2,3\}}.$$

Distance 2 Exorlink.
Given two terms, if the distance of these two terms is 2 (assume $X^{S_i} \neq X^{R_i}$, and $Y^{S_j} \neq Y^{R_j}$), then distance 2 exorlink operation can be performed on them, and two resultant terms will be generated. Please note that when distance ≥ 2, exorlink operation is not symmetric, which means $T_S \otimes T_R$ is different from $T_R \otimes T_S$.
Example 8.7
Given two terms $T_S = X^{\{0,1,3\}}Y^{\{1,3\}}$ and $T_R = X^{\{2,3\}}Y^{\{0,1\}}$.

$$T_S \otimes T_R = X^{\{0,1,3\}}Y^{\{1,3\}} \otimes X^{\{2,3\}}Y^{\{0,1\}} = X^{\{0,1,2\}}Y^{\{0,1\}} \oplus X^{\{0,1,3\}}Y^{\{0,3\}}$$

$$T_R \otimes T_S = X^{\{2,3\}}Y^{\{0,1\}} \otimes X^{\{0,1,3\}}Y^{\{1,3\}} = X^{\{0,1,2\}}Y^{\{1,3\}} \oplus X^{\{2,3\}}Y^{\{0,3\}}.$$

Distance 2 operations do not directly reduce the number of terms in an ESOP. However, these operations reshape two terms to two different terms, thus provide opportunities for reducing the cost of ESOPs at some later stages. The non-symmetry property of the Distance 2 Exorlink gives us two ways to reshape the two terms, which increases the opportunity for searching a better result. Our method on how to apply Distance 2 and Distance 3 Exorlink in ESOP minimization is discussed in sections 3 and 4.
Example 8.8
Given is an ESOP with three terms: $T_1 = X^{\{1,2\}}Y^{\{2,3\}}$, $T_2 = X^{\{2,3\}}Y^{\{1,2\}}$ and $T_3 = X^{\{0\}}Y^{\{1,3\}}$. In Figure 1, the three terms T_1, T_2 and T_3 are represented by three cubes A, B and C, respectively. $A \otimes B$ generates A' and B'; $A' \otimes C$ generates A''. The ESOP with three cubes is minimized to an ESOP with two cubes, B' corresponding to term $T_4 = X^{\{1,3\}}Y^{\{1,2\}}$, and A'' corresponding to term $T_5 = X^{\{0,1,2\}}Y^{\{1,3\}}$.

Distance 3 Exorlink.
Distance 3 Exorlink generates three resultant terms from two given terms. Distance 3 Exorlink increases the number of terms in the ESOP. However, increasing the number of terms may help to reduce the number of terms at some later stage, and subsequently lead to better results.
Example 8.9
In binary logic, a given ESOP with 4 cubes is as follows:

$$000x,\ 0x11,\ x11x,\ 1010.$$

The distances between any pair of cubes from the above set is 3. So, there are no distance 1 or distance 2 operations that can be applied to this set of cubes. Performing Distance 3 Exorlink on the first two cubes leads to three cubes:

Figure 8.1: ESOP minimization corresponding to Example 6

$$000x \otimes 0x11 = 0111 \oplus 00x1 \oplus 0000.$$

Replacing the first two cubes by these three cubes, a new ESOP with five cubes is obtained:

$$0111,\ 00x1,\ 0000,\ x11x,\ 1010.$$

Since the distance of two cubes: 0000 and 1010 is 2, a distance 2 exorlink can be performed on them:

$$0000 \otimes 1010 = x010 \oplus 00x0.$$

After this operation, the ESOP contains five cubes:

$$0111,\ 00x1,\ x010,\ x11x,\ 00x0.$$

Now, the distance of cubes $00x1$ and $00x0$ is 1, a Distance 1 Exorlink can be performed on them:

$$00x1 \otimes 00x0 = 00xx.$$

The ESOP now contains four cubes:

$$0111,\ 00xx,\ x010,\ x11x.$$

Performing Distance 2 Exorlink on cubes $x010$ and $x11x$, we obtain:

$$x010 \otimes x11x = xx10 \oplus x111.$$

The ESOP is

$$0111,\ 00xx,\ xx10,\ x111.$$

Cubes 0111 and $x111$ can be combined into one cube:

$$0111 \otimes x111 = 1111.$$

The final result is an ESOP with three cubes:

$$00xx,\ xx10,\ 1111.$$

By using Distance 3 Exorlink, the number of cubes in the ESOP is temporarily increased from 4 to 5, but this increase helps to jump out of a local minimum of the cost function, and achieve ultimately a better result of 3 cubes.

8.3 The New Exorlink Operations

This section presents the description of the new exorlink operations used in EXORCISM-MV-3. Distance 0, 1 and 2 exorlink are the same as those defined in previous section and [**?**].

Distance 2 Exorlink. Please note that Distance 2 Exorlink operation generates 4 resultant cubes:
Given are cubes: $A = X_1\ X_2$, and $B = Y_1\ Y_2$ (for simplification of notation, we avoid here writing in cubes the parts that are the same in cubes A and B).

Applying Distance 2 Exorlink to A and B, the resultant cubes are:
$cube_1$: $X_1\ (X_2 \oplus Y_2)$, $cube_2$: $Y_1\ (X_2 \oplus Y_2)$, $cube_3$: $(X_1 \oplus Y_1)\ Y_2$, $cube_4$: $(X_1 \oplus Y_1)\ X_2$.
Cubes $cube_1$ and $cube_3$ form one group and $cube_2$ and $cube_4$ form another group. After the Distance 2 Exorlink operation, either Group 1 or Group 2 is selected. But only the entire group can be selected, not a part of it. For instance, cubes $cube_1$ and $cube_2$ cannot be selected together as the result.

Distance 3 exorlink. Here we extend the Distance 3 Exorlink, as compared to [**?**]. Previously, in EXORCISM-MV-2, the Distance 3 Exorlink would generate two groups of 6 cubes. Given the cubes: $A = X_1\ X_2\ X_3$, and $B = Y_1\ Y_2\ Y_3$, the resultant cubes are:
Group 1:
$cube_1$: $(X_1 \oplus Y_1)\ Y_2\ Y_3$, $cube_3$: $X_1\ (X_2 \oplus Y_2)\ Y_3$, $cube_5$: $X_1\ X_2\ (X_3 \oplus Y_3)$.
Group 2:
$cube_2$: $(X_1 \oplus Y_1)\ X_2\ X_3$, $cube_4$: $Y_1\ (X_2 \oplus Y_2)\ X_3$, $cube_6$: $Y_1\ Y_2\ (X_3 \oplus Y_3)$.
By performing the Distance 2 Exorlink operation on above $cube_1$ and $cube_3$, the cubes $cube_7$ and $cube_8$ are obtained:
$cube_7$: $Y_1\ (X_2 \oplus Y_2)\ Y_3$, $cube_8$: $(X_1 \oplus Y_1)\ X_2\ Y_3$.
Cubes $cube_7$, $cube_8$ and $cube_5$ form a new group of resultant cubes for Distance 3 Exorlink. By doing all the permutations, the new Distance 3 Exorlink operation generates the following 12 resultant cubes (note the new enumeration of cubes):
$cube_1$: $X_1\ X_2\ (X_3 \oplus Y_3)$, $cube_2$: $Y_1\ Y_2\ (X_3 \oplus Y_3)$, $cube_3$: $(X_1 \oplus Y_1)\ Y_2\ Y_3$,
$cube_4$: $(X_1 \oplus Y_1)\ X_2\ X_3$, $cube_5$: $X_1\ (X_2 \oplus Y_2)\ X_3$, $cube_6$: $Y_1\ (X_2 \oplus Y_2)\ Y_3$,
$cube_7$: $X_1\ (X_2 \oplus Y_2)\ Y_3$, $cube_8$: $(X_1 \oplus Y_1)\ X_2\ Y_3$, $cube_9$: $X_1\ Y_2\ (X_3 \oplus Y_3)$,
$cube_{10}$: $Y_1\ (X_2 \oplus Y_2)\ X_3$, $cube_{11}$: $(X_1 \oplus Y_1)\ Y_2\ X_3$, $cube_{12}$: $Y_1\ X_2\ (X_3 \oplus Y_3)$.
which form 6 groups: Group 1: $cube_1$, $cube_3$, $cube_7$; Group 2: $cube_1$, $cube_6$, $cube_8$; Group 3: $cube_3$, $cube_5$, $cube_9$; Group 4: $cube_2$, $cube_4$, $cube_{10}$; Group 5: $cube_2$, $cube_5$, $cube_{11}$; and Group 6: $cube_4$, $cube_6$, $cube_{12}$. Concluding, after the Distance 3 Exorlink, there are as many as 6 groups to be selected from, instead of only 2 groups that were used in EXORCISM-MV-2.

8.4 Look Ahead Strategies

In this section, we define several look-ahead strategies for searching best ESOPs.

One step look ahead.
Give an array of cubes, $\{p_1, p_2, .., p_n\}$, if distance of p_1 and p_2 is 2, a Distance 2 Exorlink operation can be performed. After this operation, there are two groups, one of them to be

selected. For each of the 4 resultant cubes, q_1, q_2, q_3 and q_4, the distance between q_i (i = 1,2,3,4) and p_j ($j \neq 1$ and $j \neq 2$) is checked. If it is found that there is a distance ≤ 1, the corresponding group of resultant cubes is selected, so the total number of cubes in the array can be reduced. This is called a *one-step-look-ahead* method.

Two step look ahead.
After one-step-look-ahead, if there are no Distance 0 or Distance 1 Exorlink operations possible, then if the distance of q_i and p_j is 2, the Distance 2 Exorlink can be performed again and 4 resultant cubes r_1, r_2, r_3 and r_4 are generated. Now it can be checked whether the distance of r_i ($i = 1, 2, 3, 4$) and p_k ($k \neq 1, 2, j$) is ≤ 1. This is called a *two-step-look-ahead* method.

Consequently, we can perform *multi-step-look-ahead.* Multi-step means 2 or more steps. Look-ahead means looking for Distance 0 and Distance 1 Exorlink operations. Look-ahead on Distance 3 Eexorlink can be also applied. The Distance 3 Exorlink increases the number of cubes by 1, the Distance 0 Exorlink will reduce the number of cubes by 2, the Distance 1 Exorlink will reduce the number of cubes by 1, and the Distance 2 Exorlink will keep the number of cubes unchanged. But Distance 2, and 3 Exorlinks will bring the ESOP expression to a different point, so they may also lead to better results of subsequent operations.

Multi-step look ahead with full back tracking.
Assume $distance(p_1,\ p_2) = 2$, and 4 resultant cubes q_1, q_2, q_3 and q_4 are generated. Assume p_{i1}, p_{i2}, ..., p_{im1} was found, there are m cubes which are distance 2 with q_1. So for q_1, there are m_1 ways to go. Accordingly, there are m_2,m_3 and m_4 ways to go for q_2, q_3 and q_4 respectively. So, at this step, there are $m_1 + m_2 + m_3 + m_4$ ways to go. For each of these ways, at the next step, there are $(n_1 + n_2 + n_3 + n_4)$ ways to go. Let's assume the maximum search depth step is set to the value of K. When the K_th step is reached, the algorithm goes back one step and tries other ways. If all the possible ways at each step are tried, it is called a *full backtracking* method.

The searching space for a full back tracking is huge, when the value of K is big. (The searching space depends also on how many Distance-2 cubes can be found at each step, which in turn depends on the number of cubes currently in the array).

Multi-step look ahead with limited backtracking.
Since the full backtracking is very expensive, there are two alternatives to be investigated: the *small* K strategy, and the *limited backtracking* strategy. Small K means setting parameter K to a small value, like 2 or 3. Limited backtracking means at each step only a few ways to go are selected based on some method (random selection is used in our current implementation). Practically, we found experimentally that the method of large K with a small number of backtracks is much faster and usually gives better results than the method of small K with a large number of backtracks. Although the second one is also sometimes very helpful.

Concluding, there are two basic strategies:

1. small K and large number of backtracks.

2. large K and small number of backtracks.

Practically, we found that the first strategy leads to the very rapid reduction, and then the cost (number of PTs) remains stable for many loops (local minimum). While the second strategy leads to very slow reductions, but better reductions are found if the loops continues.

8.5 The Algorithm of EXORCISM-MV-3

8.5.1 The loops.

The first method is used to obtaind a local minimum quickly. Then the second method is used to get out of the local minimum. The main operations in our algorithm are distance checking and exorlink. Both operations have worst case complexity of n (number of variables), which is linear. Since these operations are performed in many loops, so the **run time** depends mainly on the number of loop repetitions, which relates to number of cubes in the array (the more cubes, the more loop repetitions). The **stopping criteria** are the following: during the look ahead, we stop if we reach the number of loops, or complete the backtracking, or find the reduction.

Below are the loop strategies.

Distance 2 Look Ahead.

At each step, find if Distance 0 or 1 Exorlink is possible.
If yes, execute it and quit the look ahead.
else perform the Distance 2 Exorlink when possible, and move to the next step.
else if no Distance 2 Exorlink is possible, quit the look ahead.

Distance 2 Loop.

For each pair of cubes in the array,
if the distance is 0 or 1, perform Exorlink, quit the loop.
else if the distance is 2, perform Exorlink, generate q_1, q_2, q_3, q_4.
if (distance(q_i, p_j*) < 2) perform Exorlink, quit the loop.*
else pick one group of the resultant cubes, continue to loop.

The Distance 2 Loop is equivalent to N-step Look Ahead without backtracking (N is the number of Distance-2 pairs in the array).

Distance 2 Loop2. Distance 2 Loop2 is the same as the Distance 2 Loop, except that at each step, 3 groups of cubes are picked (the two resultant groups and the original group), which have the smallest literal count.

Distance 3 Look Ahead.

At each step, find if Distance 3 Exorlink is possible.
If Distance 3 Exorling is not possible, quit the Distance 3 Look Ahead.
else, for each of the 12 resultant cubes, check if
    Distance 0 or Distance 1 Exorlinks are possible.
    If yes, execute the Exorlink and move to the next step.
    else do not execute the Exorlink, looking for the next pair.

Distance 3 Loop1.

Perform an N step Distance-3 Look Ahead without backtracking.

Distance 3 Loop2.
Perform an N step Distance-3 Look Ahead with maximum M backtracking.

8.5.2 The algorithm of EXORCISM-MV-3.

(1) Distance 2 Loop.

(2) Distance 3 Loop1.

(3) Loop steps (1) and (2), until no improvements.

(4) Distance 3 Loop2.

(5) Loop steps (1) to (4), until no improvements.

(6) Distance 2 Loop2.

(7) If no improvements on the number of cubes, stop; else go back to step 1.

8.6 Experimental Results of EXORCISM-MV-3

We compared our new program, EXORCISM-MV-3 to our previous program, EXORCISM-MV-2. Both programs are run on the same machine. We collect all the test cases from literature on ESOP minimization. By *small examples* we characterize those that satisfy the formula:

(number of input variables + number of output variables) $*$ *Initial number of PTs* $<$ *20,000*

Table **??** presents comparison on small examples. Table **??** presents comparison on large examples. As seen in Table **??**, speedup of up to 141 can be achieved on large benchmark functions. Results generated by ESPRESSO are also put into the same tables for comparison. Please note that the outputs of EXORCISM are ESOPs and the outputs of ESPRESSO are SOPs, although the definitions for their cost functions are the same. From the tables we can see that ESOP minimization is significantly slower than SOP minimization. This is partially due to the fact that ESOP minimization is a more difficult problem than its counter part; partially due to the nature of our algorithm, which needs a large number of iterations.

8.7 Conclusions

One approach to minimize ESOPs is to apply a set of cube operations iteratively on each pair of cubes in the array [**?**, **?**, **?**, **?**, **?**, **?**, **?**]. The Exorlink operation is the most powerful operation in this approach, which can link any two cubes in an array of cubes of an arbitrary distance. The superiority of the new cube operation [**?**] ensures better results than the previous operations shown in the literature. Here, the application of the Exorlink operation was further improved by use of look-ahead ideas. Our program EXORCISM-MV-3 was

Table 8.1: Comparison on Small Examples

	Inputs	Outputs	EXORCISM-MV-3			EXORCISM-MV-2			ESPRESSO		
			C_T	C_L	Time	C_T	C_L	Time	C_T	C_L	Time
con1	7	2	9	37	0.1	9	37	0.0	9	32	0.0
misex1	8	7	12	82	0.2	12	85	0.2	12	96	0.01
xor5	5	1	5	10	0.1	5	10	0.0	16	96	0.0
bw	5	28	22	319	1.1	22	300	0.6	22	349	0.20
squar5	5	8	19	87	0.8	19	92	0.3	25	119	0.04
misex2	25	8	27	210	1.5	27	210	2.5	28	213	0.03
rd53	5	3	14	57	0.4	14	57	0.3	31	175	0.01
inc	7	9	26	176	1.7	26	176	1.7	30	198	0.05
rot8	8	5	35	257	3.5	35	257	3.9	57	385	0.19
5xp1	7	10	32	170	2.2	33	182	1.5	65	347	0.09
rdm8	8	8	31	157	1.9	31	157	1.8	77	399	0.11
f51m	8	8	31	156	2.2	31	160	3.4	77	400	0.36
sao2	10	4	28	288	2.6	28	285	1.5	58	496	0.07
b12	15	9	28	169	1.2	28	163	1.6	43	207	0.25
adr4	8	5	31	144	1.7	31	150	1.8	69	383	0.08
rd73	7	4	35	188	4.2	38	186	3.0	127	903	0.15
nrm4	8	5	67	501	12.0	67	503	18.3	123	877	0.32
9sym	9	1	51	426	4.3	51	423	7	86	602	0.18
clip	9	5	63	472	7.4	64	497	8.1	120	793	0.52
mlp4	8	8	60	395	16.1	61	389	10.4	128	891	0.74
log8	8	8	83	615	48.7	89	675	29.7	131	990	0.79
wgt8	8	4	56	306	29.6	55	307	14.4	255	2070	0.61
rd84	8	4	59	322	20.2	57	317	19.9	255	2070	0.56
t481	16	1	13	53	50.1	13	53	41.2	481	5233	1.22
Total	212	155	837	5597	213.8	846	5671	173.1	2325	18324	6.59

tested on many benchmark functions, and compared to EXORCISM-MV-2. As EXORCISM-MV-2, new program EXORCISM-MV-3 is able to minimize efficiently incomplete multi-output functions with arbitrary number of values for each variable, which allows to run it with arbitrary input decoders (which is equivalent to AND/EXOR PLA decomposition to AND/EXOR PLA and several standard PLAs that realize the decoders).

By increasing the look-ahead step to infinity one can in theory obtain the exact ESOP. But increasing the step means increase of the searching space, and full backtracking means do all the permutation inside that space. The run time becomes exponential, but one can obtain exact solutions, as well as solutions as close to exact as desired by setting the values of K. In conclusion, this approach can be used to obtain exact solutions for smaller functions.

In many practical applications the ESOP minimization program must be very fast because it is used in a loop of another minimization program. The EXORCISM-MV-3 program presented here in most cases maintains or improves on the quality of EXORCISM-MV-2, but it is orders of magnitude faster on large benchmark functions.

As pointed in [?], further work is needed to minimize efficiently very weakly specified functions, such as those that have more than 95% don't cares and occur in the area of Machine

Table 8.2: Comparison on Large Examples

	Inputs	Outputs	EXORCISM-MV-3			EXORCISM-MV-2			ESPRESSO		
			C_T	C_L	Time	C_T	C_L	Time	C_T	C_L	Time
e64	65	65	65	2272	12.4	65	2210	78.7	65	2210	0.26
duke2	22	29	78	909	11.8	79	920	72.3	86	996	0.38
ex5	8	63	72	704	10.4	72	920	61.8	74	1903	0.88
ex4	128	28	316	3140	121.4	316	3136	17122.2	279	1928	6.67
table5	17	15	156	2449	32.6	156	2453	149.4	158	2501	1.07
table3	14	14	166	2491	24.2	166	2491	189.7	175	2644	1.68
cps	24	109	135	2625	37.4	135	2462	467.0	163	2836	5.08
vg2	25	8	184	1988	29.4	184	1993	224.7	110	914	0.22
pdc	16	40	184	1646	56.1	187	1678	455.9	145	1432	22.03
sqr8	8	16	106	704	50.1	108	703	174.2	188	1419	1.64
apex3	54	50	268	4155	53.2	270	3916	2562.9	280	3292	5.12
add6	12	7	127	800	24.4	127	819	104.6	355	2551	1.41
seq	41	35	248	4822	77.0	245	4833	2996.5	336	6245	10.45
spla	16	46	262	3393	82.6	260	3420	1223.4	254	3208	2.29
apex4	9	19	439	6181	124.2	438	6292	7556.6	436	5419	10.82
apex1	45	45	286	3820	108.9	285	3796	4696.8	206	2842	2.47
misex3c	14	14	226	2134	53.4	227	2123	1397.2	197	1561	4.81
ex1010	10	10	670	7466	1141.5	695	7928	16506.3	302	2895	5.87
alu4	14	8	422	4430	506.3	447	4816	6827.9	575	5087	11.09
misex3	14	14	535	6632	669.8	545	6837	18896.5	690	7784	17.39
apex5	117	88	399	4027	457.6	400	4038	20156.8	1088	7281	59.35
Total	673	723	5344	66788	3684.7	5407	67784	101921	5063	56875	159.77

Learning. It can be, however, observed, that the ideas of EXORCISM-DC for strongly unspecified functions from [?] can be also generalized to multi-output Boolean relations [?], and next combined with EXORCISM-MV-3 to create an efficient ESOP minimizer for binary and MV-input, binary-output relations.

We found also, that the look-ahead strategies can be used successfully to many other applications in logic minimization. In placement and routing algorithms, look-ahead is used for small K (usually 2 or 3) only. Our idea here is large K with small number of backtracks.

Bibliography

[1]

[2] Algotronix, "The CHS 2*4, the World's First Custom Computer," 1992.

[3] R. Anderson "Programmable High-Speed State Machine with Sequencing Capabilities," *U.S. Patent*, no. 4,965,472, October, 1990.

[4]

[5] T. Aoki, et al, *Proc. IEEE ISMVL,* pp. 364-367, May 1989.

[6] M. Amamiya, M. Takesue, R. Hasegawa, and H. Mikami, "Implementation and Evaluation of a List-Processing-Oriented Data Flow Machine," *Proc. 13th Intern. Symp. on Computer Architecture,* pp. 10-19, 1986.

[7] Z. Arevalo and J.G. Bredeson, "A method to simplify a Boolean function into a near minimal sum-of-products for programmable logic arrays," *IEEE Trans. on Computers,* Vol. C-27, No. 11, pp. 1028-1039, Nov. 1978.

[8] Aristotle, "Politics," translated by Ernest Baker, *Oxford University Press,* Oxford, England.

[9] A.A. Arkhangelskyaya, V.G. Lazarev, and V.N. Roginskii, "The Mechanization of the Process of Synthesizing Relay Networks," *Vses. Mos. Sov. Teorii Ust. Rel. Deist.,* p. 73, 1957.

[10] A.A. Arkhangelskyaya, V.G. Lazarev, and V.N. Roginskii, "A Machine for the Synthesis of Contact Networks," *Prob. Pered. Infor.,* Vol. 1., pp. 41-52, 1959.

[11] A.A. Arkhangelskyaya, V.G. Lazarev, et al., "A Machine for the Synthesis of Relay-Contact Networks," *Prob. Pered. Infor.,* No. 6., pp. 5-23, 1960.

[12]

[13] Arnold, et al, "SPLASH 2," *Report,*

[14] R.L. Ashenhurst, "The Decomposition of Switching Functions," *Proc. Int'l Symp.Theory of Switching Functions,* pp.74-116,1959.

[15]

[16] A. Ast, R. Hartenstein, et al, "Novel High Performance Machine Paradigms and Fast-Turnaround ASIC Design Methods: A Consequence of, and, a Challenge to, Field-programmable Logic," in *FPL '92.*

[17] Ch. Babbage, *"Passages from the Life of a Philosopher,"* London: Longman, Green, 1864. Reprinted in 1968 by Dawsons of Pall Mall (London).

[18]

[19] P. Barrie, P. Cockshott, at al, "SPACE: A Scalable Cellular Array Architecture," *in FPL '92.*

[20] K.E. Batcher, "Design of a Massively Parallel Processor," *IEEE Trans. on Computers*, C-29,, 9,

[21] E.T. Bell, *"Men of Mathematics"*, New York: Simon & Schuster, 1965.

[22] V.E. Benes, "Mathematical Theory of Connecting Networks and Telephone Traffic," *Academic Press*, 1965.

[23]

[24]

[25] P. Bertin, D. Roncin, and J. Vuillemin, *Introduction to Programmable Active Memories*, PRL Research Report 3, Digital Equipment Corp., Paris Research Lab, June 1989.

[26] P. Bertin, D. Roncin, and J. Vuillemin, *Programmable Active Memories: a Performance Assessment*, PRL Research Report 24, Digital Equipment Corp., Paris Research Lab, March 1993.

[27]

[28] P. Bertin and P. Boucard, *DECPeRLe-1 Hardware Installation Manual*, Paris Research Liboratory, Digital Equipment Corp., January 1993.

[29] P. Bertin, *DECPeRLe-1 Software Version 1.2 Installation Guide and General Description*, Paris Research Liboratory, Digital Equipment Corp., January 1993.

[30] Ph. W. Besslich and M. W. Riege, "An Efficient Program for Logic Synthesis of Mod-2 Sum Expressions," *Proc. EUROASIC,* pp. 136-141, Paris, France, 1991.

[31] Bic, L., "Execution of Logic Programs on a Dataflow Architecture," *SIGARCH Newsletter,* Vol. 12., No. 3., pp. 290-296, June 1984.

[32] D. Bobrow, "A Symbolic Logic Machine to Minimize Boolean Functions of Four Variables, and Application to Switching Circuits," 12 pp., privately printed manuscript, Bronx 1952.

[33] M. Boden, *"Artificial Intelligence and Natural Man,"* New York, Basic Books, 1977.

[34] B. Boghosian and D. Levermore, "A cellular automaton for Burgers' equation," *Complex Systems*, 1, 1987, 17.

[35] Bolt, Beranek an Newman, Inc., "Development of a Butterfly Test Bed," Rep. 5872, *Quaterly Technical Report,* Number 1, 1985.

[36] M. Bolton, "Digital Systems Design with Programmable Logic," ISBN 0-201-14545-6 pp.108

[37] G. Boole, *"The Mathematical Analysis of Logic"*, Cambridge, 1847. reprinted Oxford, 1948; in Rush Rhees (ed.), Studies in Logic and Probability, Watts and Co., London, 1952.

[38] G. Boole, *"The Laws of Thought,"* London 1854, in Rhees, Studies in Logic.

[39] A. Borning, "Thinglab: A Constraint-Oriented Simulation Laboratory," *Stanford University Computer Science Department*, Rep. 79-746, 1979.

[40] D. Brand, and T. Sasao, "On the Minimization of And-Exor Expressions," *23 FTC*, pp. 1 - 9, 1990.

[41] D. Brand and T. Sasao, "Minimization of AND-EXOR Expressions using rewrite rules," *IEEE Trans. on Comput.,* Vol. C42, no. 5, pp. 568-576, May 1993.

[42] R. Brayton, G.D. Hachtel, L. Hemachandra, A.R. Newton and A.L. Sangiovanni-Vincentelli, "A comparison of logic minimization strategies using ESPRESSO. An APL program package for partitioned logic minimization," *Proc. Int. Symp. on Circ. and Syst.,* pp. 43-49, Rome, May 1982.

[43]

[44]

[45]

[46]

[47]

[48] R.K. Brayton, G. D. Hachtel, C.T. McMullen and A.L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis,* Boston, MA: Kluwer Academic Publishers, 1984.

[49] R.K. Brayton, P.C. Mc Geer, J.V. Sanghavi, and A.L. Sangiovanni-Vincentelli, "A New Exact Minimizer for Two-Level Logic Synthesis," in T. Sasao (ed), "Logic Synthesis and Optimization", *Kluwer Academic Publishers,* pp.1-31, 1993.

[50] Y. Breitbart, and R. Vairavan, "The Computational Complexity of a Class of Minimization Algorithms for Switching Functions," *IEEE TC*, Vol.C-20, No. 12, December 1979, pp.941-943.

[51] M.A. Breuer, (ed.), "Design Automation of Digital Systems," Vol.1, *Prentice Hall*, Englewood Cliffs, New Jersey, 1972.

[52] M.A. Breuer, and K. Shamsa, "A hardware router," *J.Digital Systems*, Vol IV, No. 4, pp. 393-408, 1981.

[53] G. Broomell, and J.R. Heath, "Classification Categories and Historical Development of Circuit Switching Topologies," *Computing Surveys*, 15, (2), pp. 95-133.

[54] M.F. Brown, "Reduced Solutions of Boolean Equations," *IEEE Trans. on Computers,* Vol. C-19, pp. 976-981, 1970.

[55] M.F. Brown, "Single-parameter Solutions of Flip-Flop Equations," *IEEE Trans. on Computers,* Vol. C-20, pp. 452-454, April, 1971.

[56] M.F. Brown, "On a Convenient Division of Labor in the Generation of Prime Implicants," *Computers and Electrical Engineering,* Vol. 6., pp. 267-271, 1979.

[57] M.F. Brown, and S. Rudeanu, "Consequences, Consistency and Independence in Boolean Algebra," *Notre Dame J. Formal Logic,* Vol. 22, No. 1., pp. 45-62, 1981.

[58] M.F. Brown, "Segmental Solutions of Boolean Equations," *Discrete Applied Mathematics,* Vol. 4, pp. 87-96, 1982.

[59] M.F. Brown, and S. Rudeanu, "Recurrent Covers and Boolean Equations," *Proc. Colloq. on Lattice Theory,* Szeged, Hungary, Aug. 1980. Published in *Colloquia Mathematica Societatis Janos Bolyai,* North-Holland Pub. Co., Vol. 33, pp. 55-86, 1983.

[60] M.F. Brown, and S. Rudeanu, "Prime Implicants of Dependency Functions," *Analele Universitatii Bucuresti,* Vol. 37, No. 2., pp. 6-11, 1988.

[61]

[62] F.M. Brown, "Boolean Reasoning. The Logic of Boolean Equations," *Kluwer Academic Publishers*, Boston/Dordrecht/London, 1990.

[63] D.W. Brown, "A state machine synthesizer," *Proc. 18th Design Auto. Conf.,* pp. 301-304, Nashville, Jun. 1981.

[64] S.D. Brown, R.J. Francis, J. Rose, Z.G. Vranesic, "Field Programmable Gate Arrays," *Kluwer Academic Publishers*

[65] S.A. Browning, "A Tree Machine," *Lambda Magazine* 1, 2, pp. 31-36.

[66] J.A. Brzozowski, and M. Yoeli, "Digital Networks," *Englewood Cliffs,* N.J., Prentice-Hall, 1976.

[67] B. Burack, "An Electrical Logic Machine," *Science,* Vol. 109, June 17, 1949, p. 610.

[68] A.W. Burks, D.W. Warren, and J.B. Wright, "An Analysis of a Logical Machine Using Parenthesis-free Notation," *Mathematical Tables and other Aids to Computation,* Vol. 8., April, 1954, p. 53.

[69] C.R. Carroll, "Hardware Path Finders," *Caltech VLSI Conference Proceedings*, California Institute of Technology, 1980.

[70] R.G. Casey and C. Delobel, "Decomposition of a Database and the Theory of Boolean Switching Functions," *IBM Journal Research and Development,* Vol. 17, No. 5 (Sep. 1973), pp. 374-386.

[71] *Cellular Automata and the Modeling of Complex Physical Systems,* Proc. of 1989 Les Houches Workshop, Springer, Berlin, 1989.

[72] K.K. Chakrabarti, A.K. Choudhury, and M.S. Basu, "Complementary Function Approach to the Synthesis of Three-Level NAND Network," *IEEE Trans. Comput.,* Vol. C-19, pp. 509-514, June 1970.

[73] S. Chakradhar, V. Agrawal, and M. Bushnell, "Neural Net and Boolean Satisfiability Models of Logic Circuits," *IEEE Design and Test of Computers,* October 1990.

[74] U.S. Chakravarthy, S. Kasif, M. Kohli, J. Minker, and D. Cao, "Logic Programming on ZMOB: A Highly Parallel Machine," *Departament of Computer Science*, University of Maryland, College Park, MD.,

[75] A.H. Chan, "Using decision trees to derive the complement of a binary function with multiple-valued inputs," *IEEE Trans. on Comp.,* Vol. C-36, pp. 212 - 214, Febr. 1987.

[76]

[77] P.K. Chan, M. Schlag, and M. Martin, "BORG: A Reconfigurable Prototyping Board using Field-Programmable Gate Arrays," in *FPGA '92.*

[78] H. Chang, "Bubbles for Relational Database," *Fourth Annual Workshop on Computer Architecture for Non-Numeric Processing*, pp. 110-115.

[79] A.K. Choudhury, K.K. Chakrabarti, and D. Sharma, "Some Studies on the Problem of Three-level NAND Network Synthesis," *Int. Journal of Control,* Vol. 6., No. 6., pp. 547-572, 1967.

[80] D. Chyan, and M.A. Breuer, "A placement algorithm for array processors," *Proc.20th Design Automation Conference,* pp.182- 188, June 1983.

[81] M. Ciesielski, "Minimization Based on Graph Coloring," *Proc. of the International Conference on Computer Design:*, VLSI in Computers, ICCD'89, October 1989, pp. 262-265. ????????????/

[82]

[83] M. Ciesielski, S. Yang, and M.A. Perkowski, "Multiple-Valued Minimization Based on Graph Coloring", *Proc. ICCD'89,* pp. 262-265, October 1989.

[84] M.J. Ciesielski, and S. Yang, "PLADE: A Two-Stage PLA Decomposition," *IEEE Trans. on CAD*, Vol. 11, No. 8., pp. 943-954, Aug. 1992.

[85] E.F. Codd, "Relational Completeness of Data Base Sublanguages," in "Database Systems," R. Rustin (ed), *Prentice Hall.*

[86] M. Cohn, "Switching Function Canonical Forms over Integer Fields," *Ph.D. Dissertation,* Harvard University, Cambridge, MA, Dec. 1960.

[87] M. Cohn, "Inconsistent canonical forms of switching functions," *IRE Trans. Electron. Comput.,*

[88]

[89] Concurrent Logic, Application Notes and other company materials, 1992.

[90] G.P. Copeland, G.J. Lipovski, and S.Y.W. Su, "The Architecture of CASSM: A Cellular System for Non-numeric Processing," *Proceedings of the 1st Annual Symposium on Computer Architectures*, pp. 121-128.

[91] A. Coppola, M.A. Perkowski, R. Anderson, J. Freedman, and E. Pierzchala, "Synthesis of Very Fast Distributed Controllers Based on Tokenized State Machine Model," *Proc. IFIP Work. on Control-Dominated Synthesis,* Grenoble, pp. 1-15, September 1992.

[92] A. Coppola, R.E. Anderson, J. Freedman, and M.A. Perkowski, "VHDL Synthesis of Concurrent State Machines to a Programmable Logic Device", *VHDL Intern. User's Forum,* Scottsdale, Arizona, May 1992.

[93] T.H. Cormen, Ch. E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, The MIT Press and McGraw-Hill Book Company, Cambridge, Massachusetts, 1989.

[94] O. Coudert, "Two-level Logic Minimization: an Overview," *Integration, the VLSI Journal,* Vol. 17, No. 2, pp. 97-140.

[95]

[96] C.E. Cox, and W.E. Blanz, "GANGLION - A fast field programmable gate array gate array implementation of a connectionist classifier," *Report No. RJ 8290 (75651), IBM Research Division,* Almaden Research Centre, 1990.

[97] L. Csanky, M. Perkowski, and I. Schaefer, "Exact Minimization of Mixed-Radix Exclusive Sums of Products and Generalized Reed-Muller Forms," *IEE Transactions,* FIND. 1992?.

[98]

[99] L. Csanky, M. Perkowski, and I. Schaefer, "Canonical Restricted Mixed-Polarity Exclusive Sums of Products and the Efficient Algorithm for Their Minimization," *Proc. IEEE Intern. Symp. on Circuits and Systems,* pp. 17-20, San Diego, May 1992.

[100]

[101] R. Cucchhiara, et al, and T.S. Cinotti, "FPGAs boost flexibility and performance of a fine grain mesh connected SIMD array for computer vision applications," *in FPL '92.*

[102] H.A. Curtis, "A New Approach to the Design of Switching Ciruits," *Princeton, N.J., Van Nostrand,*1962.

[103] Cypress Semiconductor, in *BIMOS CMOS Data Book*, 1992.

[104] Cypress Semiconductor, *WARP2 Documentation,* 1996.

[105] M.R. Dagenais, V.K. Agarwal and N.C. Rumin, "McBOOLE: A new procedure for exact logic minimization," *IEEE Trans. on Computer-aided Design,* Vol. CAD-5, No. 1, pp. 229-237, Jan. 1986.

[106] C.J. Date, *An Introduction to Database Systems,* Addison-Wesley, 1975.

[107] Daumas, (NAME?) *"History of Technology,"* Vols. I + II, (EDITOR??)

[108] M. Davio, and J.P. Deschamps, "Classes of Solutions of Boolean Equations," *Philips Research Report,* Vol. 24., pp. 373-378, October 1969.

[109] M. Davio, J.P. Deschamps, and A. Thayse, *"Discrete and Switching Functions,"* McGraw-Hill Book Co., Inc., New York, 1978.

[110] M. Davis, and H. Putnam, "A Computing Procedure for Qualification Theory," *J. Assoc. for Computing Machinery,* Vol. 7., pp. 201-215, 1960.

[111] R. Davis, and D. Thomas, "Geometric Arithmetic Parallel Processor," *NCR*, 1984, preprint.

[112] C. Delobel, and R.G. Casey, "Decomposition of a Data Base and the Theory of Boolean Switching Functions," *IBM J. Res. & Develop.,* Vol. 17., pp. 374-386, 1973.

[113] G. De Michelli, A.L. Sangiovanni-Vincentelli, and T. Villa, "Computer-Aided Synthesis of PLA-Based Finite State Machines," *Proc. IEEE 1983 ICCAD*, pp. 154-156. Sept. 1983.

[114] G. De Michelli and A. Sangiovanni-Vincentelli, "Multiple folding of programmable logic arrays," *Proc. Int. Symp. on Circ. and Syst.,* pp. 1026-1029, Newport Beach (CA), May 1983.

[115] G. De Michelli and A. Sangiovanni-Vincentelli, "PLEASURE: A computer program for simple/multiple constrained/unconstrained folding of programmable logic arrays," *Proc. 20th Design Automation Conference,* pp. 530-537, Miami Beach (FL), Jun. 1983.

[116] G. De Michelli and A. Sangiovanni-Vincentelli, "Multiple constrained folding of programmable logic arrays: theory and applications," *IEEE Trans. on CAD of Int. Circ. and Syst.,* Vol. CAD-2, No. 3, pp. 167-180, Jul. 1983.

[117] G. De Michelli and M. Santomauro, "SMILE: A computer program for partitioning of programmed logic array," *Computer Aided Design,* No. 2, pp. 89-97, Mar. 1983.

[118] G. De Michelli and M. Santomauro, "Topological partitioning of programmable logic arrays," *Proc. Int. Conf. on Comp. Aid. Des.,* pp. 182-184, Santa Clara (CA), Sept. 1983.

[119] G. DeMicheli, R. Brayton, and A. Sangiovanni-Vincentelli, "Optimal State Assignment for Finite State Machines," *IEEE Trans. on CAD,* Vol. CAD-4, No. 3, July 1985, pp. 268-284.

[120] M.M. Denneau, "The Yorktown simulation engine," *In Proc. 19th Design Automation Conf.*, pp.55-59, June 1982.

[121] J.B. Dennis, and K.S. Weng, "Applications of Data Flow Computation to the Weather Problem", in "High Speed Computer and Algorithm Organization", D.J. Kuck, D.H. Lawrie, and A. Sameh (eds.), *Academic Press*, pp. 143-157.

[122] J.B. Dennis, "Data Flow Supercomputers," *Computer*, 18, pp. 42-56.

[123] J.P. Deschamps, "Maximal Classes of Solutions of Boolean Equations," *Philips Research Reports,* Vol. 26., pp. 249-260, August 1971.

[124]

[125]

[126] D.L. Dietmeyer, "Logic Design of Digital Systems," *Allyn and Bacon,* 1971. 2nd Edition, *Allyn and Bacon*, Boston, 1978

[127] K.M. Dill, K. Ganguly, R. Safranek, and M.A. Perkowski, "A New Linearly Independent, Galois-Field Reed-Muller Logic," *submitted to Reed-Muller'97.*

[128] E. Domany and W. Kinzel, "Equivalence of cellular automata to Ising models and directed percolation," *Phys. Rev. Lett.* 53, 1984, 311.

[129] T. Downs and M.F. Schulz, *Logic Design with Pascal Computer-Aided Design Techniques*, Van Nostrand Reinhold, New York, 1988.

[130] B. Dunahm, and H. Wang, "Towards Feasible Solutions to the Tautology Problem," *Ann. Math. Logic*, Vol. 10., pp. 117-154, 1976.

[131]

[132] ELTEC GmbH, "An Important Step in Image Processing," 1991.

[133] C. Engelbarts, *The Multiple-valued Cube Calculus Machine Version 2.5*, Dept. of Electrical Engineering, Portland State University, August 1993.

[134] C. Espinosa, and M.A. Perkowski: "Implementation of a Hierarchical Hough Transform on a Personal Computer," Preprint. 1991.

[135] C. Espinosa, M.A. Perkowski, "Hierarchical Hough Transform for the Vision System of a Wheelchair Robot," *IPCCC'92, IEEE International Phoenix Conference on Computers and Communications,* PAGES.

[136] A.C. Ewing, et al., "Algorithms for Logical Design," *Comm. & Electronics,* No. 56., pp. 450-458, 1961.

[137] B.J. Falkowski, and M.A. Perkowski, "Algorithms for the Calculation of Hadamard-Walsh Spectrum for Completely and Incompletely Specified Boolean Functions," *Proceedings of IEEE International Phoenix Conference on Computers and Communication*, Scottsdale, Arizona, March 1990, pp. 868-869.

[138] B.J. Falkowski, and M.A. Perkowski, "Walsh Type Transforms for Completely and Incompletely Specified Multiple-Valued Input Binary Functions," *Proc. of the 20th ISMVL, International Symposium on Multiple-Valued Logic,* Charlotte, NC, pp. 75-82, May 1990.

[139] B.J. Falkowski, and M.A. Perkowski, "Algorithm and Architecture for Gray Code Ordered Fast Walsh Transform," *Proc. of the ISCAS'90, International Symposium on Circuits and Systems,* New Orleans, 1-3 May 1990, pp. 1596-1599.

[140] B.J. Falkowski, and M.A. Perkowski, "A Family of All Essential Radix-2 Addition/Subtraction Multi-Polarity Transforms: Algorithms and Interpretations in Boolean Domain," *Proc. of the ISCAS'90, International Symposium on Circuits and Systems,* New Orleans, 1-3 May 1990, pp. 2913-2916.

[141] B.J. Falkowski, and M.A. Perkowski, "Essential Relations between Classical and Spectral Approaches to Analysis, Synthesis and Testing of Completely and Incompletely Specified Boolean Functions," *Proc. of the ISCAS'90, International Symposium on Circuits and Systems,* New Orleans, 1-3 May 1990, pp. 1656-1659.

[142] B.J. Falkowski, and M.A. Perkowski, "One More Method for the Calculation of the Hadamard-Walsh Spectrum for Completely and Incompletely Specified Boolean Functions," *International Journal of Electronics,* Vol. 69, No. 5, pp. 595-602, Nov. 1990.

[143] B. Falkowski, I. Schaefer, and M. Perkowski, "A Fast Computer Algorithm for the Generation of Disjoint Cubes for Completely and Incompletely Specified Boolean Functions," *Proc. of the Midwest Symp. on Circuits and Systems,* August 1990, Alberta, Canada, pp. 1119-1122.

[144] B.J. Falkowski, and M.A. Perkowski, "On the Calculation of Generalized Reed-Muller Canonical Expansions from Disjoint Representation of Boolean Functions," *Proc. of*

the 33rd Midwest Symp. on Circuits and Systems, August 1990, Alberta, Canada, pp. 1131-1134.

[145]

[146] B.J. Falkowski, and M.A. Perkowski, "An Algorithm for the Generation of Disjoint Cubes for Completely and Incompletely Specified Boolean Functions," *International Journal of Electronics,* Vol. 70, No. 3, pp. 533-538, March 1991.

[147] B.J. Falkowski, and M.A. Perkowski, "One More Way to Calculate Generalized Reed-Muller Expansions of Boolean Functions," *International Journal of Electronics,* Vol. 71, No. 3, pp. 383-396, September 1991.

[148]

[149] B.J. Falkowski, and M.A. Perkowski, "An Algorithm for the Generation of GRM forms," *Intern. J. Electronics,* Sept., 1991, Vol. 71, No. 3, pp. 383-396.

[150]

[151] B. Falkowski*, I. Schaefer*, M. Perkowski, "Effective Computer Methods for the Calculation of Rademacher-Walsh Spectrum for Completely and Incompletely Specified Boolean Functions," *IEEE Trans. on Computer-Aided Design,* pp. 1207 - 1226, October 1992.

[152] D. Farmer, T. Toffoli, and S. Wolfram, (Eds) "Cellular automata: Proceedings of the interdisciplinary workshop," *Physica D*, 10 (1,2), 1984.

[153] L.C. Farrell and M.E. Balogh, *An Application of Boolean Minimization to Database Normalization*, Techical Report, No. TR-90-25, Department. of Computer Science, Portland State University.

[154] C. Files, Andre Reis, M. Perkowski, M. Robert, and D. Auvergne "Minimization of Networks with Complex Gates," *Workshop on Boolean Problems,* Freiberg, Germany, September 1996.

[155] C. Files, R. Drechsler, and M.A. Perkowski, "Functional Decomposition of MVL Functions using Multi-Valued Decisions Dragrams," *Proc. of the IEEE ISMVL'97,* Halifax Nova Scotia, May 1997, pp. 27-32.

[156] H. Fleisher and L. Maissel, "An introduction to array logic," *IBM J. of Res. & Develop,* Vol. 19, pp. 98-109, Mar. 1975.

[157] J. Florine, "Optimization of Binary Functions with a Special-Purpose Electronic Computer," *Automation and Remote Control,* Vol. 28., pp. 956-962, 1967.

[158] J. Florine, "The Design of Logical Machines," *Crane, Russak & Co.,* New York, 1973.

[159]

[160] D.W. Foote, "The Design, Realization and Testing of the ILU of the CCM2 Using FPGA Technology," *Master Thesis*, Portland State University, Department of Electrical Engineering, 1994.

[161]

[162] J.A.B. Fortes, K.S. Fu, and B.J. Wah, "Systematic Approaches for Algorithmically Specified Systolic Arrays in Computer Architecture: Concepts and Systems," *Milutinovic ed.,* North Holland, 1988.

[163] J. Frackowiak, "The Minimization of Hazardless TANT Networks," *IEEE Trans. Comp.,* Vol. C-21., No. 10, pp. 1099-1108, Oct. 1972.

[164] R. Freivald, "Probabilistic Machines Can Use Less Running Time," *IFIP 77*, North-Holland, Amsterdam, 1977, pp. 839-842.

[165] U. Frisch, B. Hasslacher, and Y. Pomeau, "Lattice-gas automata for the Navier-Stokes equation," *Phys. Rev. Lett.* 56, 1986, 1505.

[166] M. Fujita, Y. Matsunaga, T. Kakuda, and K-Ch. Chen, "On Application of Boolean Unification to Combinational Logic Synthesis," *Proc. Intern. Workshop on Logic Synthesis*, May 7-10, 1991. Vol 1.

[167] M. Fujita, Y. Tamija, Y. Kukimoto, and K-Ch. Chen, "Application of Boolean Unification to Combinational Logic Synthesis," *Proc. ICCAD'91*, Nov. 1991, pp.510-513.

[168] H. Fujiwara, *Logic Testing and Design for Testability,* Computer System Series, The MIT Press, 1986.

[169]

[170] F. Furtek, G. Stone, and I.W. Jones, "Labyrinth: A Homogeneous computational medium," *Proc. IEEE Custom Integrated Circuits Conference,* May 1990, paper 31.1.

[171]

[172] F. Furtek, "An FPGA Architecture for Massively Parallel Computing," in *FPL'92.*

[173] M. Gardner, "Logic Machines and Diagrams," McGraw-Hill, 1958. Harvester, Brighton, 1983.

[174]

[175]

[176]

[177] M.R. Garey, and D.S. Johnson, "Computers and Intractability. A Guide to the Theory of NP-Completeness," *W.H. Freeman and Company,* San Francisco 1979.

[178] H.L. Garner, and J.S., Squire, "Iterative Circuit Computers," in "Computer Organization", Barnum (ed.), *Spartan Books,* 1963.

[179] M.A. Gavrilov, and A.D. Zakrevskii (eds.), "LYaPAS: A Programming Language for Logic and Coding Algorithms," *Academic Press,* New York, 1969.

[180] M.R. Genesereth, and N.J. Nilsson, "Logical Foundations of Artificial Intelligence," *Morgan Kaufmann,* Los Altos, CA, 1987.

[181]

[182] G.B. Gerace, et al, "TOPI-A Special-Purpose Computer for Boolean Analysis and Synthesis," *IEEE TC,* Vol. C-20, pp. 837-842, Aug. 1971.

[183] P. Gilliam, "A Practical Parallel Algorithm for the Minimization of Kronecker Reed-Muller Expansions," *M.S. Thesis*, Portland State Univ., August 1991.

[184] J.F. Gimpel, "The Minimization of TANT Networks," *IEEE TEC,* Vol. EC-16, pp. 18-38, Febr. 1967.

[185]

[186] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, "Building and Using a Highly Parallel Programmable Logic Array," in *FPL '92.*

[187] H.H. Goldstein, *"The Computer from Pascal to von Neumann,"* Princeton University Press, Princeton, 1972.

[188] L. Gomez-Gonzalez, "Estudio Teorico, Concepcion y Realizacion de un Sistema Electronico Para Simplificar Funciones Logicas," *Dissertation,* Dpto. Electricidad y Electronica, Facultad de Ciencias, Universidad de Granada, Spain, 1977.

[189] R.L. Goodstein, *"Development of Mathematical Logic,"* New York, Springer Verlag, 1971.

[190] A. Gottlieb, and J.T., Schwartz, "Networks and Algorithms for Very-Large-Scale Computation," *Computer*, January 1982, pp. 27-36.

[191] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. on Computers*, C-32, 2, pp. 175-189.

[192]

[193] J.P. Gray, and T.A. Kean, "Configurable Hardware: A New Paradigm for Computation," *Proc. Decennial Caltech Conference on VLSI,* Pasadena, CA, March 1989.

[194]

[195] D. Green, *Modern Logic Design,* Workingham, English: Addison-Wesley, 1986. Electronic Systems Engineering Series.

[196] D.H. Green, "Families of Reed-Muller Canonical Forms," *Intern. J. of Electr.,* pp. 259-280, February 1991, No 2.

[197] H. Gutowitz, (Ed), "Cellular Automata," MIT/North-Holland, 1991. (reprints from Physica D), published by *Center for Nonlinear Studies at Los Alamos National Laboratory.*

[198] G.D. Hachtel, A.R. Newton and A.L. Sangiovanni-Vincentelli, "An algorithm for optimal PLA folding," *IEEE Trans. on CAD of Int. Circ. and Syst.,* Vol. 1, No. 2, pp. 63-77, Apr. 1982.

[199] G.D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, Boston, 1996.

[200] P.R. Halmos, "Lectures on Boolean Algebras," *Springer-Verlag,* New York, 1974.

[201]

[202] R. Halverson, "Hawaii Parallel Computer Project: the Boolean Processor," in *FPL'92.*

[203] P.L. Hammer, and S. Rudeanu, "Boolean Methods in Operations Research," *Springer-Verlag,* New York, 1968.

[204] M.A. Harrison, "Introduction to Switching and Automata Theory," *McGraw-Hill,* New York, 1965.

[205] R.W. Hartenstein, K. Schmidt, et al, "A Novel Compilation Technique for a Machine Paradigm Based on Field-Programmable Logic," in *FPGAs,* pp. 255-270.

[206]

[207] R.W. Hartenstein, and K. Lemmert, "A CHDL-based CAD-system for the synthesis of Systolic Architectures," *Proc. Intl. Symposium on Hardware Description Languages and their Applications,* Washington D.C., North Holland Publ. Company, Amsterdam 1989.

[208]

[209] R.W. Hartenstein, A.G. Hirschbiel, and M. Weber, "A Novel Paradigm of Parallel Computation and its Use to Implement Simple High Performance Hardware," *Proc. of Joint Conference on Vector and Parallel Processing, CONPAR 90*, VAPP IV, Zurich, Sept, 1990.

[210]

[211] R.W. Hartenstein, A.G. Hirschbiel, and M. Weber, "XPUTERS: An Open Family of Non-von Neumann Architectures," *Nr. 195/89,* Universitaet Kaiserlautern, Fachbereich Informatik, Postfach 3049, D-6750 Kaiserlautern, 1989.

[212]

[213] R.W. Hartenstein, et al, "A Novel Paradigm of Parallel Computation and its use to implement simple High Performance Hardware," *Intl. Conf. on Information Technology,* Tokyo, Japan. Oct. 1990.

[214] R.W. Hartenstein, K. Schmidt, H. Reinig, and M. Weber, "A Novel Compilation Technique for a Machine Paradigm Based on Field-Programmable Logic," in *FPGAs,* pp. 255-270.

[215]

[216] R.W. Hartenstein, et al, "The machine Paradigm of Xputers and its Application to Digital Signal Processing," *Proc. of 1990 Intern. Conf. on Parallel Processing,* St. Charles, Oct. 1990.

[217]

[218] R.W. Hartenstein, et al, "A Novel ASIC Design Approach Based on New Machine Paradigm," *IEEE Journal of Solid-State Circuits,* July 1991, Vol. 26, Nr. 7, pp..

[219] R. Hartley, "The Stanhope Demonstrator," *Mind,* Vol. 4, April 1979.

[220] P.B. Hawthorn, and D. DeWitt, "Performance Analysis of Alternative Database Machine Architectures," *IEEE Trans. on Software Engineering* SE-8, 1, pp. 61-74, 1982.

[221] L.S. Haynes, R.L. Lau, D.P. Siewiorek, and D.W. Mizell, "A Survey of Highly Parallel Computing," *Computer*, January 1982.

[222]

[223] B. Heeb, and C. Pfister, "Chameleon: A Workstation of a Different Colour," in *FPL '92.*

[224]

[225]

[226]

[227] M. Helliwell, and M.A. Perkowski, "A Fast Algorithm to Minimize Multi-Output Mixed-Polarity Generalized Reed-Muller Forms," *Proc. 25-th ACM/IEEE Design Automation Conference,* paper 28.2, pp. 427-432, June 12- June 15, 1988.

[228] M. Helliwell, and M.A. Perkowski, "GAL-based Hardware Accelerator to Find Exact Minimum Solutions for Mixed-Polarity Generalized Reed Muller Forms," *report PSU.*

[229] C.E. Hewitt, "The Apiary Network Architecture for Knowledgeable Systems," *Proceedings of Lisp Conference*, Stanford, pp. 107-118, 1980.

[230] W.D. Hillis, *"The Connection Machine,"* The MIT Press, Cambridge, Massachusetts, 1985.

[231] T. Hirayama, and Y. Nishitani, "A Simplification Algorithm of AND-EXOR Expressions for Multiple-Output Functions," *Proc. of the Second Workshop on Applications of Reed-Muller Expansion in Circuit Design,* Chiba City, Japan, 27-29 August 1995, pp. 88-93.

[232]

[233] D.R. Hofstadter, *"Goedel, Escher, Bach: An Eternal Golden Braid,"* Vintage Books, 1980.

[234] A. Hodges, *"Alan Turing: The Enigma of Intelligence,"* Unwin, London, 1983.

[235]

[236]

[237] S.J. Hong, R.G. Cain, and D.L. Ostapko, "MINI: A heuristic approach for logic minimization," *IBM J. Res. & Develop.,* Vol. 18, pp. 443 - 458, Sept. 1974.

[238] B. Ho, "NAND Synthesis of Multiple-Output Combinational Logic Using Implicants Containing Output Variables," *Ph.D. Dissertation,* U. of Wisconsin, 1976.

[239]

[240]

[241] P.M. Ho, and M.A. Perkowski, "Systolic Architecture for Solving NP-Hard Combinational Problems of Logic Design and Related Areas," *Proc. ISCAS'89,* pp. 1170-1173, 1989.

[242] P.M. Ho, and M.A. Perkowski, "Evaluation of Systolic Architecture for Solving NP-Hard Combinatorial Problems of Logic Design and Related Areas," *Report*, PSU.

[243] P.M. Ho, and M.A. Perkowski, "Massively Parallel Processors for Solving Combinational Problems," *Portland State University*, Portland, OR, 1992.

[244] F. Hohn, "Applied Boolean Algebra," *Macmillan,* Second Edition, New York & London, 1966.

[245] K. Hrbacek and T. Hech, *Introduction to Set Theory*, Marcel Dekker, Inc., New York, 1984.

[246] D.A. Huffman, "Solvability Criterion for Simultaneous Logical Equations," *MIT Research Lab. of Electronics,* Quarterly Progress Report No. 48, AD 156-161, 15 Jan. 1958.

[247] S.L. Hurst, *The Logical Processing of Digital Signals, Crane-Russak,* New York and Edward Arnold, London, 1978.

[248] K. Hwang, and F.A. Briggs, "Computer Architecture and Parallel Processing," *McGraw-Hill*, New York, 1984.

[249] T. Ibaraki, "Theoretial comparisons of search strategies in branch-and-bound algorithms," *Intern. Journ. Comp. Sci.,* 5 (1976), pp. 315-344.

[250] M.E. Isenman, and D.E. Shasha, "Performance and Architectural Issues for String Matching," *IEEE JSSC,* Vol. 39, No. 2, pp. 238-250, Febr. 1990.

[251]

[252] P.C. Jackson, *"Introduction to Artificial Intelligence,"* New York, Petrocelli Charter, 1975.

[253] W.S. Jevons, *"The Principles of Science",* Dover, New York, 1958.

[254] W.S. Jevons, "Pure Logic or the Logic of Quality Apart from Quantity," *Stanford,* London, 1864.

[255] W.S. Jevons, "On the Mechanical Performance of Logical Inference," *Philosophical Transactions of the Royal Society,* CLX, 1870. pp. 497-518.

[256] D.S. Johnson "Worst-Case Behavior of Graph Coloring Algorithms,"*Proc. of the 5th SE Conf. on Combinatorics, Graph Theory and Computing,* Winnipeg, Canada, Utilitas Mathematica Publishing, pp. 513-528, 1974.

[257]

[258]

[259] D. Johnson, "The NP-Completeness Column: An Ongoing Guide," *Journal of Algorithms,* Academic Press, each issue.

[260] W.C. Kabat, and A.S. Wojcik, "Automated Synthesis of Combinational Logic Using Theorem-Proving Techniques," *Proc. 12 Int. Symp on Multiple-Valued Logic,* pp. 178-199, May 1982. IEEE Trans. on Comp. Vol. C-34, No. 7., pp. 610-632, July 1985.

[261] D. Kalish, and R. Montague, "Logic: Techniques of Formal Reasoning," *Harcourt Brace Jovanovich,* New York, 1964.

[262] R. Kane, and S. Sahni, "A systolic design rule checker," TR-83-13, Computer Science Dept., University of Minnesota, July 1983.

[263] S. Kang and W.M. VanCleemput, "Automatic PLA synthesis from a DDL-P description," *Proc. of the 18th Design Automation Conf.*, pp. 391-397, Jun. 1981.

[264]

[265] Ch. J. Kappler, "The MusiCAL Lambda Reduction Machine," in *FPL '92.*

[266] R.M. Karp, "Reducibility Among Combinatorial Problems," Complexity of Computer Computation, *Plenum Press*, ed. Miller, pp. 85-103, New York, 1972.

[267] V.I. Kashirov, et al., "Problems in Realizing the L-Machine," *Dokl. Rev. Ustr. Kon. Avt,* p. 138, 1962.

[268] P. Kashubin, A. Ojha and E. Tuers, "An Scaleable Minterm Sorter/Absorber Using Interative Circuit Techniques," Project report, Dept. of Electrical Engineering, Portland State University, Spring 1997.

[269]

[270] E. Katona, "Cellular Processing," Chapter 6 in "Fuzzy, Holographic, and Parallel Intelligence," By B. Soucek and the IRIS Group, pp. 215-229, John Wiley & Sons, 1992.

[271]

[272] T. Kean, "Using CAL to Accelerate Maze Routing of CAL Designs," in *FPL '92.*

[273] P. Kerntopf, and A. Michalski, "Selected Problems in Synthesis of Combinatorial Logic Circuits," *PWN*, Warsaw, 1972 (in Polish).

[274] L. Kida, "Documentation of Cube Calculus Machine," *PSU EE Dept. Report,* 1989.

[275]

[276]

[277] L. Kida, and M.A. Perkowski, "The Cube Calculus Machine: A Ring of Asynchronous Automata to Process Multiple-Valued Boolean Functions," *Proc. of the ISCAS'92, International Symposium on Circuits and Systems,* Sheratons on Harbor Island, San Diego, CA, May 10-13, 1992, pp. 807-810.

[278] L. Kida, "Associative Processing Implemented with Content-Addressable Memories," *M.S. Thesis*, PSU, 1991.

[279]

[280] S. Kleene, *"Introduction to Mathematical Logic,"* New York, John Wiley, 1967.

[281] G.J. Klir, "Introduction to the Methodology of Switching Circuits," *D. Van Nostrand Co.,* New York, 1972.

[282]

[283] A. Koestler, *"The Art of Creation,"* New York: Dell, 1966. A. Koestler, "The Art of Creation," *Dell*, New York, 1966.

[284] K.S. Koh, "A Minimization Technique for TANT Networks," *IEEE Trans. Comp.,* January 1971, pp. 105-107.

[285]

[286] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1978. (new edition, Computer Science Press).

[287] T. Kozlowski, "Heuristic Minimisation of ESOPs," *Reed-Muller Colloquium, UK'95*, pp. 4/1-4/5.

[288] T. Kozlowski, "Applications of Exclusive-OR logic in Technology Independent Logic Optimization," *Ph.D. Thesis,* Dept. of Electrical and Electronic Engn., University of Bristol, U.K., January 1996.

[289] Z. Kulpa, "Synthesis of Quasi-Minimal Logic Circuits of Many Variables with use of NAND and NOR gates," *M.Sc. Thesis,* Institute of Automatic Control, Warsaw Technical University, 1970.

[290] Kung, H.T., "Why Systolic Architectures," *IEEE Computer,* pp. 37-46, January 1982.

[291] H.T. Kung, and P.L. Lehman, "Systolic (VLSI) arrays for relational database operations," *International Conference on Management of Data,* May 1980.

[292] H.T. Kung, and C.E. Leiserson, "Systolic Arrays", in "Introduction to VLSI Systems," C.A. Mead and L.A. Conway, *Addison-Wesley.*

[293] Kung, H.T., "Special Purpose Devices for Signal and Image Processing: An Opportunity in Very Large Scale Integration (VLSI)," *SPIE*, Vol.241, Real Time Signal Processing III, pp.74-84, 1980.

[294] S.Y. Kung, "On Supercomputing with Systolic/Wavefront Array Processors," *Proc. IEEE*, Vol.72, pp.867-884, July 1984.

[295]

[296] Y.S. Kuo, "Generating essential primes for a boolean function with multiple-valued inputs," *IEEE Trans. on Computers,* Vol. 36, pp. 356-359, Mar. 1987.

[297] T. Lang, and H.S. Stone, "A Shuffle-Exchange Network with Simplified Control," *IEEE Transactions on Computers*, C-25, 1, pp. 55-65, 1976.

[298] V.G. Lazarev, and Yu. L. Sagalovich, "The Switchboard of a Machine for Synthesizing Contact Networks," *Prob. Pered. Infor.,* Vol. 4, pp. 124-132, 1959.

[299] V.G. Lazarev, and P.P. Parkhomenko, "A Machine for the Process of Analysis and Synthesis of the Structure of Relay Devices," *Proc. of the First Intern. Congress of the Intern. Federation of Automatic Control*, Moscow, 1960, (Butterworth, London, 1961), Vol. 2, pp. 853-859.

[300] V.G. Lazarev, O. A. Oganov, and V.N. Roginskii, "Fundamentals in the Design of Contactless Machines for the Synthesis of Relay-Contact Networks," *Prob. Pered. Infor.,* Vol. 8, pp. 5-19, 1961.

[301]

[302] H. V. D. Le, and M. A. Perkowski, "A New General Purpose Systolic Architecture for Matrix Computations," *Proc. of the Intern. Conf. on Computer and Information Systems,* Toronto, Ontario, Canada, May 23-29, 1989, pp. 182-185.

[303] H.V.D. Le, and M.A. Perkowski, "Size Independent Implementation of Matrix Operations on TASA - A Two-Dimensional Array Matrix Architecture," *Proc. of the IEEE International Phoenix Conference on Computers and Communication,* Scottsdale, Arizona, March 1990, pp. 889-890.

[304] H. V. D. Le, and M. A. Perkowski, "Realization of Extensions to Faddev Algorithm on Array of SIMD Processors," *Proc. of the ISCAS'90, International Symposium on Circuits and Systems,* New Orleans, 1-3 May 1990, pp. 2312-2315.

[305] R.M. Lea, "SCAPE: a Single-chip Array Processing Element for Signal and Image Processing," *IEE Proceedings,* Pt. E., Vol. 133, No. 3., pp. 145-151, May 1986.

[306] R.M. Lea, "VLSI and WSI Associative String Processor for Structured Data Processing," *IEE Proceedings,* Pt. E., Vol. 133, No. 3., pp. 153-162, May 1986.

[307] R.M. Lea, "ASP: A Cost-effective Parallel Microcomputer," *IEEE Micro,* Vol. 8, No. 5, pp. 10-29, October 1988.

[308] E. B. Lee, and M. Perkowski, "A new approach to structural synthesis of automata," *University of Minnesota, Department of Electrical Engineering,* preprint, 1982.

[309]

[310] E. B. Lee, and M. Perkowski, "Concurrent Minimization and State Assignment of Finite State Machines," *Proceedings of the 1984 Intern. Conf. on Systems, Man, and Cybernetics,* IEEE, Halifax, Nova Scotia, Canada, October 9 - 12, 1984.

[311] C.Y. Lee, "Intercommunicating Cells, Basis for a Distributed-Logic Computer," *Proc. of 1962 Fall Joint Computer Conference.*

[312] C.Y. Lee, and M.C. Paul, "A Content-Addressable Distributed-Logic Memory with Applications to Information Retrieval," *IEE Proceedings*, 51, pp. 924-932, 1963.

[313] H-P.S. Lee, "An Algorithm for Minimal TANT Network Generation," *IEEE Trans. Comp.* Vol. C-27, No. 12, Dec. 1978, pp. 1202-1206.

[314] R.S. Ledley, "Mathematical Foundations and Computational Methods for a Digital Logic Machine," *J. Ops. Res. Soc. Amer.,* Vol. 2, pp. 249-274, 1954.

[315] R.S. Ledley, "Digital Computational Methods in Symbolic Logic, with Examples in Biochemistry," *Proc. Nat'l. Acad. Sci.,* Vol. 41., pp. 498-511, July 1955.

[316] R.S. Ledley, "Logical Aid to Systematic Medical Diagnosis (and Operational Simulation in Medicine)," *J. Ops. Res. Soc. Amer.,* Vol. 4, No.3., p. 392, Aug. 1956.

[317] R.S. Ledley, "Reasoning Foundations of Medical Diagnosis," *Science,* Vol. 130, No. 3366, pp. 9-21, 3 July 1959.

[318] R.S. Ledley, "Digital Computer and Control Engineering," *McGraw-Hill Book Co.,* New York, 1960.

[319] R.S. Ledley, 'Use of Computers in Biology and Medicine," *McGraw-Hill Book Co.,* New York, 1960.

[320] G. Leibniz, *"On the Art of Combination,"* in "Parkinson Logical Papers."

[321] Ch.E. Leiserson, "FAT-TREES: Universal Networks for Hardware-Efficient Supercomputing," *1985 International Conference on Parallel Processing,* IEEE Computer Society, August 1985.

[322]

[323] A. Linde, T. Nordstroem, and M. Taveniku, "Using FPGAs to Implement a Reconfigurable Highly Parallel Computer," in *FPL '92.*

[324] L. Loewenheim, "Uber die Aufloesung von Gleichungen im Logischen Gebietekalkul," *Math. Ann.* Vol. 68., 1910, pp. 169-207. *Translation:* "The Solution of Equations in the Calculus of Logic", AFCRL-69-0149, Air Force Cambridge Research Laboratories, April 1969.

[325] D. Luckham, "The Resolution Principle in Theorem-Proving," *Machine Intelligence I,* (N.L. Collins and D. Michie, Eds.), *Oliver & Boyd,* Edinburgh & London, 1967.

[326]

[327] W. Luk, and I. Page, "Parametrising Designs for FPGAs," in *FPGAs,* pp. 285-295.

[328] Raymon Lullus, (a.k.a. Ramon Llull), "Ars Major", "Ars Generalis", "Ars Veritatis Inventiva".

[329] G. Mago, "A Network of Microprocessors to Execute Reduction Languages, Part I," *Intern. Journal of Computer and Information Sciences*, 8, 5, pp. 349-385.

[330] J. Maluszynski, and H.J. Komorowski, "Unification-free Execution of Logic Programs," *Proc. of the 1985 Symposium on Logic Programming,* July 15-18, Boston, MA, pp. 78-86, 1985,

[331] R. Malvi, "The Column Multiplicity Problem in Decomposition of Functions and Relations," M.S. Thesis, PSU, 1997.

[332] E. Marczewski, "Independence in Algebras of Sets and Boolean Algebras," *Fundamenta Mathematicae,* Vol. 48, pp. 135-145, 1960.

[333]

[334] M.A. Marin, "Investigation of the Field of Problems for the Boolean Analyzer," *Ph.D. Dissertation,* Univ. of California, Los Angeles, 1971. early version in Report No. 68-28.

[335] A. Marquand, "A Logical Diagram for N terms," *Philosophical Magazine,* Vol. 12., pp. 266-270, 1881.

[336] A.J. Martin, "A Synthesis Method for Self-Timed VLSI Circuits," *Proc. 15th Annual International Symposium of Computer Architecture,* May 30-June 2, 1988, Honolulu, Hawaii, pp. 224-229.

[337] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus, "The First Asynchronous Microprocessor: The Test Results," *Proc. 16th Annual International Symposium of Computer Architecture,* May 28-June 1, 1989, Jerusalem, Israel, pp. 95-110.

[338] U. Martin, and T. Nipkow, "Unification in Boolean Rings," *Journal of Automated Reasoning,* Vol. 4., pp. 381-396, 1988.

[339] U. Martin, and T. Nipkow, "Boolean Unification - the Story so Far," *Journal of Symbolic Computation,* Vol. 7., pp. 275-293, 1989.

[340] W. Mays, and D. G. Prinz, "A Relay Machine for the Demonstration of Symbolic Logic," *Nature,* 165, 1950, p.197.

[341] W. Mays, "Note on the Exhibition of Logical Machines," *Mind,* Vol. 60, April 1951, p. 262.

[342] W. Mays, and D. Henry, "Exhibition of the Work of W. Stanley Jevons," *Nature,* 170, 1952, pp.696-697.

[343] W. Mays, and D. Henry, "Jevons and Logic," *Mind,* LXII, 1953, pp.484-505.

[344] S. Mazor and P. Langstraat, *A Guide to VHDL*, 2nd edition, Kluwer Academic Publishers, Boston, 1993.

[345] H. McCall, "Symbolic Reasoning," *Mind,* 1880, p. 46.

[346]

[347] J.T. McCall, J.G. Tront, F.G. Gray, R.T. Haralick, and W.M. McCormack, "Parallel Computer Architectures and Problem Solving Strategies for the Consistent Labeling Problem," *IEEE TC.,* Vol. C-34, No. 11, Nov. 1985.

[348] E. J. McCluskey, Jr., "Minimization of boolean functions," *Bell System Tech. J.,* Vol. 35, pp. 1417-1443, Apr. 1956.

[349] C.J.H. McDiarmid, "Determining the Chromatic Number of a Graph," *SIAM J. Computing,* Vol. 8, pp. 1-19. 1979.

[350] C. Mead, and L. Convay, "Introduction to VLSI Systems," *Addison-Wesley*, 1980.

[351] Mentor, Patent for emulation machine, 1992.

[352] Mentor Graphics, *QuickHDL Release Notes*, Software Version 8.5_4.6c, Mentor Graphics Corporation, Wilsonville, OR, 1996.

[353] Mentor Graphics, *QuickHDL User's and Reference Manual*, Software Version 8.5_4.6c, Mentor Graphics Corporation, Wilsonville, OR, 1996.

[354] R.E. Michalski, and R.E. Flick, "Automated Construction of Classifications: Conceptual Clustering Versus Numerical Taxonomy," *Pattern Matching and Machine Intelligence*, July 1983, pp. 410-416, 1983.

[355] R.E. Miller, *"Switching Theory",* Vol. 1 and 2, John Wiley, New York, 1965.

[356] M. Minsky, "Computation: Finite and Infinite Machines," *Prentice-Hall*, 1967.

[357] M. Minsky, and S. Papert, "Perceptrons," MIT Press, second edition, 1972.

[358] M. Minsky, "K-Lines: A Theory of Memory," *Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo 514*, Reprinted in Cognitive Science, 1980, pp. 117-133.

[359] D. Mithani, "Implementation of NAND Synthesis Using Implicants Containing Output Variables," *M.S. Thesis,* Dept. Electr. Engng., Univ. of Wisconsin, 1977.

[360] S. Mohamed, M.A. Perkowski, and L. Jozwiak, "Fast Approximate Minimization of Multi-Output Boolean Functions in Sum-of-Product Condition-Decoders Structures," *Proc. Euromicro '97*, September 1997.

[361] E.F. Moore, "Sequential Machines: Selected Papers," *Addison-Wesley*, 1964.

[362]

[363] W. Moore, and W. Luk, "FPGAs," Edited from the Oxford 1991 International Workshop on Field Programmable Logic and Applications, *Abingdon EE & CS Books,* Abingdon England,1991.

[364] E. Morreale, "Recursive operators for prime implicant and irredundant normal form determination," *IEEE Trans. on Computers,* Vol. C-19, p. 504, 1970.

[365] P. and E. Morrison, *"Calculating Engines,"*

[366]

[367] P. Morrison, and J. Morrison, (Eds.), *"Charles Babbage and His Calculating Engines,"* New York: Dover Publications, 1961.

[368] Motorola, "MPA10XX Field Programmable Gate Array," *Product Brief,* 9/27/93.

[369] R. Murgai, N. Shenoy, R.K. Brayton, and A. L. Sangiovanni-Vincentelli, "Improved Logic Synthesis Algorithms for Table Look Up Architectures," *Proc. ICCAD-91,* pp. 564 - 567, Nov. 1991.

[370] T.A. Nodes, J.L. Smith, and R. Hecht-Nielsen, "A Fuzzy Associative Memory Module and its Applications to Signal Processing," *Proc. Intern., Conf. on Acoustics Speech and Signal Processing, ICASSP,* pp. 1511-1514, New York, 1985.

[371] A. Newell, J.C. Shaw, and H.A. Simon, "Empirical Explorations with the Logic Theory Machine: A Case Study in Heuristics," *Proceedings of the Western Joint Computer Conference,* 15, 1957, in Feigenbaum and Feldman, "Computers and Thought", p. 115.

[372] A. Newell, J.C. Shaw, and H.A. Simon, "The Logic Theory Machine," in Feigenbaum and Feldman, *"Computers and Thought",* pp. 121 - ??.

[373] A. Newell, and H.A. Simon, "GPS, A Program that Simulates Human Thought," in Feigenbaum and Feldman, *"Computers and Thought",* p. 286.

[374]

[375]

[376] L. Nguyen, M. Perkowski, and N. Goldstein, "PALMINI - Fast Boolean Minimizer for Personal Computers," *Proc. of the 24th Design Automation Conference,* June 28 - July 1, 1987, Miami, Florida, Paper 33.3, pp. 615-621.

[377]

[378] J. Nievergelt, J.C. Farrar, and E.M. Reingold, *"Computer Approaches to Mathematical Problems,"* Englewood Cliffs, N.J.: Prentice Hall, 1974.

[379] S.E. Orcutt, "Implementation of Permutation Functions in ILLIAC IV-Type Computers," *IEEE Trans. on Computers*, C-25, 9, pp.929-936, 1976.

[380] J.V. Oldfield, "Logic Programms and an Experiment Architecture for their Execution," *IEE Proc. Pt.E.,* Vol. 133, No. 3, pp. 163-167, May 1986.

[381] J.V. Oldfield, Ch. Storman, and M. Brule, "The Application of VLSI Content-Addressable Memories to the Acceleration of Logic Programming Systems," *Proc. of VLSI and Computers, First Intern. Conf. on Computer Technology, Systems, and Applications,* pp. 27-30, IEEE, Hamburg, May 11-15, 1987.

[382] J.V. Oldfield, R.D. Williams, and N.E. Wiseman, "Content-Addressable Memories for Storing and Processing Recursively Subdivided Images and Trees," *Electronic Letters,* Vol. 23, No. 6, pp. 262-263, 12 March 1987.

[383] R.A. O'Keefe, "A Comment on "A Hardware Unification Unit: Design and Analysis"," *Computer Architecture News,* Vol. 14, No. 1, pp. 2-3, January 1986.

[384] S.A. Ozkarahan, S.A. Schuster, and K.C. Sevcik, "A Data Base Processor," Tech. Rep. CSRG-43, *Computer Systems Research Group,* University of Toronto, 1974.

[385] S.A. Ozkarahan, S.A., Schuster, and K.C. Sevcik, "Performance Evaluation of a Relational Associative Processor," *ACM Trans. Database Systems*, 2, 2, 1977.

[386]

[387] I. Page, and W. Luk, "Compiling OCCAM into FPGAs," in *FPGAs,* pp. 271-283.

[388] Ch.A. Papachristou, "Content-Addressable Memory Requirements for Multivalued Logic," *11th Int. Symp. on Multi-Valued Logic,* pp. 62-72, 1981.

[389] B. Parhami, "Rapid: A Rotating Associative Processor for Information Dissemination," UCLA-ENG-7213, *University of California*, Los Angeles.

[390] Paris Research Liboratory, Digital Equipment Corp., *DECPeRLe-1 Introductory Tutorial*, April 1993.

[391] P.P. Parkhomenko, "Automatizing the Process of Analyzing Relay Contact Networks," *Vses. Mos. Sov. Teorii Ust. Rel. Deist.,* p. 70, 1957.

[392] P.P. Parkhomenko, "A Large Capacity Logic Machine for Relay Circuit Analysis," *Izv. A.N. O.T.N. E i A.,* No. 3., pp. 179-180, 1959.

[393] P.P. Parkhomenko, "Machine Analysis of Relay Networks," *Avt. i Telem.* Vol. 20, No. 4., pp. 486-497, 1959.

[394] P.P. Parkhomenko, "The Principles of Mechanization of the Analysis of Relay-Contact Networks," *Dokl. A.N.,* Vol. 124, No. 1., pp. 83-86, 1959.

[395] P.P. Parkhomenko, "A Large Capacity General-Purpose Computer for the Analysis of Relay Networks," *Teoriya i Prim. Disk. Avt. Sistem,* 1960.

[396] C.S. Peirce, ed., "Studies in Logic," *By Members of the Johns Hopkins University, Little Brown & Co.,* Boston, 1883.

[397] C.S. Peirce, "Logical Machines," *Amer. J. Psychology,* Vol. 1. pp. 165-170, 1887.

[398] M. Perkowski, "Relational-structure languages and their application in the system for automatic design of block synthesis of digital systems of automatic control," *Proceedings of the 6th National Conference on Automatic Control,* Poznan, Poland, 9-11 September 1974, Vol. 1, pp. 695-707 (in Polish).

[399] M. Perkowski, "An example of heuristic programming application in the three-level combinational logic design," *Proceedings of the 3rd Symposium on Heuristic Methods,* Polish Cybernetical Society, Warsaw, 25 September 1976, Vol. 1. pp. 105-132.

[400] M. Perkowski, "Synthesis of multioutput three level NAND networks," *Proc on the Seminar on Computer Aided Design,"* Budapest 3-5, November 1976, pp.238-265.

[401] M. Perkowski, "Some concepts on reasoning by analogy. The heuristic programming approach," *Proceedings of IVth International Symposium on Heuristic Methods,* Warsaw, 24 September, 1977, Vol. 2, pp. 25-119.

[402] M. Perkowski, A. Rydzewski, and P. Misiurewicz, "Theory of Logic Circuits, Selected Problems," *Publishers of the Technical University of Warsaw,* Ed. 1 - 1977, Ed. 2 - 1978, Ed. 3 - 1984 (in Polish).

[403] M. Perkowski, "An application of general problem-solving methods in computer-aided design: the MULTICOMP system and its problem-oriented source language," *Proceedings of IVth International Symposium on Heuristic Methods,* Warsaw, 24 September, 1977, Vol. 3, pp. 55-102.

[404] M. Perkowski, "The state-space approach to the design of multipurpose problem-solver for logic design" *"Artificial Intelligence and Pattern Recognition in Computer-Aided Design,"* J. C. Latombe (ed.), North Holland, Amsterdam, pp. 124-140, 1978. (book chapter).

[405] M. Perkowski, "The state-space approach to the design of multipurpose Conference "Artificial Intelligence and Pattern Recognition in Computer-Aided Design," Grenoble, France, 17-19 March 1978, J. C. Latombe (ed.) North Holland, Amsterdam, pp. 124-140, 1978.

[406] M. Perkowski, and M. Gluch, "Knowledge Engineering Language Using Strategies (KELUS)," *Institute of Automatic Control, Technical University of Warsaw,* 1979.

[407] M. Perkowski, and A. Zasowska, "Minimal area MOS asynchronous automata," Proceedings of the International Symposium on Applied Aspects of Automata Theory, Warna, Bulgaria, 14-19 May 1979, pp. 284-298.

[408]

[409] M. Perkowski, "The method of solving combinatorial problems in the automatic design of digital systems," Institute of Automatic Control, Technical University of Warsaw, Ph.D. Thesis, 1980, (in Polish).

[410] M. Perkowski, "Multistrategical Problem Solver," *Proceedings of the 2nd International Meeting on Intelligent Robotics and Knowledge ¿¿¿¿¿¿¿*

[411]

[412]

[413] M. Perkowski, "General methods for solving combinational problems," Chapter 4 in A. Goralski (ed.) "Problem, method, solution", Vol. 4, *Scientific-Technical Publishers*, Poland, Warszawa 1982.

[414] M. Perkowski, A Goralski, and G. Zielinski, "Elements of Artificial Intelligence," Book in Polish. Preprint 1982.

[415] M. Perkowski, B. Goldstein, "A new algorithm for multioutput Boolean functions minimization based on reduction to graph coloring," University of Minnesota, Department of Electrical Engineering, *report.*

[416] M. Perkowski, "Fast Approach to Multiple-Valued Constraints Problem: Systolic VLSI Realization of System MULTICOMP," *PSU Report*, December 1984.

[417] M. Perkowski, "Design of Finite State Machines with SuperPeg," *Seminar of Department of Electrical Engineering,* Oregon State University, Corvallis, Oregon, February 12, 1985.

[418]

[419] M. Perkowski, "A Systolic Processor for Approximate Solutions to NP-Complete Combinatorial Problems," *Symposium on Complexity of Approximately Solved Problems,* Computer Science Department, Columbia University, New York, NY 10027, April 17 -19 , 1985.

[420]

[421]

[422] M.A. Perkowski, "Systolic Architecture for the Logic Design Machine," *Proceedings of the IEEE and ACM International Conference on Computer Aided Design - ICCAD 85,* Santa Clara, 19-21 November 1985, pp. 133-135.

[423] M. Perkowski, "Reduction of Problems for the Logic Design Machine," Portland State University, *report.*

[424]

[425] M. Perkowski, N.B. Goldstein, and L.B. Nguyen, "PLA with tail gives estimation from lower bound and has a decreased area," *Preprint*, PSU, 1985.

[426] M. Perkowski, and N. Nguyen, "Minimization of Finite State Machines in SuperPeg," *Proceedings of the Midwest Symposium on Circuits and Systems,* Luisville, Kentucky, 22-24 August 1985.

[427] M. Perkowski, "On the Reduction of Combinatorial Problems and Parallel Computers for Solving Generic Combinatorial Problems," *System Science Seminar PSU,* February 11, 1986.

[428] M. Perkowski, "Logic Design Machine," *Carnegie Mellon University,* Department of Electrical Engineering, Invited Lecture, 1986.

[429] M. Perkowski, D. Smith, and R. Krzywiec, "Logic Simulation/Design/Verification Environment in Prolog," *Proc. of the 17th Annual Pittsburgh Conference on Modelling and Simulation,* 24-25 April, 1986, University of Pittsburgh, Pittsburgh, Pennsylvania, pp. 945-958.

[430] M. Perkowski, "Parallel Programs for Tautology, Satisfiability, and Complementation of Boolean Functions," *INTEL Scientific Computers Report*, September 1986.

[431] M. Perkowski, H. Uong, H. Uong, "Automatic Design of Finite State Machines with Electronically Programmable Devices," *Record of Northcon 87,* Portland 1987, paper 13/4.

[432] M. Perkowski, and J. Liu, "A System for Fast Prototyping of Logic Design Programs," *Proc. of the 30th Midwest Symposium on Circuits and Systems,* Syracuse, New York, August 16-18, 1987, paper TA 2.6.3.

[433] M. Perkowski, and J.E. Brown, "An Unified Approach to Designs Implemented with Multiplexers and to the Decomposition of Boolean Functions," *Proc. of the 1988 ASEE National Conference,* Portland, Oregon, June 19-23, 1988, pp. 1610-1619.

[434]

[435]

[436] M. Perkowski, J. Liu, and J. Brown, "Quick Software Prototyping: CAD Design of Digital CAD Algorithms," *In G. Zobrist (ed) "Progress in Computer Aided VLSI Design",* Vol. 1., Ablex Publishing Corp., 1989, pp. 353-401.

[437] M.A. Perkowski, P. Wu, and K.A. Pirkl, "KUAI-EXACT: A New Approach for Multi-Valued Logic Minimization in VLSI Synthesis," *Proc. of the 1989 ISCAS - International Symposium on Circuits and Systems,* May 9-11, 1989, pp. 401-404.

[438] M.A. Perkowski, M. Helliwell and P. Wu, "Minimization of Multiple-valued Input Multi-output Mixed-radix Exclusive Sum of Products for Incompletely Specified Boolean Functions," *Proc. of the 19th International Symposium on Multiple-Valued Logic,* pp. 256-263, May 1989.

[439] M.A. Perkowski, Martin Helliwell, Pan Wu, "Minimization Of Multiple Valued Input Multi-Output Mixed-Polarity Generalized Reed-Muller Forms For Incompletely Specified Boolean Functions," *Report,* Full Version, Electrical Engn. Dept., Portland State University, 1989.

[440] M.A. Perkowski, and J. Brandenberg, "Solving Basic Boolean Algebra Problems on a Hypercube Computer," *Report PSU*, 1989.

[441] M.A. Perkowski, M. Driscoll, J. Liu*, D. Smith*, J. Brown**, L. Yang*, A. Shamsapour*, M. Helliwell**, B. Falkowski*, and A. Sarabi*, "Integration of Logic Synthesis and High-Level Synthesis into the DIADES Design Automation System," *Proc. of the 1989 ISCAS - International Symposium on Circuits and Systems,* May 9-11, Portland, OR, 1989, pp. 748-751.

[442] M.A. Perkowski, P. Dysko, and B.J. Falkowski, "Two Learning Methods for a Tree-Search Combinatorial Optimizer," *Proceedings of IEEE International Phoenix Conference on Computers and Communication,* Scottsdale, Arizona, March 1990, pp. 606-613.

[443] M.A. Perkowski, and J. Liu, "Generation of Finite State Machines from Parallel Program Graphs in DIADES," *Proc. of the ISCAS'90, International Symposium on Circuits and Systems,* New Orleans, 1-3 May 1990, pp. 1139-1142.

[444] M.A. Perkowski, and J. Liu, "A Program for Synthesis of Three Level NAND Networks," *Proc. of the ISCAS'90, International Symposium on Circuits and Systems,* New Orleans, 1-3 May 1990, pp. 1118-1121.

[445]

[446]

[447]

[448] M.A. Perkowski, and M. Chrzanowska-Jeske, "An Exact Algorithm to Minimize Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions," *Proc. of the ISCAS'90, International Symposium on Circuits and Systems,* New Orleans, 1-3 May 1990, pp. 1652-1655.

[449] M. Perkowski, M. Chrzanowska-Jeske, and T. Shah, "Minimization of Multioutput TANT Networks for Unlimited Fan-In Network Model," *Proc. IEEE International Conference on Computer Design,* Cambridge, Massachusetts, pp. 360-363, September 17-19, 1990.

[450] M. Perkowski, L. B. Nguyen*, "The Encoding Program for Concurrent Finite State Machines Realized Using PLD Devices," *Proc. of the Midw. Symp. Circ. Syst., Aug. 1990, Alberta, CA, pp. 204-207.*

[451] M.A. Perkowski, A.Sarabi, and I. Schaefer, "Aplication of Orthogonal Transforms in Image Processing," *Proc. of Northcon'91,* Portland, 1-3 October 1991, Session S1, pp. 303-309.

[452] M. A. Perkowski, and P. Johnson, "Canonical Multivalued-Input Reed-Muller Trees Expressions," *Proc. of the Third NASA Symposium on VLSI Design,* Moscow, Idaho, October 30-31, 1991.

[453] M.A. Perkowski and students, "The Multiple Valued Cube Calculus Machine, CCM2," *Report,* EE Dept., Portland State University, 1991.

[454] M.A. Perkowski and Pan Wu, "A new Approach To Exact Minimization Of Boolean Functions With Multiple Valued Inputs," *Diades research group report*, Portland State University, 1991.

[455]

[456]

[457] M.A. Perkowski, "A Universal Logic Machine," *Proc. of the IEEE ISMVL'92, the 21st International Symposium on Multiple-Valued Logic,* Sendai, Japan, May 27-29, 1992, pp. 262-271, invited address.

[458] M.A. Perkowski, "The Generalized Orthonormal Expansion of Functions with Multiple-Valued Inputs and Some of its Applications," *Proc. of the ISMVL'92, the 21st IEEE International Symposium on Multiple-Valued Logic,* Sendai, Japan, May 27-29, 1992, pp. 442-450.

[459] M.A. Perkowski, W. Zhao, and D. Hall, "Concurrent Two-Dimensional State Minimization and State Assignment of Finite State Machines," *Proc. IEEE International Conference "VLSI Design '92",* Bangalore, India, January 4-7, 1992.

[460] M.A. Perkowski, A. Coppola, "A State Machine PLD and Associated Minimization Algorithms," *Proc. of the FPGA'92, 1992 ACM/SIGDA First International Workshop on Field-Programmable Gate Arrays,* pp. 109 - 114, Berkeley, February 16-18, 1992.

[461] M. Perkowski, "A Fundamental Theorem for Exor Circuits," *Proc. Reed-Muller'93*, pp. 52-60.

[462] M. Perkowski, A.Sarabi, F. Beyl, "XOR Canonical Forms of Switching Functions," *Proc. of Reed-Muller'93*, pp. 27-32.

[463] M. Perkowski, A.Sarabi, and F. Beyl, "Fundamental Theorems and Families of Forms for Binary and Multiple-Valued Linearly Independent Logic," *Proc. Reed-Muller'95*, pp. 288-299.

[464] Marek A. Perkowski, Stainslaw Grygiel, and the Functional Decomposition Group, "A Survey of Literacture on Function Decomposition", Version iv., Nov 20, 1995.

[465] M.A. Perkowski, T. Luba, S. Grygiel, P. Burkey, M. Burns, N. Iliev, M. Kolsteren, R. Lisanke, R. Malvi, Z. Wang, H. Wu, F. Yang, S. Zhou, and J.S. Zhang, "Unified Approach to Functional Decompositions of Switching Functions," *Technical Report*, Portland State University, 1995.

[466] M.A. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J.S. Zhang, "Decomposition of Multi-Valued Relations," *Proc. of the IEEE ISMVL'97,* Halifax Nova Scotia, May 1997, pp. 13-18.

[467] M.A. Perkowski, "ULM-Lisp, the tutorial.", 1997.

[468] M.A. Perkowski, and L. Jozwiak, "Two-Dimensional State Minimization of Finite State Machines," submitted.

[469] M. Perkowski, L. Jozwiak, and D. Foote, "Architecture of a Programmable FPGA Co-processor for Constructive Induction Approach to Machine Learning and other Discrete Optimization Problems," *Proc. 4th Reconfigurable Architectures Workshop,* Geneva, Svitzerland, April 1-5, 1997.

[470] M. Perkowski, L. Jozwiak, and R. Drechsler, "A Canonical AND/EXOR Form that includes both the Generalized Reed-Muller Forms and Kronecker Reed-Muller Forms," *Proc. Reed-Muller'97.*

[471] M.A. Perkowski, L. Jozwiak, and R. Drechsler, "Two hierarchies of Generalized Kronecker Trees, Forms, Decision Diagrams, and Regular Layouts," *Proc. Reed-Muller'97.*

[472] M. Perkowski, L. Jozwiak, R. Drechsler, and B. Falkowski, "Ordered and Shared, Linearly Independent, Variable-Pair Decision Diagrams for Incompletely Specified Functions," *Proc. ICICS'97*,1997.

[473] M. Perkowski, B. Steinbach, "Efficient Exact Minimization of Incomplete AND/EXOR Forms," *in preparation*, 1997.

[474] Perkowski, M.A., "Synthesis of Multioutput Three Level NAND Networks", *Proc. of the Seminar on Computer Aided Design,* Budapest, 3-5 Nov. 1976, pp. 238-265.

[475] Perkowski, M. A., Chrzanowska-Jeske, M., and T. Shah, "Minimization of Multioutput TANT Networks for Unlimited Fan-In Network Model," *Proc. ICCD'90*, pp. 360 - 363, Boston, MA, September 1990.

[476] Perkowski, M.A., and M. Chrzanowska-Jeske, "An Exact Algorithm to Minimize Mixed-Radix Exclusive Sums of Products for Incompletely Specifed Boolean Functions," *Proc. ISCAS'90,* pp. 1652 - 1655, New Orleans, May 1-3, 1990.

[477] Perkowski, M. A., "The Generalized Orthonormal Expansion of Functions with Multiple-Valued Inputs and Some of its Applications," *Proc. ISMVL'92,* pp. 442 - 450, Sendai, Japan, May 27-29, 1992.

[478] Perkowski, M. A., "A Fundamental Theorem for EXOR Circuits," *Proc. of IFIP W.G. 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design,"* Hamburg, Germany, Sept. 16-17, pp. 52 - 60, 1993.

[479] Perkowski, M. A., Sarabi, A., and F. R. Beyl, "Universal XOR Canonical Forms of Switching Functions," *Proc. of IFIP W.G. 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design,"* Hamburg, Germany, Sept. 16-17, pp. 27 - 32, 1993.

[480] M.A. Perkowski, Book on Functional Programming Approach to Finite State Machines, Winter 1998.

[481] M. A. Perkowski, M. Helliwell and P. Wu, *Minimization of Multiple-Valued Input Multi-Output Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions*, Proc. 19th Int. Symp. on Multiple-Valued Logic, pp. 256-263, May 1989.

[482] M. A. Perkowski and M. Chrzanowska-Jeske, *An Exact Algorithm to Minimize Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions*, Proc. of the ISCAS'90, International Symposium on Circuits and Systems, pp. 1652-1655, New Orleans, May 1990.

[483] M. A. Perkowski, T. Ross, D. Gadd, J.A. Goldman, and N. Song, *Application of ESOP Minimization in Machine Learning and Knowledge Discovery,* Proc. of the Second Workshop on Applications of Reed-Muller Expansion in Circuit Design, Chiba City, Japan, 27-29 August 1995, pp. 102-109.

[484] M. Perkowski, A.Sarabi, and F. Beyl, *Fundamental Theorems and Families of Forms for Binary and Multiple-Valued Linearly Independent Logic*, Proc. Reed-Muller'95, pp. 288-299.

[485] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J. Zhang, *Decomposition of Multi-Valued Relations,* Proc. ISMVL'97, Nova Scotia, May 1997.

[486] M. Perkowski, L. Jozwiak, R. Drechsler, and B. Falkowski, *Ordered and Shared, Linearly Independent, Variable-Pair Decision Diagrams for Incompletely Specified Functions,* submitted.

[487] M. Perkowski, L. Jozwiak, and R. Drechsler, *A Canonical AND/EXOR Form that includes both the Generalized Reed-Muller Forms and Kronecker Reed-Muller Forms,* submitted to RM'97.

[488] M.A. Perkowski, M. Chrzanowska-Jeske, and Y. Xu, *Lattice Diagrams for Reed-Muller Logic Realization in Regular Layouts with Predictable Timing,* submitted to Reed-Muller'97.

[489] M.A. Perkowski, P. Wu, and K.A. Pirkl, "KUAI-exact: A new approach for multi-valued logic minimization in VLSI synthesis", *Proc. ISCAS*, pp. 401-404, May 1989.

[490]

[491] M.A. Perkowski, M.E. Chrzanowska-Jeske, E. Pierzchala, and A. Coppola, "An Exact Solution to the Fitting Problem in the Application Specific State Machine Device," *J. of Circuits, Systems, and Computers,* Vol. 4, No. 2, pp. 173-190, June 1994.

[492] M.A. Perkowski, and A. Coppola, "A State Machine PLD and Associated Minimization Algorithms", *Proc. of FPGA'92,* Berkeley, pp. 109-114, February 1992.

[493] M.A. Perkowski, T. Ross, D. Gadd, J.A. Goldman, and N. Song, "Application of ESOP Minimization in Machine Learning and Knowledge Discovery," *Proc. Reed-Muller'95*, pp. 102-109, September 1995.

[494] M.A. Perkowski, E. Pierzchala, and R. Drechsler, *Logic/Layout Synthesis for a Submicron Technology: Mapping Linearly-Independent Expansions to Fat Regular Lattices,* submitted.

[495] M.A. Perkowski, L. Jozwiak, R. Drechsler, *Two Hierarchies of Generalized Kronecker Trees, Forms, Decision Diagrams, and Regular Layouts,* submitted to RM'97.

[496] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and Jin S. Zhang, "Decomposition of Multiple-Valued Relations," *Proc. ISMVL'97 conference*, pp. 13-18, May 1997.

[497] C. Peterson, "iWARP", *HOT Chips Symposium Record,* Santa Clara, CA, August 20-21, 1990.

[498] S.R. Petrick, "On the Minimization of Boolean Functions," *Proc. Symp. on Switch. Theory,* IFIP, Paris, June 1959.

[499] PiE Design, *company broschures*, 1992.

[500] P. Poretsky, "On the Methods for Solving Logical Equations and on the Inverse Method for Mathematical Logic," (In Russian), *Bull. de la Soc. Physico-Mathematique de Kasan,* Vol. 2., pp. 161-230, 1884.

[501] G.N. Povarov, "A Method for the Synthesis of Computing and Control Contact Networks," *Avt. i Telem.,* Vol. 18., No 2., pp. 145-162, 1957.

[502] D.K. Pradhan, *"Fault-Tolerant Computing. Theory and Techniques. Vol. I.,"* Prentice-Hall, 1987.

[503] V. Pratt, "Thinking Machines. The Evolution of Artificial Intelligence," *Basil Blackwell, Inc.,* Oxford, U.K., 1987.

[504] *Proceedings of the Workshop on FPGA custom architectures,* Napa Valey, California, April 1993.

[505] *Proceedings of ACM FPGA'92 Workshop,* Hotel Durant, Berkeley, 16-18 Febr. 1992.

[506] *Proceedings of Workshop of Field-Programmable Logic and Applications,* FPL'92, Vienna, Austria, 31 August - 2 September 1992.

[507] *Proceedings of ACM FPGA'92 Workshop,* Hotel Durant, Berkeley, 16-18 Febr. 1992.

[508] *Proceedings of Workshop of Field-Programmable Logic and Applications,* FPL'92, Vienna, Austria, 31 August - 2 September 1992.

[509] Q. Chen, *M.S. Thesis,* 1998.

[510] Quickturn Corp., Company materials, 1991.

[511] W.V. Quine, "The problem of simplifying truth functions," *Amer. Math. Monthly,* Vol. 59, p. 521, 1952.

[512] W.V. Quine, "A way to simplify truth functions," *Amer. Math. Monthly,* Vol. 62, p. 627, 1955.

[513] M.R. Quillian, "Semantic Memory", in "Semantic Information Processing," M. Minsky (ed.), *MIT Press*, pp. 227-270, 1968.

[514] J.R. Quinlan, "C4.5: Programs for machine learning," San Mateo, CA: *Morgann Kaufmann.*

[515] M.O. Rabin, "Probabilistic Algorithms," In J.Traub (ed.) *Algorithms and Complexity: New Directions and Recent Results,* Academic Press, New York, 1976, pp.21-39.

[516] C.V. Ramamoorthy, J.L. Turner, and B. W. Wah, "A Design of Fast Cellular Associative Memory for Ordered Retrieval," *IEEE Trans. on Computers,* Vol. C-27, No. 9, pp. 800-815, September 1978.

[517] Randell, "On Alan Turing and the Origins of Digital Computers" in B, Melzer and D. Michie (eds) Machine Intelligence 7, Edinburgh University Press, Edinburgh, 1972.

[518] B. Randell, *"The Origins of Digital Computers,"* Springer-Verlag, Berlin, 1975.

[519] A.P. Reeves, "Parallel Computer Architectures for Image Processing," *Proc. of the Intern. Conf. on Parallel Processing,* pp. 199-206, 1981.

[520] V.T. Rhyne, P.S. Noe, M.N. McKinny and U.W. Pooch, "A new technique for the fast minimization of switching function," *IEEE Trans. on Computers,* Vol. C-24, No. 9, pp. 926-930, Aug. 1977. *IBM Thomas J. Watson Research Center,* Yorktown Heights, New York, Jan. 1972.

[521] J.C. Ribeiro, C.D. Stormaon, J.V. Oldfield, and M.R. Brule, "Content-Addressable Memories Applied to Execution of Logic Programs," *IEE Proceedings,* Vol. 136, Pt. E., No. 5, pp. 383-388, September 1989.

[522] C. Rieger, "ZMOB: A Mob of 256 Cooperative Z80A-Based Microcomputers," *Computer Science Tech. Rep.*, Series TR-825, University of Maryland, College Park, MD, 1979.

[523] C. Rieger, J. Bane, and R. Trigg, "ZMOB: A Highly Parallel Multiprocessor," *Tech. Rep. TR-911, Dept. Comp. Sci,* University of Maryland, College Park, MD, 1980.

[524]

[525] D. Rine, "Computer Science and Multiple-Valued Logic. Theory and Applications," *North-Holland,* 1984.

[526]

[527] B. Reusch, and L. Detering, "On the Generation of Prime Implicants," *Annales Societatis Mathematicae Polonae,* Series IV: Fundamenta Informaticae II, pp. 167-186, 1979.

[528] J.A. Robinson, "A Machine Oriented Logic Based on the Resolution Principle," *J. of the Association for Computing Machinery,* Vol. 12., No. 1., pp. 23-41, January 1965.

[529] P. Robinson, "The SUM: an AI Coprocessor," *BYTE,* pp. 169-180, June 1985.

[530] V.I. Rodin, "An Electronic Analyzer for Contact Networks," *Avt. i Telem.,* Vol. 18, No. 5., pp. 437-443, 1957.

[531] V.N. Roginskii, "A Machine for the Design of Relay Networks," *Vestnik Akad. Nauk S.S.S.R.,* No. 10, pp. 71-73, 1958.

[532] T.D. Ross, M.J. Noviskey, T.N. Taylor, and D.A. Gadd, "Pattern Theory: An Engineering Paradigm for Algorithm Design," *Final Technical Report WL-TR-91-1060*, Wright Laboratories, USAF, WL/AART/WPAFB, OH 45433-6543, August 1991.

[533] C.H. Roth, *Fundamentals of Logic Design*, 4th edition, West Publishing Company, St. Paul, 1992.

[534] J.P. Roth, "A calculus and an algorithm for the multiple-output 2-level minimization problem," Research Report RC 2007, *IBM Thomas J. Watson Research Center,* Yorktown Heights, New York, Feb. 1968.

[535] J.P. Roth, "Theory of cubical complexes with application to diagnosis and algorithmic description," Research Report RC 3675,

[536]

[537] J.P. Roth, *Computer Logic, Testing, and Verification*, Computer Science Press, Potomac Maryland, 1980.

[538] S. Rudeanu, "Boolean Equations and their Applications to the Study of Bridge Circuits I," *Bull. Math. Soc. Math. Phys. R. P. Roumaine,* Vol. 3., pp. 445-473, 1959.

[539] S. Rudeanu, "Boolean Functions and Equations," *North-Holland Publ. Co. & American Elsevier,* New York, 1974.

[540] R.L. Rudell, "ESPRESSO-IIC User's Manual," CAD Toolbox User's Manual, U.C. Berkeley, Jan. 1984.

[541]

[542] R.L. Rudell and A.L. Sangiovanni-Vincentelli, "ESPRESSO-MV: algorithms for multiple-valued logic minimization," *Proc. IEEE Custom Integrated Circuits Conf.,* pp. 230-234, 1985.

[543]

[544] R. Rudell, *"Multiple-Valued Logic Minimization for PLA Synthesis,"* M.S. Report, June 5, 1986. University of California, Berkeley California 94720.

[545]

[546]

[547] R.L. Rudell and A.L. Sangiovanni-Vincentelli, "Exact minimization of multiple-valued functions for PLA optimization," *Proc. ICCAD'86*, pp. 352-355, November 1986.

[548]

[549] R.L. Rudell, and A.L. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *Proc. Intern. Symp. on Multiple-Valued Logic,* pp. 198-208, May 26-28, Boston, MA, 1987.

[550] R.L. Rudell and A.L. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *IEEE Trans. on Computer-aided Design,* Vol. CAD-6, No. 5, pp. 727-749, Sept. 1987.

[551] B. Russell, *"An Outline of Philosophy,"* Allen and Unwin, London, 1927.

[552] R.A. Rutenbar, T.N. Mudge, and D.E. Atkins, "A Class of Cellular Architectures to Support Physical Design Automation," *IEEE TCAD*, Vol CAD-3, No.4., October 1984, pp.264-278.

[553] A. Sarabi, and M.A. Perkowski, "Fast Exact and Quasi-Minimal Minimization of Highly Testable Fixed-Polarity AND/EXOR Canonical Networks," *Proc. DAC'92*.

[554]

[555] A. Sarabi, N. Song, M. Chrzanowska-Jeske, and M. A. Perkowski, "A Comprehensive Approach to Logic Synthesis and Physical Design for Two-Dimensional Logic Arrays," *Proc. DAC '94,* San Diego, June 1994, pp. 321 - 326.

[556]

[557] T. Sasao, "An application of multiple-valued logic to a design of Programmable Logic Arrays," *Proc. 8th Intern. Symp. on Multiple-Valued Logic,* 1978.

[558] T. Sasao, and H. Terada, "Multiple-Valued Logic and the Design of Programmable Logic Arrays with Decoders," *Proc. of 9th International Symposium on Multiple-Valued Logic,* Bath, England, pp. 27 - 37, 1979.

[559]

[560] T. Sasao, "Multiple-valued decomposition of generalized boolean functions and the complexity of programmable logic arrays," *IEEE Trans. on Computers,* Vol. 30, pp. 635-643, Sept. 1981.

[561]

[562]

[563]

[564] T. Sasao, "Input variable assignment and output phase optimization of PLA's," *IEEE Trans. on Computers,* Vol. 33, No. 10, pp. 879-894, Oct. 1984.

[565]

[566]

[567] T. Sasao, "HART: A Hardware for Logic Minimization and Verification," *Proc. ICCD'85,* pp. 713-718, Oct. 7-10, 1985.

[568] T. Sasao, "An algorithm to derive the complement of a binary function with multiple-valued inputs," *IEEE Trans. on Computers,* Vol. 34, pp. 131-140, Feb. 1985.

[569]

[570]

[571] T. Sasao, "MACDAS: Multi-level AND-OR circuit synthesis using two-variable function generators," *23-rd Design Automation Conference,* Las Vegas, pp. 86-93, June 1986.

[572] T. Sasao, and P. Besslich, "On the Complexity of MOD-2 Sum PLA," *Institute of Electronics and Communication Engineers of Japan,* FTS86-17, pp. 1-8, Nov. 17, 1986.

[573] Sasao, T., and M. Higashida, "A Design Method for Three-Level Logic Circuits", (in Japanese). *The Technical Papers of IEICE Japan,* VLD88-84, Dec. 1988.

[574] T. Sasao, "Application of Multiple-Valued Logic to a Serial Decomposition of PLAs," *Proc. ISMVL '89,* 1989.

[575] Sasao, T., "On the Complexity of Three-Level Logic Circuits", *Proc. Intern. Workshop on Logic Synthesis,* MCNC, ACM SIGDA, May 23-26 1989, paper 10.2.

[576] T. Sasao, "EXMIN: A Simplification Algorithm for Exclusive-OR-Sum-of-Products Expressions for Multiple-Valued Input Two-Valued Output Functions," *Proc. of 20th Int. Symp. on Multiple-Valued Logic,* pp. 128-135, May 1990.

[577] T. Sasao (ed.), "Logic Synthesis and Optimisation," *Kluwer Academic Publishers,* 1993.

[578] T. Sasao, *EXMIN: A Simplification Algorithm for Exclusive-OR-Sum-of-Products Expressions for Multiple-Valued Input Two-Valued Output Functions*, 20th Int. Symp. on Multiple-Valued Logic, pp. 128-135, May 1990.

[579] T. Sasao and Ph. W. Besslich, *On the Complexity of Mod-2 Sum PLAs*, IEEE Trans. on Comput., Vol 39, no. 2, pp. 262-266, February 1990.

[580] T. Sasao, *EXMIN2: A Simplification Algorithm for Exclusive-OR-Sum-of-Products Expressions for Multiple-Valued-Input Two-Valued-Output Functions*, IEEE Trans. on CAD, Vol. 12, no. 5, pp. 621-632, May 1993.

[581] T. Sasao, *An Exact Minimization of AND-EXOR Expressions Using BDDs*, Proc. of the IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design, Wilhelm Schickard-Institut fuer Informatik, 1993.

[582] T. Sasao, *An Upper Bound on the Number of Products in Minimium ESOPs*, Proc. of the Second Workshop on Applications of Reed-Muller Expansion in Circuit Design, Chiba City, Japan, 27-29 August 1995, pp. 94-101.

[583] T. Sasao, and M. Fujita (Eds.), "Representation of Discrete Functions," Kluwer Academic Publishers, 1996, chapter 13.

[584] J.M. Saul, "An Improved Algorithm for the Minimization of Mixed Polarity Reed-Muller Representations," *Proc. ICCD'90,* pp. 372-375, Sept. 1990.

[585] J. M. Saul, "An Efficient Data Structure for the Minimization of Exor Sums," *Proc. of the Second Workshop on Applications of Reed-Muller Expansion in Circuit Design,* Chiba City, Japan, 27-29 August 1995, pp. 116-122.

[586] I. Schaefer, "An Effective Cube Comparison Method for Discrete Spectral Transformations of Logic Functions," *M. Sc., Thesis,* May 1990.

[587] I. Schaefer, B.J. Falkowski, and M.A. Perkowski, "An Efficient Computer Algorithm for the Calculation of Walsh Transform for Completely and Incompletely Specified Multiple-Valued Input Binary Functions," *Proc. of the 34th IEEE Midwest Symposium on Circuits and Systems,* Monterey, CA, May 1991. PAGES?

[588] I. Schaefer, B.J. Falkowski, and M.A. Perkowski, "A Fast Computer Implementation of Adding and Arithmetic Multi-Polarity Transforms for Logic Design," *Proc. of the 34th IEEE Midwest Symposium on Circuits and Systems,* Monterey, CA, May 1991. PAGES?

[589] I. Schaefer, B.J. Falkowski, and M.A. Perkowski, "A Fast Computer Implementation of Adding and Arithmetic Multi-Polarity Transforms for Logic Design," *International Journal of Electronics.* Pages?

[590]

[591] I. Schắfer and M. A. Perkowski, "Multiple-Valued-Input Generalized Reed-Muller Forms," *Proc IEEE 21st Int. Symp. on Multiple-Valued Logic,* pp. 40-48, May 1991, Victoria, British Columbia.

[592] I. Schaefer, and M.A. Perkowski, "Multiple-Valued Input Generalized Reed-Muller Forms," *IEE Journal,* ???.

[593] I. Schaefer, and M.A. Perkowski, "An Algorithm to Find the Minimal Multiple-Valued Input Kronecker Reed-Muller Form," ???

[594] I. Schaefer, and M.A. Perkowski, "Synthesis of Multi-Level Multiplexer Circuits for Incompletely Specified Multi-Output Boolean Functions with Mapping Multiplexer Based FPGAs," *IEEE Transactions on Computer Aided Design,* ????

[595] I. Schaefer, M.A. Perkowski, and H. Wu, "Multilevel Logic Synthesis for Cellular FPGAs Based on Orthogonal Expansions," *Proc. of IFIP W.G. 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design,"* Hamburg, Germany, Sept. 16-17, pp. 42 - 51, 1993.

[596] J.T. Schwartz, "Ultracomputers," *ACM Transactions on Programming Languages and Systems*, 2, 4, pp. 484-521, 1980.

[597] J.T. Schwartz, "A Taxonomic Table of Parallel Computers, Based on 55 Designs," *Courant Institute*, New York University, 1983.

[598] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A.Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A.L. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," *Technical Report*, Memorandum No. UCB/ERL M92/41, University of California Berkeley, 1992.

[599] S. Shankar, "A Hierarchical Associative Memory Architecture for Logic Programming Unification," *Logic Programming,* Proc. of the 5th Intern. Conf. and Symp. Vol. 2., pp. 1428-1447, MIT Press, 1988.

[600] N. Sherwani, "Algorithms for VLSI Physical Design Automation,"

[601]

[602] K. Skahill, "VHDL for Programmable Logic," Addison-Wesley Publishing Inc., Reading, Massachusetts, 1996.

[603] Y. Shotobatake, and H. Aiso, "A Unification Processor Based on Uniformly Structured Cellular Hardware," *13th Intern. Symp. on Computer Architecture,* pp. 140-148, 1986.

[604] C.E. Shannon, and E.F. Moore, "Machine Aid for Switching Circuit Design," *Proc. IRE,* Vol. 41., pp. 1348-1351, Oct. 1953.

[605] D.E. Shaw, "The NON-VON Supercomputer," *Department of Computer Science*, Columbia University, 1982.

[606] D. Shumake, "The MOS Boolean Analyzer," *M.Sc. Thesis,* UCLA, 1971.

[607] H.J. Siegel, L.J. Siegel, F.C. Kemmerer, P.T. Mueller, Jr., H.E. Smalley Jr., and S.D. Smith, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Trans. on Computers,* C-30, 12, pp. 934-946, 1981.

[608] J. Siekmann, "Unification Theory," *J. of Symbolic Computation,* Vol. 7, pp. 207-274, 1989.

[609] Signetics, *PLD Data Manual,* Signetics' Approach to Logic Flexibility for the '80's", 1986.

[610] J.R. Slagle, et al, "A New Algorithm for Generating Prime Implicants," *IEEE Trans. on Computers,* Vol. C-19, pp. 304-310, 1970.

[611]

[612] N. Song, "Minimization of Exclusive Sum of Products Expressions for Multiple-valued Input Incompletely Specified Functions," *Master thesis,* Dept. of Electrical Engineering, Portland State University, 1992.

[613]

[614] N. Song and M.A. Perkowski, "EXORCISM-MV-2: Minimization of Exclusive Sum of Products Expression for Multiple-valued Input Incompletely Specified Boolean Functions," *Proc. of the 23th International Symposium on Multiple-Valued Logic,* pp. 132-137, May 24-27, 1993.

[615]

[616] N. Song, M. A. Perkowski, M. Chrzanowska-Jeske, A. Sarabi, "A New Design Methodology for Two-Dimensional Logic Arrays," *VLSI Design,* special issue on Decomposition in VLSI Design, (L. Jozwiak ed)., Vol. 3, No. 3-4, 1995, pp. 315-332.

[617] N. Song and M.A. Perkowski, "New Fast Approach to Approximate ESOP Minimization for Imcompletely Specified Multi-Output Boolean Functions," *Proc. RM'97 Conference,* Oxford Univ., U.K., September 1997, pp 61-72.

[618] N. Song, M. Perkowski, "Minimization of Exclusive Sum of Products Expressions for Multi-Output Multiple-Valued Input, Incompletely Specified Functions," *IEEE Transactions on Computer Aided Design,* Vol. 15, No. 4, April 1996, pp. 385-395.

[619] R. Spickelmier, (ed.), "OCT Tools Distribution 2.1," *Electronics Research Laboratory,* University of California, Berkeley, March 25, 1988.

[620] A. Srinivasan, T. Kam, S. Malik, and R.K. Brayton, "Algorithms for Discrete Function Manipulation," *IEEE Proc. of IEEE Int. Conf. on CAD*, ICCAD'90, pp. 92-95, 1990.

[621] STEINBACH, his machine, his cube calculus, his applications ; decomposition etc.

[622]

[623] A. Stern, "Matrix Logic," *North-Holland,* 1988.

[624] Stolfo, S.J., and D.E. Shaw, "DADO: A Tree-Structured Machine Architecture for Production Systems," *Department of Computer Science*, Columbia University, 1982.

[625] R.R. Stoll, "Set Theory and Logic," *Dover Publications, Inc.*, New York, 1979.

[626] C.D. Stormaon, M.R. Brule, J.V. Oldfield, and J.C. Ribeiro, "An Architecture Based on Content-Addressable Memory for the Rapid Execution of Prolog," *Logic Programming: Proc. of the 5th Intern. Conference and Symposium,* ed. Kenneth A. Bowen, Vol. 2., pp. 1449-1473, MIT Press, 1988.

[627] M.D. Strugala, D. Tavangarian, K. Waldschmidt, and G. Roll, "An Associative Processor as a Design Rule Check Accelerator," *Proc. of VLSI and Computers First Intern. Conf. on Computer Technology, Systems and Applications,* pp. 426-431, IEEE, Hamburg, May 11-15, 1987.

[628]

[629]

[630]

[631] Su, S.Y.H., and P.T. Cheung, "Computer Minimization of Multivalued Switching Functions," *IEEE Trans. on Computers,* Vol. 21, No. 9, pp. 995-1003, Sept. 1972.

[632]

[633] P. Suppes, *"Introduction to Logic,"* New York: Van Nostrand Reinhold, 1957.

[634]

[635] A. Svoboda, "Boolean Analyzer," *Proc. Information Processing 68,* Amsterdam, North-Holland, 1969, pp. 824-830.

[636]

[637] A. Svoboda, "Parallel Processing in Boolean Algebra," *IEEE TC,* Vol. C-22, No. 9, pp. 848-851, Sept 1973.

[638] A. Svoboda, and D.E. White, "Advanced Logical Circuit Design Techniques," *Garland STPM Press,* New York, 1979.

[639] M.A. Tapia, J.H. Tucker and A.W. Bennett, "Boolean Integration," *Proc. IEEE Southeast-Con,* Clemson, SC, April 1976.

[640] M.A. Tapia, "Boolean Differentiation and Integration using Karnaugh Map," *Proc. IEEE Southeast-Con.*, 1977.

[641] M.A. Tapia, "Application of Boolean Calculus to Digital System Design," *Proc. IEEE Southeast-Conf.,* Nashville, Tenn., 14-16 April 1980.

[642] M.A. Tapia, and J.H. Tucker, "Complete Solution of Boolean Equations," *IEEE Trans. on Comp.* Vol. C-29, No. 7., pp. 662-665, July 1980.

[643] M.A. Tapia, "Boolean Integral Calculus for Digital Systems," *IEEE Trans. on Comp.*, Vol. C-34, No. 1., pp. 78-81, Jan. 1985.

[644] D.K. Taylor, "Analyzing Relational Databases using Propositional Logic," *M.S. Thesis*, Department of Electrical Engineering, University of Kentucky, December, 1981.

[645] A. Thayse, "Boolean Differential Calculus," *Philips Res. Reports,* Vol. 26., pp. 229-246, 1971.

[646] A. Thayse, and M. Davio, "Boolean Differential Calculus and its Applications in Switching Theory," *IEEE Trans. on Computers,* Vol. C-22., pp. 409-420, 1973.

[647] B.L. Timofeev, "A Machine for Minimizing Relay Circuits in the Disjunctive Normal Form Category," *Avt. i Vyc. Tekhn.,* Vol. 2., 1959.

[648] B.L. Timofeev, "Mechanization of the Process of Determining Minimal Variants of Relay Systems in Disjunctive Normal Forms," *Voprosy Sbornik Avtomat. Upravlenie,* pp. 345-353, 1960.

[649] B.L. Timofeev, "The Minimization of Logical Functions as Disjunctive Normal Forms, with the Aid of a Specialized Computer," *Dokl. Rel. Ustr. Kon. Avt.,* 1962.

[650] P. Tison, "Generalization of Consensus Theory and Application to the Minimization of Boolean Functions," *IEEE Trans. Electronic Computers,* Vol. EC-16, pp. 446-456, 1967.

[651] D. Tavangarian, "Flag-Algebra: A New Concept for the Realisation of Fully Parallel Associative Architectures," *IEE Proc.,* Vol. 136, Pt. E, No. 5, pp. 357-365, September 1989.

[652]

[653] T. Toffoli, and M. Margolus, "Cellular Automata Machines," *MIT Press*, Cambridge, Mass., 1987.

[654] H. Touati, "Perle1DC: a C++ Library for the Simulation and Generation of DECPeRLe-1 Designs," *PRL Technical Note 4,* Digital Equipment Corp., Paris Research Lab, February 1994.

[655] P.C. Traleaven, and G.F. Moll, "A Multi-Processor Reduction Machine for User-Defined Reduction Languages," *Seventh Annual Symposium on Computer Architecture,* La Baule, France, pp. 121-130, 1980.

[656] Ph.C. Treleaven, D.R. Brownbridge, and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *Computing Surveys,* Vol. 14, No. 1, pp. 93-142, March 1982.

[657] Ph.C. Treleaven, "The New Generation of Computer Architecture," *SIGARCH Newsletter,* Vol. 11, No. 3, pp. 402-416, 1983.

[658] C. Tsai and M. Marek-Sadowska, "Multilevel Logic Synthesis for Arithmetic Functions," *Proc. DAC'96*, pp. 242-247.

[659] T.T. Tsukanov, "A Matrix Analyzer of Relay-Contact Networks," *Vses. Mos. Sov. Teorii Ust. Rel. Deist.,* p. 72, 1957.

[660] T.T. Tsukanov, "Problems in Mechanization of the Analysis of Relay-Contact Circuits," *Trudy Tomsk. Elekt.Inst. Inzhen. Zheleznykh Dorog,* Vol. 23, pp. 129-149, 1957.

[661] T.T. Tsukanov, "Some Instances of Application of the Matrix Analyzer," *Trudy Tomsk. Elekt.Inst. Inzhen. Zheleznykh Dorog,* Vol. 23, pp. 150-116, 1957.

[662] S. Turing, *"Alan M. Turing,"* Cambridge, U.K.: W. Heffer & Sons, 1959.

[663] T. Uehara, and N. Kawato, "Logic Circuit Synthesis using Prolog," *New Generation Computing,* Vol. 1., No. 2., 1983.

[664] J.D. Ullman, "Computational Aspects of VLSI," *Computer Science Press*, 1984.

[665]

[666]

[667] M.E. Ulug, and B.A. Bowen, "A Unified Theory of the Algebraic Topological Methods for the Synthesis of Switching Systems," *IEEE TC,* pp. 255-267, March 1974, Vol. C-23, No. 3.

[668]

[669] M.E. Ulug, "VLSI Knowledge Representation Using Predicate Logic and Cubical Algebra," *Proc. IEEE Intern. Conference on Computers and Communications*, Arizona, 1985, pp. 292-297.

[670] M.E. Ulug, "Application of Cubical Array Operators to a Relational Database," *Proc. of the Minnowbrook Workshop,* July 23-26, 1985, Blue Mountain Lake, New York.

[671] M.E. Ulug, "A Real-Time AI System for Military Communications," *Proc. of the Third IEEE Conference on Artificial Intelligence Applications,* February 1987, Orlando, Florida.

[672] URL http://www.reconfig.com/

[673] URL http://www.fccm.org/

[674] URL http://www.ccic.gov/pubs/blue97/nsa/splash.html

[675] URL http://pam.devinci.fr/

[676] URL http://www.research.digital.com/PRL/publications/pam.html

[677] Usher, *"Mechanical Inventions,"*

[678] G.Y. Vichniac, "Boolean derivatives on cellular automata," *Physica D*, 45, pp. 63-74, 1990.

[679] J. Villasenor and W. H. Mangione-Smith, *Configurable Computing*, Scientific American, June 1997.

[680] H.A. Vink, B. Van Dolder, and J. Al, "Reduction of CC-tables Using Multiple Implication," *IEEE Trans. Comp.,* Vol. C-27, No. 10, Oct. 1978.

[681] H.A. Vink, "Minimal TANT Networks of Functions with Don't Care's and Some Complemented Input Variables," *IEEE Trans. Comp.,* Vol. C-27, No. 11., Nov. 1978.

[682]

[683] Virtual ASIC, "Concept Silicon partitions your design onto multiple FPGAs," *INCA company materials,* 1992.

[684]

[685]

[686]

[687] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and Ph. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Trans. on VLSI Systems,* Vol. 4, No. 1., pp. 56 - 69, March 1996.

[688]

[689]

[690] B.W. Wah, Y.W.E. Ma, "MANIP - A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems," *IEEE TC,* Vol. C-33, No. 5, pp. 377 - 390, May 1984.

[691] W.Wan, M.A. Perkowski, "A New Approach to the Decomposition of Incompletely Specified Multi-Output Functions Based on Graph-Coloring and Local Transformations and its Application to FPGA mapping," *Proc. of EURO-DAC'92,* Hamburg, pp. 230-235, September, 1992.

[692] H. Wang, "Towards Mechanical Mathematics," *IBM J. for Research and Development,* 4, 1960, pp. 2-22.

[693] J. Weissman, "Boolean Algebra, Map Coloring, and Interconnections," Amer. Math. Monthly, Vol. 69., pp. 606-613, 1962.

[694] I. Williams, "Using FPGAs to Prototype New Computer Architectures," in *FPGAs*, pp. 373-382.

[695] T.A. Welch, "An Investigation of Descriptor Oriented Architecture," *Computer Architecture News,* Vol. 4, No. 4, pp. 141-146, January 1976.

[696] J. Werner, "Japanese trends in system design. Part 2. Advanced Computer-Aided Design Tools," *VLSI Design,* July 1984, pp.36-42.

[697] N.S. Woo, "A Hardware Unification Unit: Design and Analysis," *12th International Symposium on Computer Architecture,* pp. 198-205, 1985.

[698] W.S. Wojciechowski, and A.S. Wojcik, "Multiple-Valued Logic Design by Theorem Proving," *Proc. 9-th Intl. Symp. on Multiple-Valued Logic,* Bath, England, 1979, pp. 196-199.

[699] W.S. Wojciechowski, "Multiple-Valued Combinational Logic Design using Theorem Proving," *Dissertation, Ill. Inst. of Tech.,* University Microfilms, No. KRA80-2162, May 1980.

[700] W.S. Wojciechowski, and A.S. Wojcik, "Automated Design of Multiple-Valued Logic Circuits by Automated Theorem Proving Techniques," *IEEE Trans. on Computers,* Vol. C-32, pp. 785-798, Sept. 1983.

[701] Wolfe, Mays and D. G. Prinz, "A Relay Machine for the Demonstration of Symbolic Logic", Nature, 165, 1950, p.197.

[702] Wolfe, Mays and Desmond Henry, "Exhibition of the Work of W. Stanley Jevons", Nature, 170, 1952, pp.696-697.

[703] Wolfe, Mays and Desmond Henry, "Jevons and Logic", Mind, LXII, 1953, pp.484-505.

[704] S. Wolfram, "Cellular Automata as Models of Complexity," *Nature*, 311,4, pp. 419-424, 1984.

[705] S. Wolfram, *Theory and Applications of Cellular Automata,* World Scientific Press, Singapore, 1986.

[706] L. Wos, R. Overbeek, E. Lusk, and J. Boyle, "Automated Reasoning: Introduction and Applications," *Prentice-Hall,* Englewood Cliffs, N.J., 1984.

[707] L-.F. Wu, and M.A. Perkowski, "Minimization of Permuted Reed-Muller Trees for Cellular Logic Programmable Gate Arrays," *Proc. of the 2nd Intern. Workshop on Field-Programmable Logic and Applications*, FPL'92, Vienna, Austria, August 31-September 2, 1992, pp. 7/4.1-7/4.4.

[708] Wu, S.B., and M. T. Liu, "A Cluster Structure as an Interconnection Network for Large Multicomputer Systems," *IEEE Trans. on Computers*, C-30, 4., pp. 254-264, 1981.

[709] X. Wu, X. Chen and S. L. Hurst, "Mapping of Reed-Muller Coefficients and the Minimization of Exclusive-OR Switching Functions," *IEE Proc. Pt. E,* Vol. 129, No. 1, pp. 5-20, January 1982.

[710] XILINX, Inc., *"The Programmable Gate Array Data Book,"* 1991.

[711] *XILINX: User Guide and Tutorials*, Xilinx Inc., San Jose, CA, 1991.

[712] *XILINX: XAPP Applications Handbook*, Xilinx Inc., San Jose, CA, 1992.

[713] *XILINX: The Programmable Logic Data Book*, Xilinx Inc., San Jose, CA, 1994.

[714] *XILINX: The Programmable Logic Data Book*, Xilinx Inc., San Jose, CA, 1994.

[715] *XILINX: User Guide and Tutorials*, Xilinx Inc., San Jose, CA, 1991.

[716] *XILINX: XAPP Applications Handbook*, Xilinx Inc., San Jose, CA, 1992.

[717] *Quick Start Guide for Xilinx Alliance Series 1.4*, Xilinx Inc., San Jose, CA, 1997.

[718] *Development System User Guide for Xilinx Alliance Series 1.4*, Xilinx Inc., San Jose, CA, 1997.

[719] *Xilinx Library Guide*, Xilinx Inc., San Jose, CA, 1997.

[720] Dave Van Den Bout, *The Practical Xilinx Designer Lab Book*, Prentice Hall Inc., New Jersey, Dec. 1997.

[721] K. Yamada, and K. Yoshida, "An Application of Boolean Algebra in Practical Situations," *Hitotsubashi J. Arts & Sciences,* Vol. 5., pp. 41-57, 1965.

[722] K. Yamada, H. Nakada, A. Tsutsui, T. Fujii, and N. Ohta, "FPGA Design for Digital Telecommunication Circuits using a High Level Design System," *Report.*

[723] H. Yasura, T. Tsujimoto, K. Tamaru, "Parallel Exhaustive Search For Several NP-Complete Problems Using Content Addressable Memories," *Proc. IEEE Intern. Conference on Circuits and Systems,* ISCAS'88, pp. 333- 336.

[724] T. Yuba, T. Shimada, Y. Yamaguchi, K. Hiraki, and S. Sakai, "Dataflow Computer Development in Japan," *ACM SIGARCH News,* Vol. 18, No. 3, pp. 140-147, Sept. 1990.

[725] A.D. Zakrevskii, "A Machine for the Solution of Logical Problems of Switching Synthesis Type," *Dokl. Rel. Ustr. Kon. Avt.,* 1962.

[726] A.D. Zakrevskii, "A Universal System for the Solution of Problems Similar to Contact Network Synthesis," *Vyc. Tekh. Avt. Teorii. Inf.,* No. 42., pp. 9-37, 1963.

[727] A. Zakrevskij, "Minimum polynomial implementation of systems of incompletely specified Boolean functions," *Proc. Reed-Muller'95,* pp.250-256.

[728] A. Zakrevskij, "Looking for Shortest Solutions of Systems of Linear Logical Equations: Theory and Applications in Logic Design," *Proc. of 2-nd Workshop Boolesche Probleme,* (B. Steinbach, Ed.), 19-20 September 1996, Freiberg/Sachsen, Germany, pp. 63-69.

[729] ZAKREVSKIJ - ADD NEW RESEARCH!!!!

[730] H. Zemanek, "Central European Prehistory of Computing," in Metropolis et al. *"History of Computing,"* (PUBLISHER??).

[731] X. Zeng, M. Perkowski, K. Dill, A. Sarabi, "Approximate Minimization of Generalized Reed-Muller Forms," *Proc. Reed-Muller'95,* pp. 221-230.

[732] I.I. Zhegalkin, "On the Calculation of Propositions in Symbolic Logic," (in Russian), *Math. Sbornik,* Vol. 34., pp. 9-28, 1927.

[733] L. Zhou, *"Testability Design and Testability Analysis of Cube Calculus Machine"*, Master thesis, Dept. of Electrical Engineering, Portland State University, 1995.

[734] B. Zupan, and M. Bohanec, "Experimental Evaluation of Three Partition Selection Criteria for Decision Table Decomposition," *Research Report,* October 19, 1996, Department of Intelligent Systems, Jozef Stefan Institute, 1000 Ljubljana, Slovenia.

[735] B. Zupan, and M. Bohanec, "Learning Concept Hierarchies from Examples by Function Decomposition," *Technical Report, Department of Intelligent Systems*, Jozef Stefan Institute, Ljubljana, Slovenia, September 1996.

[736] Zwicky, "INVENTION,"

8888888888888888888888888888888

[737] M.A. Perkowski, and K. Stanton, "Robotics for Handicapped," *Proc. of Northcon'91,* Portland, 1-3 October 1991, Session S1, pp. 278-284.

[738] C. Espinosa*, M. A. Perkowski, "Hierarchical Hough Transform Based on Pyramidal Architecture," *Proc. of the IPCCC'92, IEEE International Phoenix Conference on Computers and Communications,* pp. 0743 - 0750, Scottsdale, Arizona, April 1-4, 1992.

[739] M.A. Perkowski, A.Sarabi, I. Schaefer, "Application of Orthogonal Transforms in Image Processing," *Proc. of Northcon'91,* Portland, 1-3 October 1991, Session S1, pp. 303-309.

[740] M.A. Perkowski, "A Systolic Architecture for the Logic Design Machine," *Proc. of the IEEE/ACM Intern. Conf. on Computer Aided Design, ICCAD'85,* Santa Clara, 19-21 November 1985, pp. 133-135.

[741] P.M. Ho, and M.A. Perkowski, "Systolic Architecture for Solving NP-Hard Combinatorial Problems of Logic Design and Related Areas," *Proc. of ISCAS'89,* May 9-11, 1989, pp. 1170-1173.

[742] L. Kida, M.A. Perkowski, "The Cube Calculus Machine: A Ring of Asynchronous Automata to Process Multiple-Valued Switching Functions," *Proc. ISCAS'92,* pp. 807-810, 1992.

[743] L.F. Wu, M. A. Perkowski, "Minimization of Permuted Reed-Muller Trees for Cellular Logic Programmable Gate Arrays," In R. Hartenstein, (ed.), "Lecture Notes in Computer Science", *Springer Verlag,* 1993.

[744] A. Coppola, M.A. Perkowski, R. Anderson, J.S. Freedman, E. Pierzchala, "Tokenized State Machine Model for Synthesis of Sequential Circuits into EPLDs and FPGAs," In G. Saucier, (ed.), North-Holland, 1993.

[745] B. Falkowski, I. Schaefer, and M. A. Perkowski, "An Efficient Computer Algorithm for the Calculation of the Walsh Transform for Incompletely Specified Multiple-Valued Binary Functions," *International Journal of Electronics,* Vol. 75, No. 2., pp. 163-175, 1993.

[746] H. Wu, M. A. Perkowski, N. Zhuang, "Canonical Restricted Partially-Mixed-Polarity Reed-Muller Expansion and its Fast Computation," *IEEE Transactions on Computers*, August 1993.

[747] H. Wu, N. Zhuang, and M. A. Perkowski, "Synthesis for Reed-Muller Directed-Acyclic-Graph network," *IEE Proceedings, Pt. E.*, June 1993.

[748] I. Schaefer, and M. A. Perkowski, "Synthesis of Multi-Level Multiplexer Circuits for Incompletely Specified Multi-Output Boolean Functions with Mapping Multiplexer Based FPGAs," *IEEE Transactions on Computer Aided Design,* 23 pages, October 1992.

[749] I. Schaefer, M. A. Perkowski, "Multiple Valued Input Generalized Reed-Muller Forms," *IEE Proceedings, Pt.E,* Vol. 139, No. 6., pp. 519 - 527, November 1992.

[750] L. Csanky, M. Perkowski, I. Schaefer, "Canonical Restricted Mixed-Polarity Exclusive-Or Sums of Products and the Efficient Algorithm for their Minimization," *IEE Proceedings, Pt.E,* Vol. 140, No. 1, pp. 69 - 77, January 1993.

[751] I. Schaefer, B. J. Falkowski, and M. A. Perkowski, "Generation of Adding and Arithmetic Multi-Polarity Transforms for Incompletely Specified Boolean Functions," *International Journal of Electronics,* Vol. 73, No. 2., pp. 321 - 331, 1992.

[752] I. Schaefer, M. A. Perkowski, and H. Wu, "Multilevel Logic Synthesis for Cellular FPGAs Based on Orthogonal Expansions," *Proc. of IFIP W.G. 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design,* Hamburg, Germany, September 16-17, pp. 42 - 51, 1993.

[753] M. A. Perkowski, "A Fundamental Theorem for EXOR Circuits," *Proc. of IFIP W.G. 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design,* Hamburg, Germany, September 16-17, pp. 52 - 60, 1993.

[754] M. A. Perkowski, A. Sarabi, and F. R. Beyl, "Universal XOR Canonical Forms of Switching Functions," *Proc. of IFIP W.G. 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design,* Hamburg, Germany, September 16-17, pp. 27 - 32, 1993.

[755] A. Sarabi, P.F. Ho, K. Iravani, W. R. Daasch, and M. A. Perkowski, "Minimal Multi-Level Realization of Switching Functions Based on Kronecker Functional Decision Diagrams," *Proc. of IEEE International Workshop on Logic Synthesis, IWLS '93,* Tahoe City, CA, pp. P3a-1 - P3a-6, May 1993.

[756] N. Song, and M. A. Perkowski, "A New Design Methodology for Two-Dimensional Logic Arrays," *Proc. of IEEE International Workshop on Logic Synthesis, IWLS '93,* Tahoe City, CA, pp. 1 - 17, May 1993.

[757] N. Song, and M. A. Perkowski, "EXORCISM-MV-2: Minimization of Exclusive Sum of Products Expressions for Multiple-Valued Input Incompletely Specified Functions," *Proc. of the 23nd IEEE International Symposium on Multiple Valued Logic, ISMVL '93,* pp. 132 - 137, Sacramento, CA, May 24-27, 1993.

[758] W. Wan, M. A. Perkowski, "A New Approach to the Decomposition of Incompletely Specified Functions based on Graph-Coloring and Local Transformations and Its Application to FPGA Mapping," *Proc. of the IEEE EURO-DAC '92, European Design Automation Conference,* pp. 230 - 235, Sept. 7-10, Hamburg, 1992.

[759] A. Sarabi, and M. A. Perkowski, "Fast Exact and Quasi-Minimal Minimization of Highly Testable Fixed-Polarity AND/XOR Canonical Networks," *Proc. of the IEEE/ACM Design Automation Conference,* pp. 30 - 35, Anaheim, CA, June 8-12, 1992.

[760] M. A. Perkowski, "The Generalized Orthonormal Expansion of Functions with Multiple-Valued Inputs and Some of its Applications," *Proc. of the 22nd IEEE International Symposium on Multiple Valued Logic, ISMVL'92,* pp. 442 - 450, Sendai, Japan, May 27-29, 1992.

[761] M. A. Perkowski, "A Universal Logic Machine," invited address, *Proc. of the 22nd IEEE International Symposium on Multiple Valued Logic, ISMVL'92,* pp. 262 - 271, Sendai, Japan, May 27-29, 1992.

[762] R. Anderson, A. Coppola, J. Freedman, M. A. Perkowski, "VHDL Synthesis of Concurrent State Machines to a Programmable Logic Device," *Proc. of the IEEE VHDL International User's Forum,* May 3 - 6, Scottsdale, Arizona, 1992.

[763] Concurrent Logic Inc., "CLI 6000 Series Field Programmable Gate Arrays," *Preliminary Information*, Dec. 1, 1991, Rev. 1.3.

[764] M. Davio, J.P. Deschamps, A. Thayse, "Discrete and Switching Functions," *McGraw Hill*, 1978.

[765] D. Debnath, T. Sasao, "GRMIN: A Heuristic Simplification Algorithm for Generalised Reed-Muller Expressions," *Proc. Reed-Muller'95*, pp. 257-264.

[766] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M.A. Perkowski, "Efficient representation and manipulation of switching functions based on Ordered Kronecker Functional Decision Diagrams," *Proc. Design Automation Conference*, pp. 415-419, 1994.

[767] R. Drechsler, "Pseudo Kronecker Expressions for Symmetric Functions," *submitted, WHERE??*. 1996.

[768] R. Drechsler, B. Becker, A. Jahnke, "On Variable Ordering and Decomposition Type Choice in OKFDDs," *Proc. Reed-Muller'95*, pp. 131-136.

[769] R. Drechsler, B. Becker, "Dynamic Minimisation of OKFDDs," *Proc. Reed-Muller'95*, pp. 137-142.

[770] R. Drechsler, B. Becker, S. Ruppertz, "K*BMDs: A New Data Structure for Verification," *Proc. of the European Design and Test Conf.,* pp. 2-8,1996.

[771] B. J. Falkowski, S. Rahardja, "Family of fast transforms for GF(2) orthogonal logic," *Proc. Reed-Muller'95*, pp. 273-280.

[772] J. Froessl, B. Eschermann, "Module Generation for AND/XOR-Fields (XPLAs)," *Proc. IEEE ICCD '91,* pp. 26-29, 1991.

[773] D.H. Green, "Families of Reed-Muller Canonical Forms," *Intern. J. of Electr.,* pp. 259-280, February 1991, No 2.

[774] M. Helliwell, and M. Perkowski, "A Fast Algorithm to Minimize Multi-Output Mixed Polarity Generalised Reed-Muller Forms," *Proc. of the IEEE/ACM DAC*, 1988, paper 28.2, pp. 427- 432.

[775] P. Ho, M. A. Perkowski, "Free Kronecker Decision Diagrams and their Application to ATMEL 6000 FPGA Mapping," *Proc. Euro-DAC '94 with Euro-VHDL ' 94*, pp. 8 - 13, September 19-23, 1994, Grenoble France.

[776] Ch. Tsai and M. Marek-Sadowska, "Logic Synthesis for Testability," private information.

[777] M. Perkowski, P. Dysko, B. Falkowski, "Two learning methods for a tree search combinatorial optimizer," *Proc. of the IEEE Intern. Phoenix Conf. on Computers and Comm.*, Scottsdale, Arizona, March 1990, pp. 606-613.

[778] M. Perkowski, P. Johnson, "Canonical multi-valued input Reed-Muller Trees and Forms," *Proc. 3rd NASA Symp. on VLSI Design,* Oct. 1991, Moscow, Idaho, pp. 11.3.1-11.3.13.

[779] M.A. Perkowski, "The Generalised Orthonormal Expansion of Functions with Multiple-Valued Inputs and Some of its Applications," *Proc. ISMVL'92*, pp. 442-450.

[780] M.A. Perkowski, and M. Chrzanowska-Jeske, "An Exact Algorithm to Minimize Mixed-Radix Exclusive Sums of Products for Incompletely Specifed Boolean Functions," *Proc. of the ISCAS'90,* pp. 1652 - 1655, New Orleans, 1 - 3 May 1990.

[781] M. Perkowski, L. Csanky, A. Sarabi, I. Schaefer, "Fast Minimization of Mixed-Polarity AND/XOR Canonical Networks," *Proc. IEEE Intern. Conf. on Computer Design, ICCD'92,* Boston, October 11-13, pp. 32-36.

[782] M. Perkowski, "A Fundamental Theorem for Exor Circuits," *Proc. Reed-Muller'93*, pp. 52-60.

[783] M. Perkowski, A.Sarabi, F. Beyl, "XOR Canonical Forms of Switching Functions," *Proc. of Reed-Muller'93*, pp. 27-32.

[784] M. Perkowski, A.Sarabi, F. Beyl, "Fundamental Theorems and Families of Forms for Binary and Multiple-Valued Linearly Independent Logic," *Proc. Reed-Muller'95*, pp. 288-299.

[785] M. Perkowski, T. Ross, D. Gadd, J.A. Goldman, N. Song, "Application of ESOP Minimisation in Machine Learning and Knowledge Discovery," *Proc. Reed- Muller '95*, pp. 102-109.

[786] *Proceedings of IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, September, 1993, WSI-93-2, Universitaet Tuebingen, W. Rosenstiel, ed. (Proc. Reed-Muller '93.)

[787] *Proceedings of IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design,* Makuhari, Chiba, Japan, August 27-29, 1995. Fujiki Printing Co., LTD, Iizuka, Japan, T. Sasao, ed. (Proc. Reed-Muller '95).

[788] A. Sarabi, M.A. Perkowski, "Fast Exact and Quasi-Minimal Minimisation of Highly Testable Fixed-Polarity AND/XOR Canonical Networks," *Proc. DAC,'92*, pp. 30-35.

[789] A. Sarabi, P.F. Ho, K. Iravani, W.R. Daasch, M. Perkowski, "Minimal Multi-level Realisation of Switching Functions based on Kronecker Functional Decision Diagrams," *Proc. IWLS '93, Intern. Workshop on Logic Synthesis.*

[790] A. Sarabi, M. Perkowski, "Design for Testability Properties of AND/EXOR Networks," *Proc. Reed-Muller'93*, pp. 147-153.

[791] T. Sasao, "Representation of Logic Functions using EXOR Operators," *Proc. Reed-Muller '95*, pp. 11- 20.

[792] T. Sasao, "Optimisation of Multiple-Valued AND-EXOR Expressions using Multiple-Place Decision Diagrams," *Proc. IEEE 22nd ISMVL*, 1992.

[793] T. Sasao, H. Hamachi, S. Wada, M. Matsuura, "Multi-Level Logic Synthesis Based on Pseudo-Kronecker Decision Diagrams and Local Transformation," *Proc. Reed-Muller'95*, pp. 152-160.

[794] T. Sasao (editor), "Logic Synthesis and Optimisation," *Kluwer Academic Publishers,* 1993.

[795] T. Sasao, "An Exact Minimisation of AND-EXOR Expressions Using BDDs," *Proc. Reed-Muller'93*, pp. 91-98.

[796] T. Sasao, M. Perkowski, "Exor Logic Synthesis," *Kluwer Academic Publishers,* in preparation.

[797] T. Sasao (editor), "Representation of Boolean Functions," *Kluwer Academic Publishers,* 1996.

[798] I. Schaefer, M. Perkowski, "Synthesis of Multi-Level Multiplexer Circuits for Incompletely Specified Multi-Output Boolean Functions with Mapping Multiplexer Based FPGAs," *IEEE Trans. on CAD,*, Vol. 12, No. 11, November 1993. pp. 1655-1664.

[799] I. Schaefer, M.A. Perkowski, "Multiple-Valued Input Generalised Reed-Muller Forms," *IEE Proc.,* Pt.E, Vol. 139, No. 6, pp. 519-527, November 1992.

[800] I. Schaefer, M.A. Perkowski, "Multiple-Valued Input Generalised Reed-Muller Forms," *Proc. of the IEEE ISMVL'91*, pp. 40-48.

[801] B. Steinbach, A. Wereszczynski, "Synthesis of Multi-Level Circuits Using EXOR-Gates," *Proc. Reed-Muller'95*, pp. 161-168.

[802] U. Kebschull, E. Schubert, and W. Rosenstiel, "Multilevel Logic Synthesis based on Functional Decision Diagrams," *Proc. IEEE Euro-DAC*, pp. 43-47, 1992.

[803] N. Song, M. Perkowski, "EXORCISM-MV-2: Minimisation of Exclusive Sum of Products Expressions for Multiple-Valued Input Incompletely Specified Functions," *Proc. ISMVL'93*, May 24, 1993, pp. 132-137.

[804] A. Sarabi, N. Song, M. Chrzanowska-Jeske, M. A. Perkowski, "A Comprehensive Approach to Logic Synthesis and Physical Design for Two-Dimensional Logic Arrays," *Proc. DAC '94,* San Diego, June 1994, pp. 321 - 326.

[805] T. Sasao, "Representation of Logic Functions using EXOR Operators," *Proceedings IFIP WG 10.5 Workshop on Applications of the Reed Muller Expansion in Circuit Design, Reed-Muller'95,* pp. 11-20, August 27-29, 1995, Makuhari, Chiba, Japan.

[806] N. Song, M. A. Perkowski, M. Chrzanowska-Jeske, A. Sarabi, "A New Design Methodology for Two-Dimensional Logic Arrays," *VLSI Design,* special issue on Decomposition in VLSI Design, (L. Jozwiak ed)., Vol. 3, No. 3-4, 1995.

[807] A. Zakrevskij, "Minimum polynomial implementation of systems of incompletely specified Boolean functions," *Proc. Reed-Muller'95,* pp.250-256.

[808] X. Zeng, M. Perkowski, K. Dill, A. Sarabi, "Approximate Minimization of Generalized Reed-Muller Forms," *Proc. Reed-Muller'95,* pp. 221-230.

[809] X. Zeng, M. Perkowski, H. Wu, A. Sarabi, "A New Exact Algorithm for Highly Testable Generalized Partially-Mixed-Polarity Reed-Muller Forms," *Proc. Reed-Muller'95*, pp. 231-239.

[810] S.B. Akers, "Binary decision diagrams," *IEEE Transactions on Computers,* Vol. C-27, No. 6, pp. 509-516, June 1978.

[811] R.E. Bryant, "Graph-based algorithms for boolean function manipulation, *IEEE Transactions on Computers*, C-35, No. 8, pp. 667-691, 1986.

[812] R.E. Bryant and Y.A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams", *Proc. of the 32nd ACM/IEEE Design Automation Conference,* June 1995.

[813] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M.A. Perkowski, "Efficient representation and manipulation of switching functions based on Ordered Kronecker Decision Diagrams," *in the Proc. of 31st ACM/IEEE Design Automation Conference,* pp. 415-419, 1994.

[814] R. Drechsler, H. Hengster, H. Schaefer, B. Becker, Testability of AND?EXOR Expressions, " *Proc. of the IEEE European Test Workshop*, Montpellier, June, 1996.

[815] M.A. Heap, W.A. Rogers, and M.R. Mercer, "A Synthesis Algorithm for two-level XOR based circuits, *IEEE 1992 Intern. Conf. On Computer Design: VLSI in Computers and Processors, ICCD'92*, pp. 459-462, 1992.

[816] L. Jozwiak, and H. Mijland, "On the use of OR-BDDs for Test Generation, *Microprocessing and Microprogramming,* 35 (1992), pp. 159-166. (also *Proc 18th Euromicro Conference,* Paris, Sept. 1992).

[817] K. Karplus, "ITEM: An If-Then-Else Minimizer for Logic Synthesis," *Proc. EURO-ASIC*, pp. 2-7, June 1992, Paris, France.

[818] U. Kenschull, E. Schubert, and W. Rosenstiel, "Multilevel logic synthesis based on functional decision diagrams, *In Proc. EDAC 92*, pp. 43-47, 1992.

[819] U. Kebschull, and W. Rosenstiel, "Efficient graph based computation and manipulation of functional decision diagrams, *Proc. EDAC'93*, pp. 43-47, 1993.

[820] M.A.J. Kolsteren, "Heuristic Algorithms for the construction of OR-BDDs," Project report, coach: Dr. ir. L. Jozwiak, Eindhoven University of Technology, Faculty of Electrical Engineering, Digital Information Systems Group, July 1992-January 1993.

[821] T. Kozlowski, E. L. Dagless, and Jonathan M. Saul, "Unified Decision Diagrams: A representation for mixed AND-OR/EXOR combinational networks," *Proc. Reed-Muller'95 Workshop*, Chiba, Japan, August 1995, pp. 177-184.

[822] Y.F. Lai, M. Pedram, and S.B.K. Vrudhula "EVBDD-based algorithms for integer linear programming,spectral transformation, and functional decomposition," *IEEE Trans. Computer Aided-Design of Integrated Circuits and Systems*, Vol. 13, No. 8, pp. 959-975, 1994.

[823] C.Y. Lee, "Representation of switching circuits by binary-decision programs," *Bell Systems Technical Journal,* Vol. 38, pp. 985-999, 1959.

[824] L. McKenzie, L. Xu, and A. Almaini, "Graphical representation of generalized Reed-Muller expansion," *IFIP 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, Hamburg, Sept. 16-17, pp. 181-187, 1993.

[825] S. Minato, "Graph-Based Representations of Discrete Functions," *Proc. Reed-Muller'95 Workshop*, Chiba, Japan, August 1995, pp. 1-10.

[826] M. Perkowski, T. Ross, D. Gadd, J. A. Goldman, and N. Song, "Application of ESOP Minimization in Machine Learning and Knowledge Discovery," *Proc. Reed-Muller'95 Workshop,* Chiba, Japan, August 1995, pp. 102-109.

[827] M.A. Perkowski, M. Chrzanowska-Jeske, A. Sarabi, and I. Schaefer,"Multi-Level logic synthesis based on Kronecker and Boolean Ternary Decision Diagrams for incompletely specified functions, *VLSI Design,* special issue on Decomposition in VLSI Design, (L. Jozwiak ed)., Vol. 3, No. 3-4, 1995.

[828] M. Perkowski, L. Jozwiak, R. Drechsler, "Cellular EXOR Diagrams", New paper in writing. Will be submitted to the same symposium.

[829] Sarabi, P.F. Ho, K.Iraani, W.R. Daasch, and M.A. Perkowski, "Minimal multi-level realization of switching functions based on Kronecker functional decision diagrams, *in Proc. Intern. Workshop on Logic Synthesis*, pp. 3a1-3a6, 1993.

[830] S.M. Reddy, "Easily testable realizations for logic functions," *IEEE Trans. on Computers,* Vol. 21, pp. 1183-1188, 1972.

[831] T. D. Ross, M.J. Noviskey, T.N. Taylor, D.A. Gadd, "Pattern Theory: An Engineering Paradigm for Algorithm Design," *Final Technical Report WL-TR-91-1060,* Wright Laboratories, USAF, WL/AART/WPAFB, OH 45433-6543, August 1991.

[832] A. Sarabi and M.A. Perkowski, "Fast exact and quasiminimal minimization of highly testable fixed-polarity AND/EXOR canonical networks," *Proc. 29th ACM/IEEE DAC*, pp. 30-35, 1992.

[833] A.Sarabi, and M.A. Perkowski, "Design for Testability properties of AND/EXOR networks," *Proc. of IFI Workshop on Applications of RM Expansion in Circuit Design,* pp. 147-153, Hamburg, 1993.

[834] T. Sasao, "Ternary decision diagrams and their applications," *Intern. Workshop on Logic Synthesis*, Lake Tahoe, pp. 6c, 1-11, May 1993.

[835] T. Sasao, (ed), "Logic Synthesis and Optimization," *Kluwer Academic Publishers*, 1993.

[836] T. Sasao, "A design method for AND-OR-EXOR three-level networks", *International Workshop on Logic Synthesis*, Lake Tahoe, May 1995.

[837] T. Sasao, S. Wada, H. Hamachi, and M. Matsuura, "Multi-level logic synthesis based on pseudo-Kronecker decision diagrams and logic transformation," *IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, August 1995.

[838] T. Sasao, "Easily testable realizations for generalized Reed-Muller expressions," *Proc. IEEE The Third Asian Test Symposium,* pp. 157-162, Nov. 1994.

[839] T. Sasao, "FPGA Design by Generalized Functional Decomposition," *in "Logic Synthesis and Optimization,"* T. Sasao. (Ed), Kluwer Academic Publishers, pp. 233-258, 1993.

[840] B. Steinbach, and A. Wereszczynski, "Synthesis of Multi-Level Circuits Using EXOR-Gates," *Proc. Reed-Muller'95 Workshop,* Chiba, Japan, August 1995, pp. 161-168.

[841] ??? Toida and N.S.V. Rao, "On test generation for combinational circuits consisting of AND and EXOR gates,", *Digest of Papers of 1992 IEEE VLSI Test Symposium Design, Test and Application: ASICs and Systems-on-a-Chip*, pp. 113-118, 1992.

[842] W. Wan, and M. Perkowski, "A New Approach to the Decomposition of Incompletely Specified Multi-Output Function Based on Graph Coloring and Local Transformations and Its Application to FPGA Mapping,"

[843] Pranav Ashar, Srinivas Devadas and A. Richard Newton, "Sequential Logic Synthesis", Kluwer Academic Publishers, 1992.

[844] R.L. Ashenhurst, "The Decomposition of Switching Functions," *Proc. of the Symposium on the Theory of Switching,* April 2 - 5, 1957, Ann. Computation Lab, Harvard University, Vol. 29, pp. 74 - 116, 1959.

[845] Kenneth J. Ayala, "The 8051 Microcontroller, Architecture, Programming, and Applications," *West Publishing Company,* 1991.

[846] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis," *Kluwer Academic Publishers,* Boston, Massachusetts, 1984.

[847] R. K. Brayton, A. Sangiovanni-Vincentelli, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, S. Qadeer, R. K. Ranjan, T. R. Shiple, G. Swamy, T. Villa, G. D. Hachtel, F. Somenzi, A. Pardo, and S. Sarwary, "VIS: A System for Verification and Synthesis," in *Computer-Aided Verification,* July 1996.

[848] M. Ciesielski, and J. Shen, "A Unified Approach to Input-Output Encoding for FSM State Assignment," *Proc. Design Automation Conf.*, pp. 176 - 181, 1991.

[849] H.A. Curtis, "A New Approach to the Design of Switching Circuits," Princeton, N.J., *Van Nostrand,* 1962.

[850] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. Perkowski, "Efficient Representation and Manipulation of Switching Functions Based on Ordered Kronecker Functional Decision Diagrams," *Proc. of the Design Automation Conference,* San Diego, CA, June 1994, pp. 415-419.

[851] A. Ghosh, S. Devadas, and A.R. Newton, "Sequential Logic Testing and Verification," *Kluwer Academic Publishers*, 1992.

[852] M. J. C. Gordon and T. F. Melham (editors), "Introduction to HOL: A theorem proving environment for higher order logic," *Cambridge University Press,* 1993.

[853] Gary D. Hachtel and Fabio Somenzi, "Logic Synthesis and Verification Algorithms," *Kluwer Academic Publishers,* 1996.

[854] J. Hartmanis, and R.E. Stearns, "Algebraic Structure Theory of Sequential Machines," Prentice-Hall, Englewood Cliffs, N.J. 1966.

[855] Randy H. Katz, "Contemporary Logic Design," *Benjamin/Cummings Publishing Company,* 1994.

[856] Zvi Kohavi, "Switching and Finite Automata Theory," New York, NY: McGraw Hill, 1970.

[857] E.B. Lee, and M. Perkowski, "Concurrent Minimization and State Assignment of Finite State Machines," *Proc. IEEE Conference on Systems, Man and Cybernetics,* Halifax, Canada, Oct. 1984, pp. 248 - 260.

[858] C. Leiserson, F. Rose, J. Saxe, "Optimizing Synchronous Circuitry by Retiming, *Third Caltech Conference on VLSI*, 1983, pp. 87-116.

[859] G. De Micheli, "Synchronous Logic Synthesis: Algorithms for Cycle-Time Optimization", *IEEE Trans. on CAD*, Vol.10, No.1, Jan. 1991, pp. 63-73.

[860] G. De Micheli, *Synthesis and Optimization of Digital Circuits* McGraw-Hill, Inc., New York, 1994.

[861] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J. S. Zhang, "Decomposition of Multiple-Valued Relations," *Proc. ISMVL'97*, Halifax, Nova Scotia, Canada, May 1997, pp. 13 - 18.

[862] J.P. Roth, R.M. Karp, "Minimization over Boolean Graphs," *IBM Journal Res. and Develop.*, No. 4, pp. 227 - 238, April 1962.

[863] Tsutomu Sasao, (ed) "Logic Synthesis and Optimization", *Kluwer Academic Publishers,* Boston, MA, 1993.

[864] Tsutomu Sasao, and Masahiro Fujita, "Representations of Discrete Functions," *Kluwer Academic Publishers,* Boston, MA, 1996.

[865] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," *Tech. Rep. UCB/ERL M92/41*, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.

[866] Kevin Skahill, "VHDL for Programmable Logic", Addison Wesley, 1996.

[867] Michael Slater, "Microprocessor-Based Design, A Comprehensive Guide to Effective Hardware Design", *Prentice-Hall, Inc.,* 1989.

[868] Zhou, M. C. (Ed.), "Petri Nets in Flexible and Agile Automation," *Kluwer Academic Publishers,* Boston, MA, 1995.

[869] C. Leiserson, F. Rose, J. Saxe, "Optimizing Synchronous Circuitry by Retiming, *Third Caltech Conference on VLSI*, 1983, pp. 87-116.

[870] S. Malik, E. Sentovich, R. Brayton, A. Sangiovanni-Vincentelli, "Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques", *IEEE Trans. on CAD*, Vol.10, No.1, Jan. 1991, pp. 74-84.

[871] S.T. Chakradhar, S. Dey, M. Potkonjak, S.G. Rothweiler, "Sequential Circuit Delay Optimization using Global Path Delays", *Proc. 1993 Design Automation Conf.*, 1993, pp. 483-489.

[872] A.P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabey, R.W. Broderson, "Optimizing Power Using Transformations", *IEEE Trans. on CAD*, Vol.14, No.1, Jan. 1995, pp. 12-

[873] M. Damiani, G. De Micheli, "Recurrence Equations and the Optimization of Synchronous Circuits", *Proc. 1992 Design Automation Conf.*, 1992, pp. 556-561.

[874] M. Damiani, G. De Micheli, "Don't Care Set Specification in Combinational and Synchronous Logic Circuits", *IEEE Trans. on CAD*, Vol.12, No.3, March 1993, pp. 365-388.

[875] G. De Micheli, "Synchronous Logic Synthesis: Algorithms for Cycle-Time Optimization", *IEEE Trans. on CAD*, Vol.10, No.1, Jan. 1991, pp. 63-73.

[876] G. De Micheli, *Synthesis and Optimization of Digital Circuits* McGraw-Hill, Inc., New York, 1994.

[877] S. Dey, M. Potkonjak, S. Rothweiler, "Performance Optimization of Sequential Circuits by Eliminating Retiming Bottlenecks", *Intl. Conference on CAD*, 1992, pp. 504-509.

[878] P. Gutwin, P. McGeer, R. Brayton, "Delay Prediction for Technology-Independent Logic Equations", *Proc. IEEE Intl. Conference on Computer Design*, 1992, pp. 468-471

[879] S. Hassoun, C. Ebeling, "Architectural Retiming: Pipelining Latency-Constrained Circuits", *Proc. 1996 Design Automation Conf.*, 1996.

[880] G.D. Hachtel, M. Hermida, A. Pardo, M. Poncino, F. Somenzi, "Re-encoding Sequential Circuits to Reduce Power Dissipation", *Intl. Conference on CAD*, 1994, pp. 70-73.

[881] S. Iman and M. Pedram "Logic Extraction and Factorization for Low Power", *Proc. 1995 Design Automation Conf.*, 1995, pp. 248-253.

[882] Z. Iqbal, M. Podkonjak, S. Day, and A. Parker, "Critical Path Minimization using Retiming and Algebraic Speedup", *Proc. 1993 Design Automation Conf.*, 1993, pp. 573-577.

[883] K.N. Lalgudi, M.C. Papaefthymiou, "DelaY: An efficient Tool for Retiming with Realistic Delay Modeling", *Proc. 1995 Design Automation Conf.*, 1995, pp. 304-309.

[884] C. Leiserson, F. Rose, J. Saxe, "Optimizing Synchronous Circuitry by Retiming, *Third Caltech Conference on VLSI*, 1983, pp. 87-116.

[885] B. Lin, H. Touati, A. Newton, "Don't Care Minimization of Multi-Level Sequential Logic Networks", *Intl. Conference on CAD*, 1990, pp. 414-417.

[886] B. Lin, "Restructuring of Synchronous Logic Circuits", *Proc. European Design Automation Conference*, 1993, pp. 205-209.

[887] S. Malik, E. Sentovich, R. Brayton, A. Sangiovanni-Vincentelli, "Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques", *IEEE Trans. on CAD*, Vol.10, No.1, Jan. 1991, pp. 74-84.

[888] T. Marchock, A. El-Maleh, W. Maly, J. Rajski, "On Test Set Preservation of Retimed Circuits", *Proc. 1995 Design Automation Conf.*, 1995, pp. 176-182.

[889] J. Monteiro, S. Devadas, A. Gosh, "Retiming Sequential Circuits for Low Power", *Intl. Conference on CAD*, 1993, pp. 398-402.

[890] S. Iman, M. Pedram, "Logic Extraction and Factorization for Low Power", *Proc. 1995 Design Automation Conf.*, 1995, pp. 87-116.

[891] M. Potkonjak, S. Dey, Z. Iqbal, A. Parker. "High Performance Embedded System Optimization Using Algebraic and Generalized Retiming Techniques", *Proc. IEEE Intl. Conference on Computer Design*, 1993, pp.498-504.

[892] N. Shenoy and R. Rudell, "Efficient Implementation of Retiming", *Intl. Conference on CAD*, 1994, pp. 226-233.

[893] V. Singhal, C. Pixley, R.L. Rudell, R.K. Brayton, "The Validity of Retiming Sequential Circuits", *Proc. 1995 Design Automation Conf.*, 1995, pp. 316-321.

[894] K.J. Singh, A.Wang, R.K. Brayton, A. Sangiovanni-Vincentelli, "Timing Optimization of Combinational Logic", *Intl. Conference on CAD*, 1988, pp. 282-285.

[895] E. Sentovich et al., "SIS: A System for Sequential Circuit Synthesis", Electronics Research Laboratory, Memorandum No. UCB/ERL M92/41, Dept. of EECS, University of California, Berkeley, 1992.

[896] N. O'Neill, "Sequential Logic Synthesis Based on Retiming-Based Factorization", M.S. Thesis, ECE Dept., University of Massachusetts, Amherst, 1997.

[897] H.J. Touati and R.K. Brayton, "Delay Optimization through Partial Collapsing and Decomposition", Notes *Intl. Workshop on Logic Synthesis*, 1991.

[898] H.J. Touati and R.K. Brayton, "Computing the Initial States of Retimed Circuits", *IEEE Trans. on CAD*, (12)1, Jan. 1993, pp. 157-162.

[899] C.Y. Tsui, M. Pedram, C-A Chen, and A. Despain, "Technology Decomposition and Mapping Targetting Low Power Dissipation", *Proc. 1994 Conference on CAD*, 1994, pp. 82-87.

[900] D. Wallace, M. Chandrasekhar, "High level delay estimation for technology-independent logic equations", *IEEE Intl. Conference on Computer Aided Design*, 1990.

[901] E. Lehman, Y. Watanabe, J. Grodstein, H. Harkness, "Logic Decomposition during Technology Mapping", *Proc 1996 Design Automation Conf.*, 1996.

[902] N. Wehn, J. Biesenack, T. Langmaier, M. Münch, M. Pilsl, S. Rumler, and P. Duzy, "Scheduling of Behavioral VHDL by Retiming Techniques", *Proc. Euro-DAC'94*, Sept. 1994.

[903] M. Perkowski - How to Do Research.

[904] Janaka Jayawardena et al - Introduction to EE User Environment.

[905] Douglas Hall - Mentor Tools tutorial session # 1.

[906] Douglas Hall - Mentor Tools tutorial session # 2.

[907] Douglas Hall - Mentor Tools tutorial session # 3. How to access Mentor tools.

[908] David Foote - XACT Development System Tutorial.

[909] Djoikc et al. - "Parallel Algorithms for Generating Subsets and Set Partitions"

[910] M. Perkowski - Explanation of Mini-Project II. Fencing State Machines.

[911] Akl et al - "A simple optimal systolic algorithm for generating permutations."

[912] Ivan Stojmenovic - "A Simple Systolic algorithm for Generating combinations in Lexicographical order".

[913] Djokic et al. "Parallel Algorithms for Generating Subsets and Set Partitions".

[914] H. Akkary - "A VHDL Simulation Model for a Cube Calculus Machine".

[915] Coen Engelbarts, "The Multiple-valued Cube Calculus Machine Version 2.5."

[916] Dwaine Franzke, Frank Ma, Jerzy Kolinski - Hough Transform Design.

[917] Frank Ma - Hough Transform, another version.

[918] Jerzy Kolinski,"Class VHDL Project, Address Decoder Design".

[919] Hens Vanderschoot, "Class VHDL Project, Asynchronous State Machines".

[920] Hens Vanderschoot, "Programmable Deskew Circuit, Class VHDL Project."

[921] Hens Vanderschoot, "Class VHDL Project, Parallel Systolic Sorter".

[922] Vance Navarette, "Systolic Sorter", Class VHDL Project.

[923] Tom Roth, "Class VHDL Project, Systolic Sorter".

[924] Tsutomu Sasao and Marek Perkowski - EXOR logic synthesis. Chapters 1,2,3,4.

[925] Drechsler et al, DAC'95 paper on Ordered Kronecker Functional Decision Diagrams.

[926] Perkowski et al, Journal of VLSI paper on " Multi-level Logic Synthesis Based on Kronecker Decision Diagrams and Boolean Ternary Decision Diagrams for Incompletely Specified Functions."

[927] A. Sarabi, M. Perkowski, "Design for Testability Properties of AND/EXOR Networks".

[928] Ning Song, M. Perkowski, "A New Design Methodology for Two-Dimensional Logic Arrays".

[929] Wei Wan, M. Perkowski , EURO-DAC '92 paper on TRADE decomposition.

[930] A. Sarabi et al, DAC'94 paper on two-dimensional logic arrays.

[931] Perkowski et al, "Application of Orthogonal Transforms in Image Processing".

[932] Pierzchala, Perkowski, Grygiel, ISMVL'94 paper on Field Programmable Analog Array.

[933] Pierzchala, Perkowski, Transparencies on Field Programmable Analog Arrays.

[934] Song et al, "A New Design Methodology for Two-Dimensional Logic Arrays".

[935] Haomin Wu et al "Generalized Partially-Mixed-Polarity Reed-Muller Expansion and its Fast Computation."

[936] Wei Wan - Transparencies on TRADE decomposition.

[937] Bernd Steinbach - Data Structures and Algorithms for Solving Very Large Boolean Problems, and Design of Circuits Using Decomposition Methods.

[938] Steinbach's (XBOOLE) AND/OR/EXOR decomposition.

[939] Marek Perkowski - Transparencies on Binary Orthogonal Logic. Universal cells. Generalized PLAs.

[940] Robert C. Minnick - A Survey of Microcellular Research.

[941] Marek Perkowski - Transparencies on Multiple-Valued Orthogonal Logic.

[942] Haomin Wu, Marek Perkowski, Nan Zhuang - Synthesis of MUX DAG Network with Application to FPGAs and BDDs.

[943] Marek Perkowski - Transparencies on Families of Canonical AND/EXOR forms.

[944] M. Perkowski - SREP Proposal to Air Force Office of Scientific Research. Functional Decomposition of Binary, Multiple-valued and Fuzzy logic for Applications in Pattern Theory.

This is a guidelines for us what we are supposed to do. Also, list of decomposition literature with 443 positions.

[945] Transparencies for "Decomposition of Information Systems Using Decision Tables." This paper is easier to understand that other Luba's papers and has also no errors. It has good link to automatic knowledge acquisition in expert systems.

[946] M. Perkowski - presentation about decomposition from the point of view of future research at Wright Labs.

[947] Selvaraj, Luba et al, - "A Generalized Decomposition of Boolean Functions and its Application in FPGA-based Synthesis."

[948] Luba and Lasocki - "Decomposition of Multiple-valued Boolean Functions". Take care - many errors.

[949] Transparencies according to Kandel and Francioni. This paper is good starting point to fuzzy logic, but decomposition is now obsolete.

[950] Wei Wan - Master thesis about TRADE and decomposition. Read this first. Try to change from maps and/or cubes to BDDs. Many good ideas.

[951] Selvaraj - Ph.D thesis on serial/parallel decomposition. The most recent explanation of Luba's approach, but not very well written. Many good examples.

[952] Pedram et al. - DAC '93 paper on Curtis decomposition for multiple-output, incomplete functions based on standard BDDs and ternary BDDs (with dont;t cares). Many errors. Idea of cut in BDD being the same as multiplicity index.

[953] Luba et al - Decomposition of Information Systems Using Decision Tables. Good paper. Reed it as first Luba paper.

[954] Pedram et al. - Logic Synthesis using Multiple-Output Function Decomposition. Better explanation than DAC'93. Beware of errors.

[955] Craig Files, - Using a search heuristics in an NP-complete problem in Ashenhurst-Curtis Decomposition. Idea of step-by-step selection of variables for bound set. Very good, can be further improved with better function representation. Can be also extended to multiple-valued functions.

[956] Francioni and Kandel - the first paper in fuzzy decomposition. Good literature and intro.

[957] Tsutomu Sasao - "FPGA design by Generalized Functional Decomposition" Historically valid. good examples, easy to read.

[958] Designer's views mixed on analog HDLs.

[959] EE 573 FPAA Project Topics.

[960] E. Pierzchala, M. Perkowski, Field Programmable Analog Arrays (FPAAs). Transparencies.

[961] IMP's EPAC IMP50E10. Data sheet of the first commercial FPAA.

[962] E. Pierzchala, M. Perkowski, High Speed Field Programmable Analog Arrays (FPAAs) Architecture Design. Transparencies.

[963] E. Pierzchala et al., "Current-Mode Amplifier/Integrator for a Field-Programmable Analog Array".

[964] Salter et al. Analogix Corporation's Analog Cell.

[965] Lee and Gulak, - "MOS Transconductor Based Field-Programmable Analog Array".

[966] Frank Goodenough - "Analog Counterparts of FPGAs Ease System Design".

[967] M. Perkowski - A Survey of Literature on Function Decomposition, report to WL. Sept. 1994.

[968] Bryant and Chen - "Verification of arithmetic functions with Binary Moment Diagrams.

[969] M. Perkowski - Class Transparencies on Tree and Heuristic Search. This should help you in programming tree search algorithms for graph coloring, set covering, assignment/encoding, variable partitioning and so on.

[970] Information on software packages POLO and Espresso in directory /stash/polo.

[971] Jeff Goldman - Pattern Theoretic Knowledge Discovery. Explains functional decomposition from the machine learning point of view.

[972] Transparencies about ESOP minimizers, especially EXORCISM-MV-2 by Ning Song.

[973] Helliwell and Perkowski - DAC'88 paper about first efficient ESOP minimizer. Now only of historic value, but easy to understand and many examples to practice.

[974] M. Perkowski and Chrzanowska - "An Exact Algorithm to Minimize Mixed-Radix..... - the first exact ESOP minimizer from 1990. Since then superseded by Sasao and Koda.

[975] M. Perkowski, Lecture notes on possible improvements to our ESOP exact minimizer program.

[976] Z. Pawlak, "Rough Sets - Theoretical Aspects of Reasoning about Data," *Kluwer Academic Publishers*, Boston, London, Dordrecht, 1991.

[977] Q. Chen, "The Design of Cube Calculus Machine Using SRAM-Based FPGA Reconfigurable Hardware DEC's PeRLe-1 Board", Master's Thesis, Portland State University, 1998.

[978] M. Muraszkiewicz and H. Rybinski, "Towards a Parallel Rough Sets Computer", In Wojciech P. Ziarko (Ed.), Rough Sets, Fuzzy Sets and Knowledge Discovery (pp. 434-443), New York: Springer-Verlag.

[979] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age", *IEEE Transactions on VLSI System*, Vol. 4, No. 1, pp. 56-69, March 1996.

[980] L. Moll, J. Vuillemin and P. Boucard, "High-Energy Physics on DECPeRLe-1 Programmable Active Memory", *Proceedings of the 1995 ACM International Symposium on FPGAs*, Monterey, California, Feb 1995.

[981] M. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, Vol. C-21, pp. 948-960, 1972.

[982] G. Almasi and A. Gottlieb, "Highly Parallel Computing," *The Benjamin/Cummings Publishing Company,* Inc., Redwood City, California, 1994.

[983] P.J. Lingras and Y.Y. Yao, "Data Mining Using Extensions of the Rough Set Model," *Journal of the American Society for Information Science*, 49(5):415-422, 1998.