Chapter 1

1 Computer-Aided Logic Design

1.1 Introduction

Hardware components of computers are physical models of logical reasoning. Procedures based on ligical disciplines of mathematics are used to design these components[1]. Examples of such procedures will be presented here in the form of APL programs intended to solve basic problems of computer logic design. [2]

The three blocks of programs given here were used by the partticipants of the Short Course, ***Advanced Logical Circuit Design Techniques***, presented by UCLA Extension (March 1977). They are:

1. SYSTEM –
    a. permits the transformation of a problem specification into a set of Boolean functions defined by a truth table;
    b. derives the Existence Function of the system;
    c. and provides a tool
        i. for minimization
        ii. and for logical relation analysis.
2. OPTIMA – permits the optimal design of a two-level multiple-output combinational circuit based on a rigorous mathematical principle.
3. BOOL – solves systems of Boolean equations of the general type; used for computer-aided design of sequential circuits.

To use the programs effectively, it is necessary that one understand:

1. the general philosophy of problem specification and solution;
2. the programming symbolism; and the interpretation of the printout.

1.2 General Philosophy of Problem Specification and Solution

The following unified point of view is recommended for solving problems in logic design:

1. SPECIFICATIONS should be presented in propositional calculus, Boolean algebra, or in the algebra of sets (classes);
2. EXECUTION of the solution procedure should be based on the algebra of sets (chart methods);
3. RESULTS are represented formally in Boolean algebra or by graphics.

The hardware design deals with physical phenomena related by a ***cause-effect*** relationship between ***states*** (events). The ***state*** of a ***system*** can be described as a configuration of validities of propositions concerning measurable quantities (i.e., voltages, currents, etc.).

---

[1] Which led to the creation of RTL languages such a VHDL and Verilog, and the collapse of drawn schematics to create a design.

[2] This should not be restricted to "computer" design – i.e., covers any logic design.

EXAMPLE 1. For voltage measurement of terminals X1, X2, … , Xj, … we use ***Propositional Variables*** (x 1), (x 2), … , (X j), … attached to the propositions describing the outcome of voltage measurement:

(X j) ≡ (Terminal XJ is HIGH) ≡ (Voltage at XJ is above 4.5V)[3]

($\underline{X}$ j) ≡ (Terminal XJ is LOW) ≡ (Voltage at XJ is below 0.5V)

Note:

1. The existence of two thresholds and their separation
2. ($\underline{X}$ j) is a negation of (X j) so that

   [(X j) is TRUE] → [($\underline{X}$ j) is FALSE] and visa versa;
3. The underlining of literals is used to express ***negation*** (complementation)

The ***transition*** of a system from one state to another will be described by ***two subsequent states***. The first will be called the ***cause*** of the state which follows it, which in turn will be called its ***effect***.

Logical time will be defined later and the definition will be derived from the ***cause-effect*** relationship previously mentioned.

A ***subsystem*** is a subset of variables of a system that possesses certain properties. For instance, input variables of a combinational circuit have the property that they are mutually independent, and the output variables of the circuit have the property that each one is a Boolean function of the input variables (exclusively). In this case, we have two subsystems within a system. There may be more than two subsystems to consider when solving some problems of circuit design.

The ***logical relation*** between subsystems belonging to a system will be explained here for *two* subsystems by the use of Marquand Charts[4] of Boolean Functions.

Example 2: Subsystem with X-variables: (X j); j = 1, 2, 3; and the subsystem with Y-variables: (Y k); k = 1, 2; together these form a system. The validities of X-variables can take on eight different configurations; the validities of Y-variables can take on four configurations. When there is no logical relation between the subsystems, each of the 32 validity configurations of all five variables of the system is equally possible. When the system obeys postulated conditions (constraints), there will be a set of configurations (here from the set of 32) that will be ruled out (discarded). The configurations that survive the process of elimination define the ***Existence Function*** of the system as a whole.

To describe the Marquand Chart suitable to explain the concept of logical relation, the configurations of validities of X- and Y-variables are identified (labeled) by integers in the usual way. (See *Figure 1-1 A Marquand Chart for a 5-Variable System*)

The eight possible configurations of validities of X-variables will be identified by the integer IX, where IX ε {0, 1, 2, 3, 4, 5, 6, 7} under the rule that IX, written as a three-bit binary number $(x_3, x_2, x_1)_2$, belongs to the configuration of validities: (X 3) = x_3, (X 2) = x_2, (X 1) = x_1. For instance, IX = 3 = $(011)_2$ stands for (X 3) = 0 (false) and (X 2) = (X 1) = 1 (true).

The Marquand Chart for the system of our example is shown in *Figure 1-1.* The horizontal scale of the chart belongs to the subsystem X: (X j); j = 1, 2, 3; NX = 3. The columns are labeled in IX from left to right, IX = 0, 1, 2, 3, 4, 5, 6, 7; the number of columns (total number of validity configurations) is designated by NNX, NNX = 8. The vertical scale of the charts belongs to the subsystem Y: (Y k); k = 1, 2; NY = 2, NNY = 4 (number of rows). The rows are labeled in IY = 0, 1, 2, 3.

[3] 5V system

[4] 1880 mathematical paper by Marquand introduced the mapping that predates the Karnough map

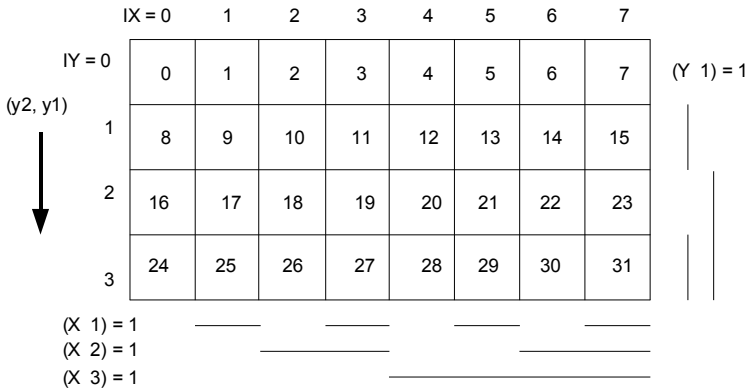

Figure 1-1 A Marquand Chart for a 5-Variable System

Marquand conceived his chart in the binary way (in agreement wit the labeleing practices of today). Written as binary numbers, the identifiers IX, IY produce the variables' validity configuration (Y2, Y1, X3, X2, X1) of all five variables of the system. The identifier of that configuration IS = $(y2\ y1\ x3\ x2\ x1)_2 = 8 \times IY + IX$. Each window in *Figure 1-1* is labeled with the corresponding value of IS.

$IS = (1\ 1\ 1\ 1\ 1)_2 = 31$, while $IS = (0\ 0\ 0\ 0\ 0)_2 = 0$, and $IS = (0\ 1\ 1\ 1\ 1)_2 = 15$

The values of IS follow each other in a natural way. This rule holds true for a Marquand Chart pf any dimension and any shape. The logical distance of a pair of windows on the chart is the sum of the disagreements in bits of their binary identifiers, IS. Two windows that are at the logical distance of one unit possess identifier IS values that differ by 2^k (where k is an integer), thus implying that two windows that are at the logical distance of one unit must fall both in the same row or both in the same column.

Finally, the binary background of the chart leads to the following simple rule: If a Marquand Chart of any size or shape is divided into vertical bands of equal width 2^{k+1}, then any two windows within the same band possessing the horizontal distance of 2^k (half of band) have a logical distance of one unit. The same rule holds for division into horizontal bands of the equal width, 2^{k+1}. Any two windows in the vertical distance of 2^k (both being in the same column, of course) that fall in the same horizontal band, have the logical distance of one unit.

Example 3: Four vertical bands in *Figure 1-1* have the width $2^1 = 2$ windows. For that reason, any two windows at the horizontal distance of 20 (1 window) falling in the same band have the logical distance of one unit; for instance, pairs of windows labeled in IS: (0, 1), (12, 13), (26, 27). But pair (21, 22) which has a horizontal distance of one window, is composed of elements that do fall in the same band; their logical distance is not equal to one unit. Horizontal band division with band width 2 shows that (21, 29) are windows of logical distance of one unit, but that windows (10, 18) are not. Vertical band division with band width 4 indicates that (17, 19) are at logical distance of one unit and the (19, 21) are not.

Returning back to the logical relation between subsystems, four examples are offered. (See *Figure 1-2*)

Example 4. *Figure 1-2a* shows the Existence Function of a system whose subsystems X, Y are completely independent. The system $X \cup Y$ is without constraints. The chart is filled with "1"s to express that every possible validity configuration ***exists***.

Example 5. *Figure 1-2b* shows the Existence Function of a system subjected to some constraints. In general, a given Existence Function can belong to many different sets of constraints. We will mention the most obvious:

1. For $IX \in \{2, 3, 4, 5, 6\}$, the value of IY is uniquely determined. In other words, IY is a function of IX within that domain.
2. For IX = 1, it is IY = 1 XOR 2 (exclusive OR).

   In another form

   $(IX = 1) \rightarrow (Y2 \neq Y1)$
3. For IX = 0, it is IY = (any). In other words,

   $(IX = 0) \rightarrow$ (any one from all)
   (don't care which IY)

4. The input configuration belonging to IX = 7 is forbidden as the circuit has no steady-state for X3 = X2 = X1.

5.

IY \ IX	0	1	2	3	4	5	6	7
0	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1

(a)

IY \ IX	0	1	2	3	4	5	6	7
0	1	0	1	0	1	0	0	0
1	1	1	0	0	0	0	0	0
2	1	1	0	1	0	0	1	0
3	1	0	0	0	0	1	0	0

(b)

IY \ IX	0	1	2	3	4	5	6	7
0	1	0	0	0	0	0	0	0
1	0	1	1	0	1	0	0	0
2	0	0	0	1	0	1	1	0
3	0	0	0	0	0	0	0	1

(c)

IY \ IX	0	1	2	3	4	5	6	7
0	0	0	0	1	0	1	0	1
1	1	1	0	0	0	0	0	0
2	1	0	1	0	1	0	1	0
3	0	0	0	0	0	0	0	0

(d)

Figure 1-2 Examples of Existence Functions

Example 6: *Figure 1-2c* shows the Existence Function of the full adder (Figure 1-3). It is a combinational circuit: A definite output signal configuration belongs to any input signal configuration. In other words, the chart of the Existence Function must have exactly ***one*** non-zero in each column. In another form, IY = f(IX). We say that the subsystem Y is a ***function*** of the subsystem X. Symbolically, IX -> IY. The constraint for the full adder, written in APL, is:

$$((Y\ 1) + 2x(Y\ 2)) = (X\ 3) + (X\ 2) + (X\ 1) \qquad (1.1)$$

The equation means that the sum $\sum_j (X\ j)$ (count of HIGHs at the input of the full adder) is equal to the binary number $(y2\ y1)_2$ (represented in HIGHs at the outputs.

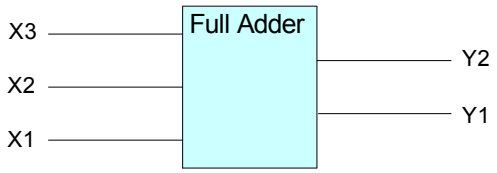

Figure 1-3 A Combinational Circuit

It is important to point out that *Figure 1-2c* is the chart of the Existence Function of the full adder and not the truth table of functions generated by the full adder. The relation between the Existence Function and the truth table is very simple:

1. The Existence Function can be ***replaced*** by a truth table ***uniquely*** if and only if each column of the (normalized) chart of the Existence Function contains exactly one non-zero.
2. The truth table function ((Y k) = 1) -> (Z k) can be deciphered from the Existence Function by reading IX values for which (Y k) = 1.

To get the truth table of the full adder from its Existence Function in *Figure 1-2c*, we start with (Y 1) = 1 to get (Z 1). Configurations with (Y 1) = 1 are all on the rows IY $\in$ {1, 3},a and the Existence Function indicates that only four cases exist with IX $\in$ {1, 2, 4, 7}, so that (Z 1) $\equiv$ (0110 1001). Similarly, (Y 2) = 1 is true only for configurations in rows IY $\in$ {2, 3}. The Existence Function indicates four cases: IX $\in$ {3, 5, 6, 7}, so (Z 2) $\equiv$ (0001 01111).

The complete truth table (presented horizontally, as by the APL programs) is shown in Figure 1-5.

(X 1) ≡ 0 1 0 1 0 1 0 1

(X 2) ≡ 0 0 1 1 0 0 1 1

(X 3) ≡ 0 0 0 0 1 1 1 1

(Z 1) ≡ 0 1 1 0 1 0 0 1

(Z 2) ≡ 0 0 0 1 0 1 1 1

Figure 1-4 Truth Table of Full Adder

Example 7: *Figure 1-2d* shows the Existence Function of a NOR flip-flop with a reset terminal, X3. The conventional diagram of this flip-flop is shown in *Figure 1-5*. The corresponding system is composed of the input subsystem (X j); j = 1, 2, 3, and the output subsystem (Y k); k = 1, 2. The diagram in *Figure 1-5* was postulated as the only constraint of the system. The equations of the circuit, written in APL,

((Y 2) = (⍱ Y 1) ^ (⍱ X 1)) ^ ((Y 1) = (⍱ Y 2) ^ (⍱ X 2) ^ (⍱ X 3))

are satisfied for validity configurations corresponding to windows where the Existence Function (*Figure 1-2d*) is true.

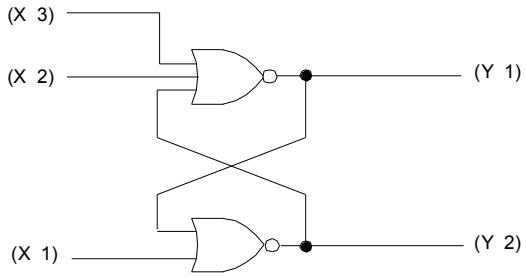

Figure 1-5 NOR Flip-Flop with Reset

The circuit properties can be derived from that function:

1. It is clear that the Existence Function in *Figure 1-2d* cannot be replaced by a truth table because not every column contains exactly one nonzero (see column IX = 0). Thus, the circuit is not combinational but rather sequential (containing feed-backs).
2. There are exactly nine steady states: Two for IX = 0 and one for each IX ∈ {1, 2, 3, 4, 5, 6, 7}.
3. When (X 3) = 1 (reset signal HIGH), then IX ∈ {4, 5, 6, 7}. All four existing validity configurations for that domain (right-hand half of the chart) have (Y 1) = 0 in common. That means that X3 -> Y1, independent of anything else.
4. When (X j) = 0 for all j, then IX = 0 and the circuit can be either of two steady states: IX ∈ {1, 2}, in which case (Y 2) ≠ (Y 1).
5. (Refer to *Figure* 1-6) Starting with the steady state: IS = 16, the change of X1 alone (IX = 0→1, Column IX = 1) produces the unstable state: IS = 17, which goes over to the steady state: IS = 9. The change of X2 alone (IX = 0 →2, column IX = 2) produces the unstable state: IS = 10, which goes over to the stable state: IS = 18.

   A change of X2 alone (IX = 2 → 0) enforces the steady state: IS = 16. Flip-flop transition is thus illustrated.

IS	Y2	Y1	X3	X2	X1
16	1	0	0	0	0
17	1	0	0	0	1
9	0	1	0	0	1
9	0	1	0	0	1
8	0	1	0	0	0
8	0	1	0	0	0
10	0	1	0	1	0
18	1	0	0	1	0
18		0	0	1	0
16	1	0	0	0	0

Figure 1-6 State Transaction for Flip-Flop of Figure 1-5

The reader is now invited to go to Chapter 2, which illustrates the use of the program SYSTEM to specify Boolean functions either by the procedure SPACE (Existence Function development) or by the procedure TABLE, producing a truth table of functions, either from their sufficient functions or by listing).

The library program SYSTEM has two groups of procedures. The first prepares truth tables or Existence Functions (discriminants) of a system subjected to a set of constraints. The second group contains important design procedures for the special treatment of Boolean functions such as charting, minimization of ∑∏ and ∏∑ forms, listing prime implicants, and evaluation of the Boolean difference. Some of the algorithms used in the APL programs differ from those found in teaching texts --- the triadic ordering of implicants, minimization by extension of a ∑∏ form, and multiple-output design optimization based on S-minimization of a mosaic Boolean function are mentioned to name he most important. Explanations of these algorithms will be given in he second section of this text, and logical instruments will be offered there as efficient means of teaching the basic concepts.

The library program BOOL solves systems of Boolean equations of a general type by enumeration. The procedure is entered by calling BILL, and the equations are entered by calling FORMULA. The first literals of the alphabet represent the Boolean constants (for instance: A, b, c, D), and the literals that follow them in their natural sequence represent the unknowns (for instance: E, F). The values 0 (false) and 1 (true) may be used (alone!) on the right-hand side of the formula only. The sum of products form must be used on both sides of the formula. Only two relations between the sides are accepted by the programs, EQUIVALENCE (=) and IMPLICATION (→) (the APL right-arrow). Underlining may be used to represent negation.

Example 8. Examples of correctly-composed formulas (It does not matter how many variables are constants and how many are unknowns):

C$\underline{A}$ + $\underline{B}$D = $\underline{AB}$ + C$\underline{D}$A

A$\underline{C}$D + B = 0

E$\underline{D}$ +$\underline{E}$D = 1

A$\underline{B}$D -> C$\underline{E}$

DC + C$\underline{A}$ -> $\underline{E}$ +BA

EDCB -> A + B + $\underline{D}$

Note: Spaces within a product or around the signs are acceptable. The sign “+” means OR; the sign “->” means “implies”.