11 Chapter 11 Designing with MSI-LSI

11.1 Introduction

Once a set of design constraints have been established, there are several options available to the designer to be used in implementation. Circuit built with SSI-level logic may be faster, may be minimal in parts count, may be irreducible, and, therefore, may be more testable than those built using MSI or LSI devices.[1] SSI-level designs take a relatively longer time to develop, debug, and document. It is acknowledged to be the most difficult. The criteria that determine the desirability of doing an implementation with SSI-level logic are:

1. The required speed is faster than would be available with any other implementation;

and

2. The anticipated production volume justifies the expense of doing an SSI-level design.

Where total testability is not a major concern, as is true for most commercial-level applications, MSI design with multiplexers and functions blocks (think IP) is attractive (reduces TTM – time to market). While multiplexers are non-minimal implementations of their output functions, which constrain their testability, they are easier to deal with and allow reduction in board space.[2] Speed and power requirements are the determining factors in choosing to use multiplexers and other MSI function blocks.

Programmable multiplexers (PMUXs), gate arrays (PGA, CGA), logic arrays (PLA, FPLA) and array logic devices (PAL) are available now in a number of configurations and sizes. Most of these are intended to assist in reducing the parts count for implementation of combinational functions. Some of the PALs have registered outputs and feedback paths and are for use in sequential logic implementation. All of there devices are contributing toward modular hardware designs.

Sequential control functions may be implemented with microprograms using PROM/ROMs with the more complex controls using a microprogram sequencer.[3] Higher-speed control functions are being implemented using bipolar bit-slice devices (such as the Am2910), while less speed-restricted applications use one or more of the microprocessor/microcomputers (such as the Intel 8085). All of these devices are considered to be LSI and all of them require a software investment. The bit-slice devices require microprogramming, which may be accomplished with a pseudo-assembly –level language and a development system. [Meta-assembler.] The fixed-instruction-set microcomputers are programmable in assembly or higher-level languages, with some of the new devices to be programmed in ***PASCAL*** or a similar algorithmic language. [Today it would be C^{++} or Pearl.]

As the implementation shifts from SSI to LSI, software design techniques, specifically modularity and structured programming, begin to become mandatory. Testing is no longer a hardware checkout operation, but requires software diagnostic packages. One advantage of LSI is the feasibility of putting the system diagnostic routines on-board the PROMs, to simplify design debug and field testing.

---

[1] Applies to the density levels of RTL in today's design kit.

[2] This discussion was originally based on discrete design. Mid-1970s. MSI and LSI were in vogue, LSI was being thought of, and ASICs hadn't arrived with a vengeance as yet.

Software development costs in computer systems have far exceeded those of hardware development. This trend is being repeated in LSI designs with the microprogramming development costs overriding the hardware costs.

11.2 SSI Design

To develop a minimal or optimal two- or three-level NAND logic circuit, with the third level for inversion of the logical variables, the equation of function y must be solved for its minimal ΣΠ-form. This is accomplished using the APL function:

MINIMA (Z 1)

which produces the desired expression.

To develop a minimal or optimal two- or three-level NOR logic circuit, the equation of the function y must be solved for its ΠΣ form. This is accomplished by using the APL function:

MINIMA ($\underline{Z}$ 1)

which produces the desired expression.

Conversion techniques exist for NOR-to-NAND and for NAND-to-NOR without restarting from the original expressions of the function y (refer to any text for a beginning course in logic design); however, the resulting network is not guaranteed to be minimal. Where an expression is factorable to produce terms of the form ($a\,\underline{b} + \underline{a}\,b$), EXOR gates may be used to reduce gate count and to simplify implementation.

Fan-in requirements can be achieved by factoring the minimal ΠΣ or ΣΠ forms. The fan-in requirements present in early SSI designs are more relaxed today with the allowances of eight (8) or more inputs allowed per gate. [The dragging of the load on the gate performance is, however, still problematic.] Fan-out requirements are generally alleviated by the use of buffer-drivers, and duplicate parallel paths. Present designs[4] use a conservative limit of seven (7) loads per output for devices rated at 10 loads.

11.3 Gate versus Connection Minimization

Muruga and Lai *(Muruga and Lei, "Minimization of Logic Networks under a Gneralized Cost Function", IEEE Trans., Sept. 1976, pp. 7893-907)* reported on the calculations of the minimal networks for NOR gates for all functions of 3 or fewer variables, and for some of the functions of 4 variables. For the 77 P-equivalence non-trivial class representative functions of 3 or fewer variables, there are only two functions for which the minimal networks under GCM (gate reduction emplasis) and CGM (connection reduction emphasis) differ.

They are:

$$\mathbf{Y_1 = X_2 \oplus X_1 \oplus X_0}$$

and

$$\mathbf{Y_2 = X_2X_1X_0 + \underline{X}_2\,(\underline{X}_1 + \underline{X}_0\,)}$$

(Underlining equals negation)

For these two functions, the minimal ΠΣ form was investigated and found to give:

- the same number of gates as does the GCM reduction
- more connections than does the CGM version
- and fewer stages (gate levels and, therefore, gate delays).

The GCM reduction reduces the gate count by one over the minimal ΠΣ form, but increases the connections by one. In one case, there is an added stage delay.

The minimal ΠΣ forms of the two functions are shown in *Figure 11-1*a and b. A table comparing the various implementations in terms of number of gates, number of connections, and number of levels, is shown in *Table 11-1 .*

[4] 1970s – late 1970s

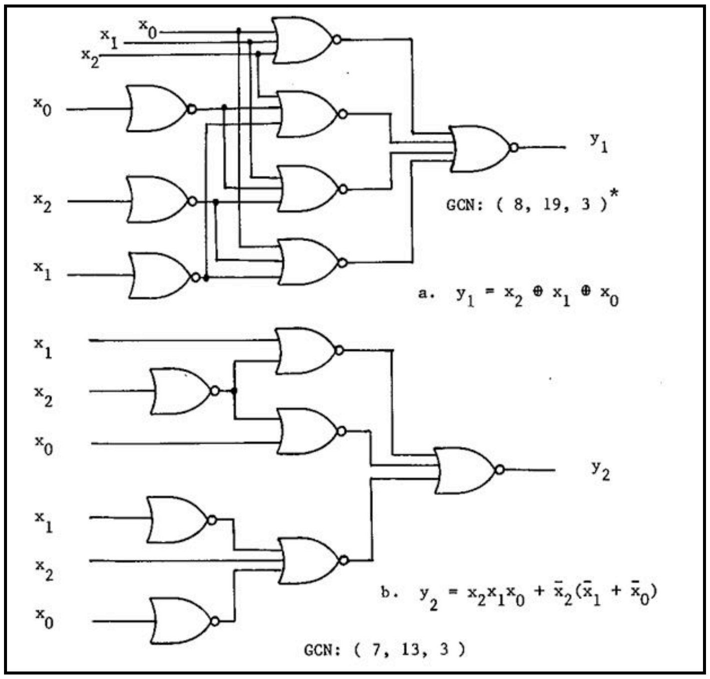


Figure 11-1 a and b. Minimal $\Pi\Sigma$ forms of y_1 and y_2

- = (# Gates, # Connections, # Gate Levels)

Table 11-1 Minimal $\Pi\Sigma$ forms of y_1 and y_2 Compared

Function	Minimal	GCM	CGM
y1	(8, 19, 3)	(7, 20, 4)	(8, 16, 6)
			(8, 16, 5)
			(8, 16, 6)
y2	(7, 13, 3)	(6, 14, 3)	(7, 12, 4)
			(7, 12, 5)

Among 312 representative functions of P-equivalence classes of 4 variables requiring at most 5 NOR gates under GCM, 26 functions were found to have different minimal networks under CGM design constraints.

Of the 26 functions:

1) For 4 of the functions, they had GCM and CGM minimal networks where the number of gates levels or stages were equal. In 22 cases, the number of levels was at least one higher for CGM design constraints.
2) For 7 of the functions, that were implemented in CGM, CGM had a gate count higher then the GCM version (higher by 1).
3) For 8 of the functions implemented in CGM, CGM had 2 fewer connections then the GCM versions of those functions. The rest had a connection count that was lower by 1.
4) For some of the functions, a gate count for the GCM version lower than the minimal $\Pi\Sigma$ form version was obtained by the introduction of redundancy, which in turn reduced testability.
5) For some of the functions, a gate count reduction was accomplished through the use of a factored, equivalent form which added one gate level.

Their conclusion was to use the gate criteria in minimization and design. (They were concerned specifically with chip area, i.e., die size, in their paper.). Our conclusion is to use the minimal $\Pi\Sigma$ and $\Sigma\Pi$ forms, factoring where possible for reduction but maintaining irredundancy (i.e., no redundancy allowed).

The length of time available to design the network is an overriding constraint in all cases. For the minimal $\Pi\Sigma$ and $\Sigma\Pi$ forms, the APL function MINIMA is available as a design tool. *Figure 11-2* examines three implementations of the function Muruga and Lai indexed as EBFF. (Their index uses HEX notation to fill in the rows of a 4-variable Marquand map.) It represents the minimal $\Pi\Sigma$ form, a GCM (minimal gate) version, and a CGM (minimal connection) version of the function.

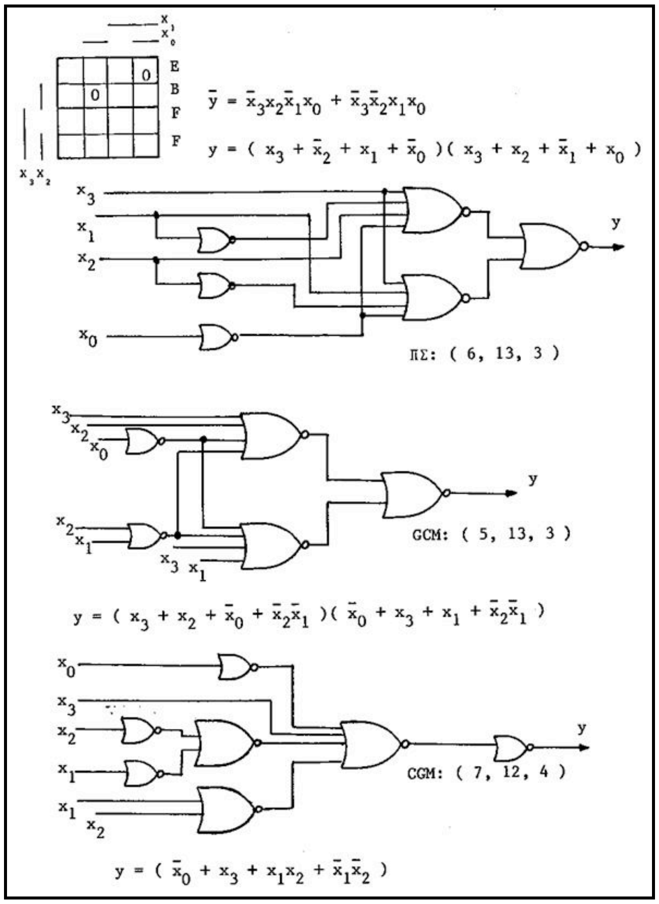

x_1
x_0
E
B
F
F
$x_3 x_2$
$\bar{y} = \bar{x}_3 x_2 \bar{x}_1 x_0 + \bar{x}_3 \bar{x}_2 x_1 x_0$
$y = (x_3 + \bar{x}_2 + x_1 + \bar{x}_0)(x_3 + x_2 + \bar{x}_1 + x_0)$
x_3
x_1
x_2
x_0
y
$\Pi\Sigma$: (6, 13, 3)
x_3
x_2
x_0
x_2
x_1
x_3 x_1
y
GCM: (5, 13, 3)
$y = (x_3 + x_2 + \bar{x}_0 + \bar{x}_2 \bar{x}_1)(\bar{x}_0 + x_3 + x_1 + \bar{x}_2 \bar{x}_1)$
x_0
x_3
x_2
x_1
x_1
x_2
y
CGM: (7, 12, 4)
$y = (\bar{x}_0 + x_3 + x_1 x_2 + \bar{x}_1 \bar{x}_2)$

Figure 11-2 Comparison of SSI Implementations

For the two functions of three variables Y_1 and Y_2 described earlier, the SSI implementation of Y_1 is improved by using EXOR gates. (*Figure 11-3*.) A comparison between the implementation of the minimal $\Pi\Sigma$ form of y in NOR gates and the EXOR implementation is shown in *Table 11-2*.

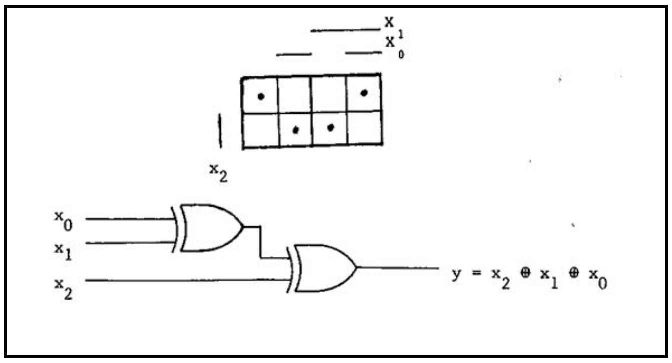


Figure 11-3 An EXOR Implementation of y_1 and Comparison of the Minimal $\Pi\Sigma$ Form (data from datasheets – 1970s)

Table 11-2 An EXOR Implementation of y_1 and Comparison of the Minimal $\Pi\Sigma$ Form

Item	**Minimal $\Pi\Sigma$ NOR**	**EXOR**
Cost	0.80	0.40
Levels	3	2
Packages`4	1	
Time (Speed)	32ns	28ns
Power	56mW	150mW

11.4 MSI Design

The decision to use SSI or MSI is dependent upon the complexity of the function being impolemented and upon esign constraints such as timing, board space, and power consumption.

Multiplexer chips[5] for 3, 4 and 5 variables are easy to use and input requirements can be read directly from the Marquand Map of the function. (Karnough Maps require permutation of certain columns to achieve a column-to-input order and are, therefore, not recommended here.) Muxs are available in some cases with true and complimented outputs and have 1, 2, or 3 chip-select or enable pins (S_0, S_1, S_2), providing design flexibility.

Multiplexers are formed internally from two-level logic, which makes them competitive in speed with SSI gate implementations. In some cases, they may even be faster than SSI doe to the reduction of interpackage[6] time delays.

The use of multiplexers reduces package count at the cost of increased power consumption per package. [Reduces macro count at the cost of increased power for the more complex macro.]

[5] for today read "macros"

[6] for today, read "intermacro" or "interconnect" delay

The other factors that must be considered are:

1) The number of connections
2) The loading on the variables
3) the loading on power and ground

The use of multiplexers may or may not render the design "flexible", i.e., able to be altered with changes in requirements and specifications. Any function implemented in multiplexers is not necessarily minimal. Coverage will be column-minimal and the cost of possible increased loading. *Figure 11-4* presents the one-of-four multiplexer in detail and an example derivation of input functions using a Marquand Map.

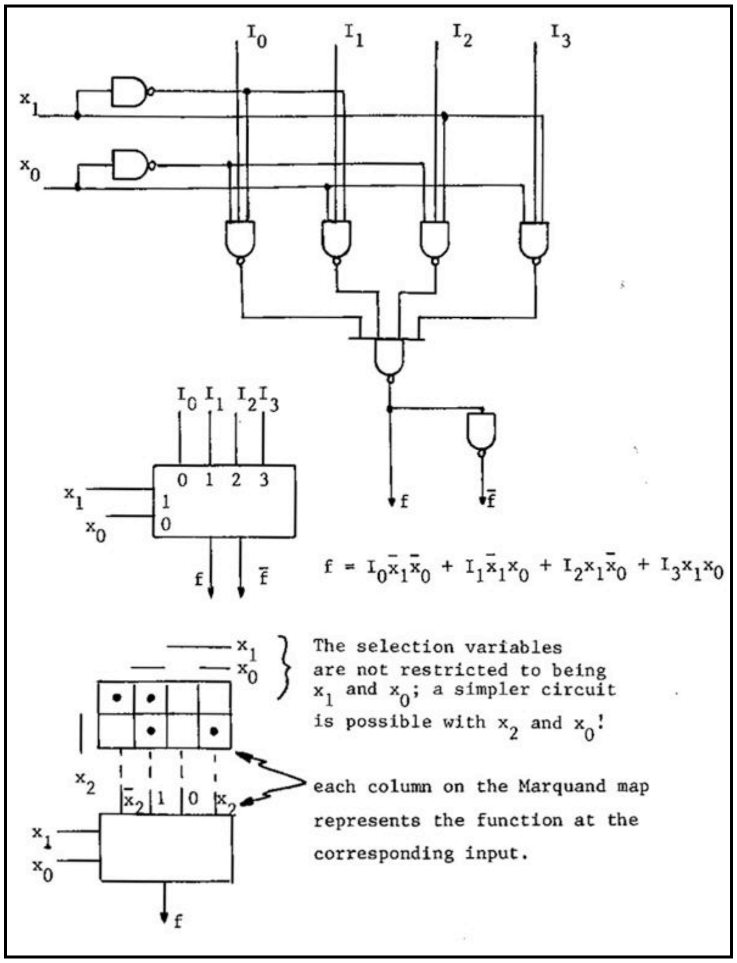

I_0
I_1
I_2
I_3
x_1
x_0
f
$\bar{f}$
$I_0\ I_1\ I_2 I_3$
0 1 2 3
1
0
$f = I_0\bar{x}_1\bar{x}_0 + I_1\bar{x}_1x_0 + I_2x_1\bar{x}_0 + I_3x_1x_0$
The selection variables are not restricted to being x_1 and x_0; a simpler circuit is possible with x_2 and x_0!
each column on the Marquand map represents the function at the corresponding input.
x_2
$\bar{x}_2$ 1 0 x_2

Figure 11-4 One-of-Four Multiplexer

Figure 11-5 presents the multiplexer implementations of Y1 and Y2 and a table summarizing the differences between the multiplexer, NOR Gate, and EXOR implementations is shown in *Table 11-3.*

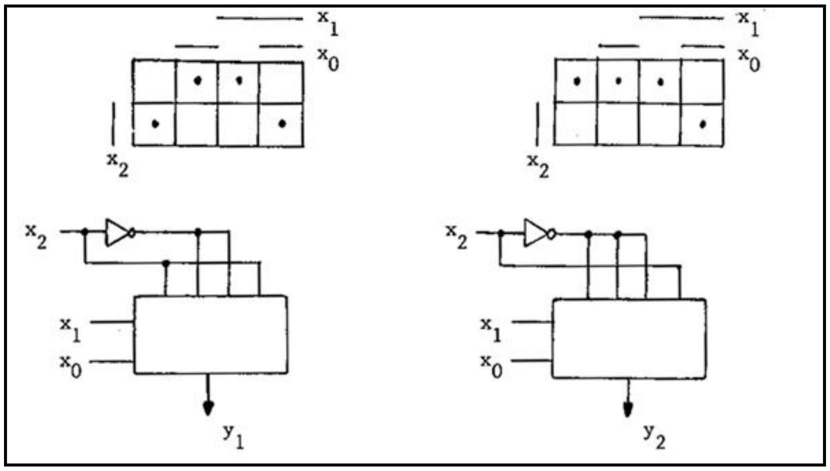


Figure 11-5 Implementation of y1 and y2 with Multiplexers

Table 11-3 Comparing the Implementations:

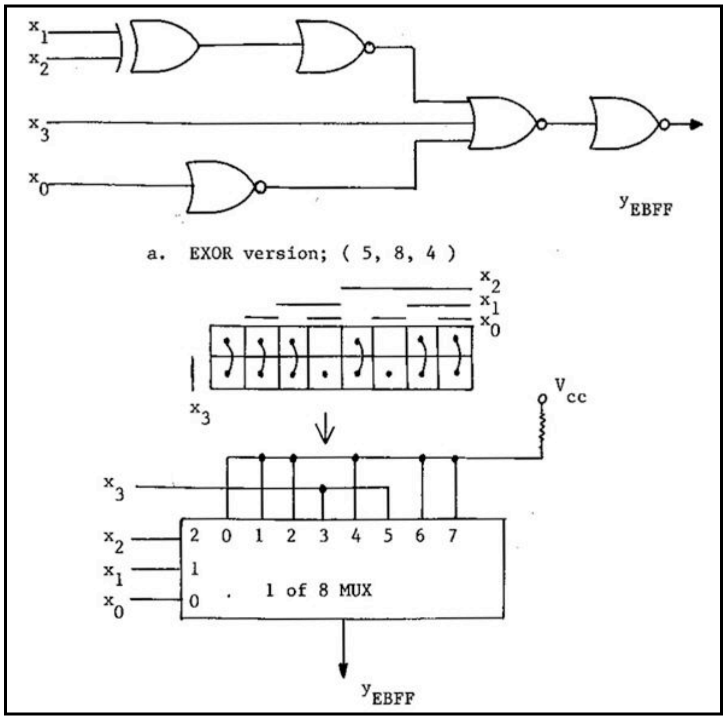
Item	**Value**	**Comment**
Time Delay:	21ns	faster than a NOR gate (in the late 1970s)
Power:	200mW	More Power, More Heat
Package count:	1	Same as for EXOR version
Cost:	$1.50	(approx) – More Expensive

Differences between the multiplexer, NOR Gate, and EXOR implementations – See Table 11-2.

Multiplexer sizes are of 1-of-2, 1-of-4, 1-of-8 and 1-of-16 (an oversized chip in those days). When a function larger than five variables is to be implemented, multiplexers may be cascaded.

The 4-variable function EBFF is shown in *Figure 11-6a*, implemented using an EXOR gate to improve the connection count (8 instead of 12 for the same number of gates) at the expense of an added level and its delay.

Figure 11-6b shows the clean simplicity of a multiplexer implementation of the same function and the associated Marquand Map.



b. Multiplexer Version: (1, 11, 1)

Figure 11-6 The Four-Variable Function y_{EBFF} from Muruga and Lai's Paper

Multiple-output problems are designed faster using multiplexers. The problem discussed in Chapter 8 and shown in Figure 8 (reference) is shown implemented with multiplexers in *Figure 11-7*. Included is a table (*Table 11-4*) comparing:

1) The individual output function SSI version
2) A reduced multiple-output SSI version
3) and a multiplexer version

The multiplexer version in this case

1) runs slightly faster
2) has fewer connections
3) uses less board space
4) can be designed relatively faste
5) and is easily debugged
6) all at a cost of 2-4 times the power consumption.

The choice is dependent upon the design constraint of specified allowable power consumption and heat dissipation capability.

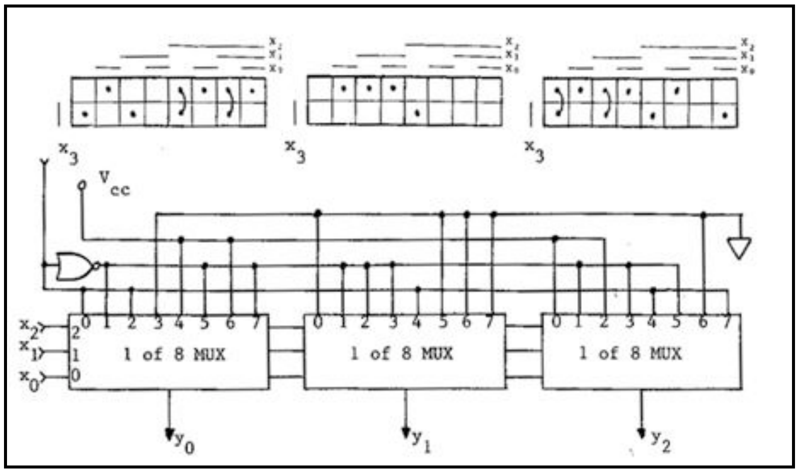


Figure 11-7 Multiplexer Implementation of the Multiple Output Problem

Table 11-4 Multiple Output Problem Implementation Comparison

Version	Gates	Packages	Connections	Spares	Levels	Power (mW)	Speed (ns)
SSI	17	6	43	0	3	170	30
Mult. Out.	15	6	38	3 gates	3	150	30
MUX	3 MUX + 1	4	34	5 inverters	2	685	18
						445	26

As more variables are introduced, choices in which multiplexers to use are added. The six-variable example discussed in Chapter 8 is implemented using a 1-of-16 MUS and again using 1-of-8 MUIXs in *Figure 11-8*

The map of the function, its equation, and a table comparing the three implementations --- SSI and the two MUX versions --- are presented in *Figure 11-9* and *Table 11-5.*

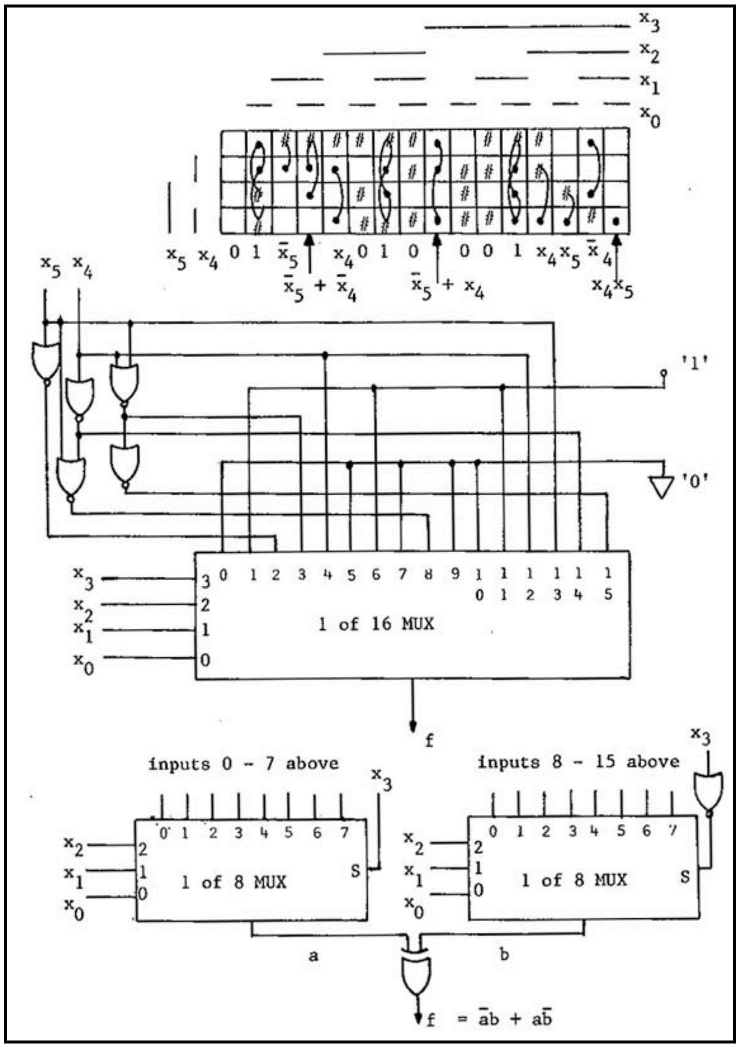


Figure 11-8 Svoboda's Six-Variable Example done with Multiplexers

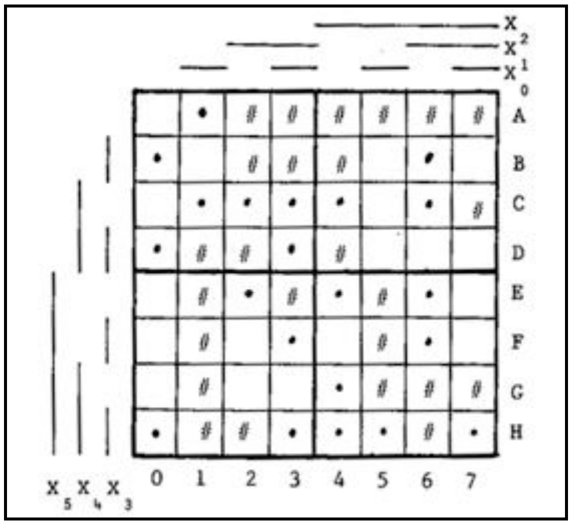


Figure 11-9 Six-Variable Marquand Map for the Example

The equation is:

$$y = \bar{x}_5 x_3 \bar{x}_1 x_0 + \bar{x}_4 \bar{x}_3 x_1 \bar{x}_0 + \bar{x}_4 x_2 x_1 \bar{x}_0 + \bar{x}_5 \bar{x}_3 \bar{x}_2 x_0 + \bar{x}_5 \bar{x}_2 x_1$$
$$+ \bar{x}_3 x_2 \bar{x}_0 + x_5 x_3 \bar{x}_2 x_0 + x_5 x_4 x_3$$

Table 11-5 Comparison of Implementations of the Six-Variable Example

Version	Stages	Connections	Packages	Time/speed	Power	Approx. Cost
SSI (NAND)	3	43	6	30ns	220mW	1.20
MSI, 1-of-8 MUX, NOR, EXOR	4	37	4	51ns	500mW	2.60
MSI, 1-of-16 MUX, NOR, INV	4	28	3	41ns	316mW	1.80

(Data from datasheets (late 1970s), Numbers are Typical Case)

11.5 LSI Design Techniques

There are in existence a number of microprogrammable devices which may be used to reduce parts count and, therefore, required board space. A few examples of the most common of these are called out in *Table 11-6* Programmable devices, with the exception of ROMs, are effective in design situations where the number of input variables is large and the number of active logic states is small --- that is, in cases where the logical map of the function is sparsely populated.

Table 11-6 Examples of Microprogrammable Devices (late 1970s)

Device	Example Product (late 1970s)
Programmable Multiplexers	Raytheon 29693 PMUX (10 inputs, 1 term/MUX input, 4 1-of-8 MUXs)
FPLA	Signetics N82S101/10 (16 inputs, 48 terms, 8 outputs, 50ns)
PLA	National DM 75 7516 (19 inputs, 70 terms, 8 outputs, 150ns)
PAL	Monolithic Memories PAL10H8 (10 inputs, 8 outputs, 4 registeresd outputs, 64 terms)
Registered PAL	Monolithic Memories PAL16X4 (8 inputs, 8 outputs, 4 registered outputs, 64 terms)
ROM: EROM	Signetics SN 74S271/371 (256x8 with 45ns Access Time)
ROM: PROM	Signetics SN 74S271/371 (256x8 with 45ns Access Time)
ROM: Registered ROM	AMD Am27S26/27 (512x8 with 20ns Cp-to-Output time – prelim)

11.5.1 Programmable Multiplexers

One of the programmable is the Raytheon 29693 PMUX listed in Table ----. This device has 10 inputs, all inverted, and 4 inverted outputs. Logically it is equivalent to four 1-of-8 multiplexers with OR gates at each of the 10 inputs. The multiple-output problem presented earlier is of the type suitable for a PMUX implementation because:

1) The MUXs have common select lines
2) The number of input variables is less than 10

Figure 11-7 is very nearly a program for the 29693 PMUX. The only change required is to use X3 + X3 as the input where Vcc is shown. Documentation of this device is one to four Marquand Maps (one for each output used) drawn with select variables as column indices and inpout variables as row indices. Note that an external inverter for X3 would be required. Note also that this example under-utilizes the PMUX ability and is for demonstration only (this would never be built).

11.5.2 Programmable Logic Arrays

A programmable logic array (PLA) is an LSI implementation of the classic digital net, a sum-of-products form for positive logic. It is general-purpose and user-definable within limits. Both the variable composition of the individual product terms and the assignments of the terms to different outputs are specified by the designer.

PLAs are characterized by the following attributes:

1) The number of inputs
2) Buffered or unbuffered inputs
3) The number of minterms or product terms
4) The number of outputs
5) The characteristics of the outputs: (late 1970s)
    a. TTL
    b. Open-Collector (OC)
    c. Tri-state
6) Programmable output inversion
7) Access time

There are two basic types of PLAs, determined by how they are programmed. A factory- or mask-programmed PLA is typically larger than 96 minterms. Factory programming uses the metal-mask technology. [7]

A field-programmable PLA (FPLA) typically has 48 minterms and may use one of three programming technologies:

1) Nichrome fuse
2) Polysilicon fuse
3) Avalanche induced migration

Given a PLA with the following characteristics:

1) 16 inputs, either true or inverted, to any AND gate
2) 8 outputs, either true ior inverted
3) 48 product terms to any OR gate
4) Typical access time of 50ns
5) Power dissipation of 620mW (Typ.)
6) 28-pin DIP (dual-in-line package) (approximate space of 3 16-pin DIPs)

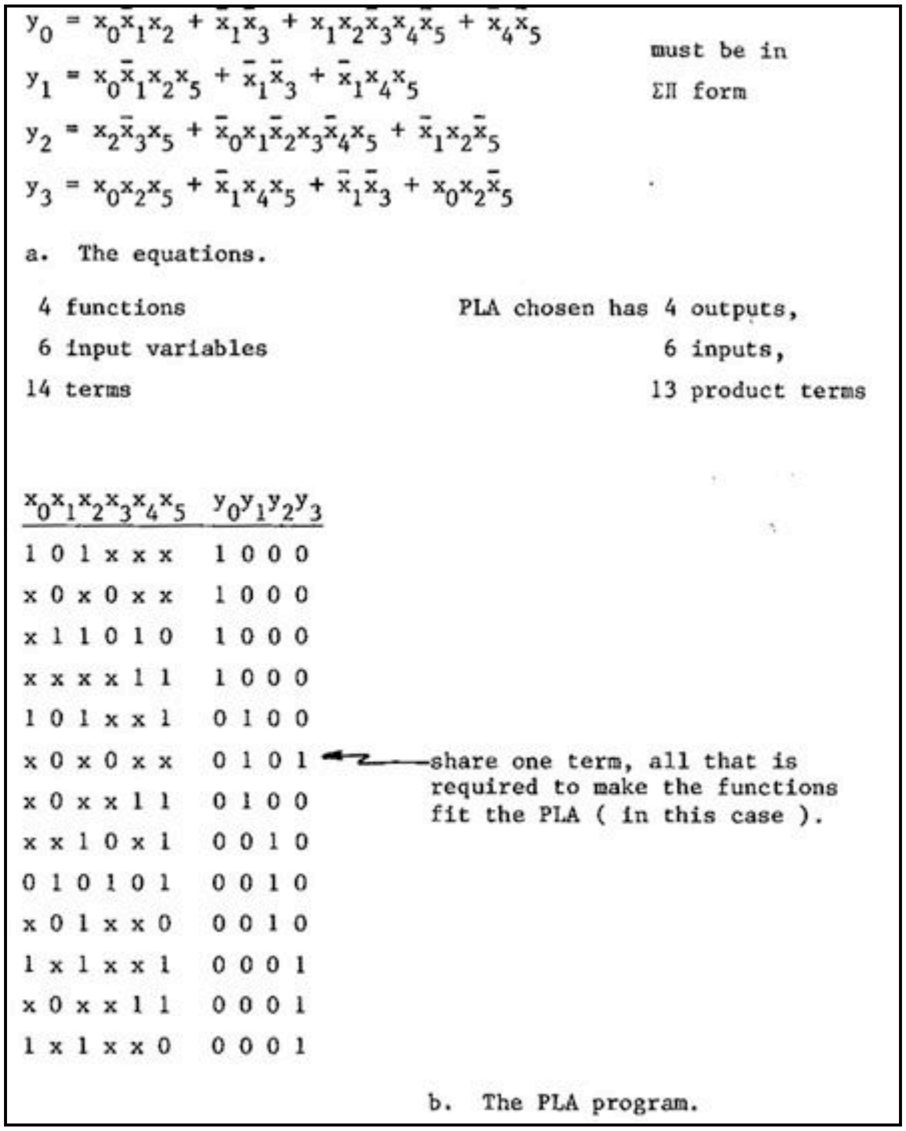
[7] sizing, technology will have varied by now of course, but the design principles remain.

The six-variable functions shown in *Figure 11-10a* are to be implemented. These functions are not necessarily minimal. These are, using the available parts above:

1) 6 inputs required
2) 14 product terms, of which 3 are seen to be identical
3) and 4 outputs.

If the equations given had not been in a sum-of-products form, expansion would have had to be performed to obtain this format as each output is a sum-of-products function. Internally the PLA is an AND-OR net with an inverter programmable at the output if desired.

For the functions shown in *Figure 11-10*a, a single PLA may be programmed as shown in *Figure 11-10*b. No minimization needs to be performed.



$$y_0 = x_0\bar{x}_1x_2 + \bar{x}_1\bar{x}_3 + x_1x_2\bar{x}_3x_4\bar{x}_5 + \bar{x}_4\bar{x}_5$$

$$y_1 = x_0\bar{x}_1x_2x_5 + \bar{x}_1\bar{x}_3 + \bar{x}_1x_4x_5$$

$$y_2 = x_2\bar{x}_3x_5 + \bar{x}_0x_1\bar{x}_2x_3\bar{x}_4x_5 + \bar{x}_1x_2\bar{x}_5$$

$$y_3 = x_0x_2x_5 + \bar{x}_1x_4x_5 + \bar{x}_1\bar{x}_3 + x_0x_2\bar{x}_5$$

must be in
ΣΠ form

a. The equations.

4 functions	PLA chosen has 4 outputs,
6 input variables	6 inputs,
14 terms	13 product terms

$x_0x_1x_2x_3x_4x_5$	$y_0y_1y_2y_3$	
1 0 1 x x x	1 0 0 0	
x 0 x 0 x x	1 0 0 0	
x 1 1 0 1 0	1 0 0 0	
x x x x 1 1	1 0 0 0	
1 0 1 x x 1	0 1 0 0	
x 0 x 0 x x	0 1 0 1	← share one term, all that is required to make the functions fit the PLA (in this case).
x 0 x x 1 1	0 1 0 0	
x x 1 0 x 1	0 0 1 0	
0 1 0 1 0 1	0 0 1 0	
x 0 1 x x 0	0 0 1 0	
1 x 1 x x 1	0 0 0 1	
x 0 x x 1 1	0 0 0 1	
1 x 1 x x 0	0 0 0 1	

b. The PLA program.

Figure 11-10 Designing with a Small PLA

Svoboda's six-variable problem could easily be implemented with one PLA-type device. There are 20 product terms required, less than half the number available in the smallest of these devices. The product terms can be trivially taken from the Marquand Map of the function without minimization. The trade-offs are increased power consumption for less board space, a simpler design effort (design time), and reduced testability (as introduced by redundancy).

Note that under-utilization of the LSI devices is quite common in practice and is cost-effective for commercial applications.

Where the product terms exceed the number (48, 96, whatever) available by a small amount, some minimization may be performed to reduce the function to fit the boundaries. (Physical constraints forcing better design practice.) Svoboda's multiple-output minimization is suitable for these cases.

Where there are enough product terms to warrant it (double or triple the number available in one PLA), a second PLA may be paralleled with the first to produce the added capability. All input and output lines would be common.

Where the number of outputs exceeds that available (8) and a simple added SSI or MUX unit is not sufficient to handle the overflow, PLAs may be paralleled where the input lines are tied common and the output lines are separate.

Combinations of these two schemes may be used to implement odd-sized problems. Note that open-collector outputs and low-level logic is required for any of these cases.[8]

An important point with PLAs is that they are pure digital internally. In cases where the design is sensitive to signal-line glitches, PLAs may be preferable to naked ROMs and should be considered. It should be noted that[9], while an 18-input ROM would require 256K internal cells and is several years away by present technology (prophetic – we met that an more!), PLAs can provide a subset of the same logical space today (late 1970s).

ASICs, not even a gleam in someone's eye at the time of this text have already lost half their production starts to very big FPGA. Designs under four million equivalent gates can be effectively done with FPGAs. In the late1970s, we had not yet envisioned millions and millions of gates. We were trying to develop methodology for what we had and what would follow.

CBA arrays were better than standard cells, and then in a flash, they lost half their starts when a new approach made standard cells smaller in die area than CBAs. Highly customized ASICs, where the designer chose a size and crammed a design into it, lasted 6-10 years and then vanished almost overnight as software allowed designers to lay out the die and take control of die size. Component houses with their own fab lines dropped the fab lines, again, almost overnight, as the size of the wafers surged from 3 to 6 to 8 to 12 inches, they are looking into 18 inch wafer (and money is the hold-up – a fab is expensive); the process dropped from 5 micron to 0.02 micron (20 nanometers) and less, and the overwhelming cost of the unique and individually developed software tools gave way to mass-produced EDA tools from firms that specialized and the foundries still standing took over all the library development in-house. From picking and choosing process foundry for a design based on technology, the choice is usually driven by the space available in a wafer production line.

Changes in technology happen fast. Methodology remains.

11.5.3 Programmable Array Logic

PALs are also referred to as programmable gate arrays. The PAL product line of Monolithic Memories is characterized by 8 to 16 inputs, available internally in true and complemented form, and 2 to 8 outputs, with a varying number of product terms. The low end of the product line is the PAL10H8 [late 1970s] with 10 input variables, 8 outputs, and two product terms per output.

A PAL is different from a PLA in that the number of terms per output is fixed, imposing more constraints on the designer. Some PALs are available with feedback and with registered output and feedback. While a PLA replaces SSI combinational logic, PALs exist that can replace sequential logic. The PAL16R6, for example, has 8 inputs, 8 outputs of which 6 are registered, with all outputs fed back. The interested designer should refer to the ***Monolithic Memories PAL Handbook***. [Monolithic Memories, Inc. "PAL Programmable Array Logic Handbook", third edition. 1983.][10]

---

[8] Someone familiar with the parts available at this time needs to comment on this statement, since technology switched from bipolar to CMOS, and NAND and NOR gates are the style of the day.

[9] Over time, the phrase "note that" and all its derivatives became entrenched on my hit list. Along with “In order to”. I will try to get back to kill them-----

[10] No version of this referenced document appears to be on the web at this time.

See Wikipedia. "*MMI was founded in 1969 by former Fairchild Semiconductor engineer Ze'ev Drori. MMI was acquired by Advanced Micro Devices (AMD) in 1987, though AMD later spun off their programmable logic division as Vantis, which was then acquired by Lattice Semiconductor*." Anyone with additional information should put it up on the web. Anyone with access to the PAL book – consider adding it to the vintage material on-line. Bit-savers collects these things.

The six-variable problem of Svoboda requires 20 product terms, 6 inputs, and 1 output. By using a small amount of minimization to reduce the number of product terms, a PAL could be utilized. *Figure 11-11* shows the program connections to construct the solution using a PAL10H8 and an OR gate. The terms programmed are from the multiplexer implementation of the problem shown in Figure 11-8. Column minimization was applied to the Marquand Map of the function in both cases. The PAL replaces two 1-of-8 MUXs and five NOR gates. The OR gate is in exchange for the EXOR of *Figure 11-8*.

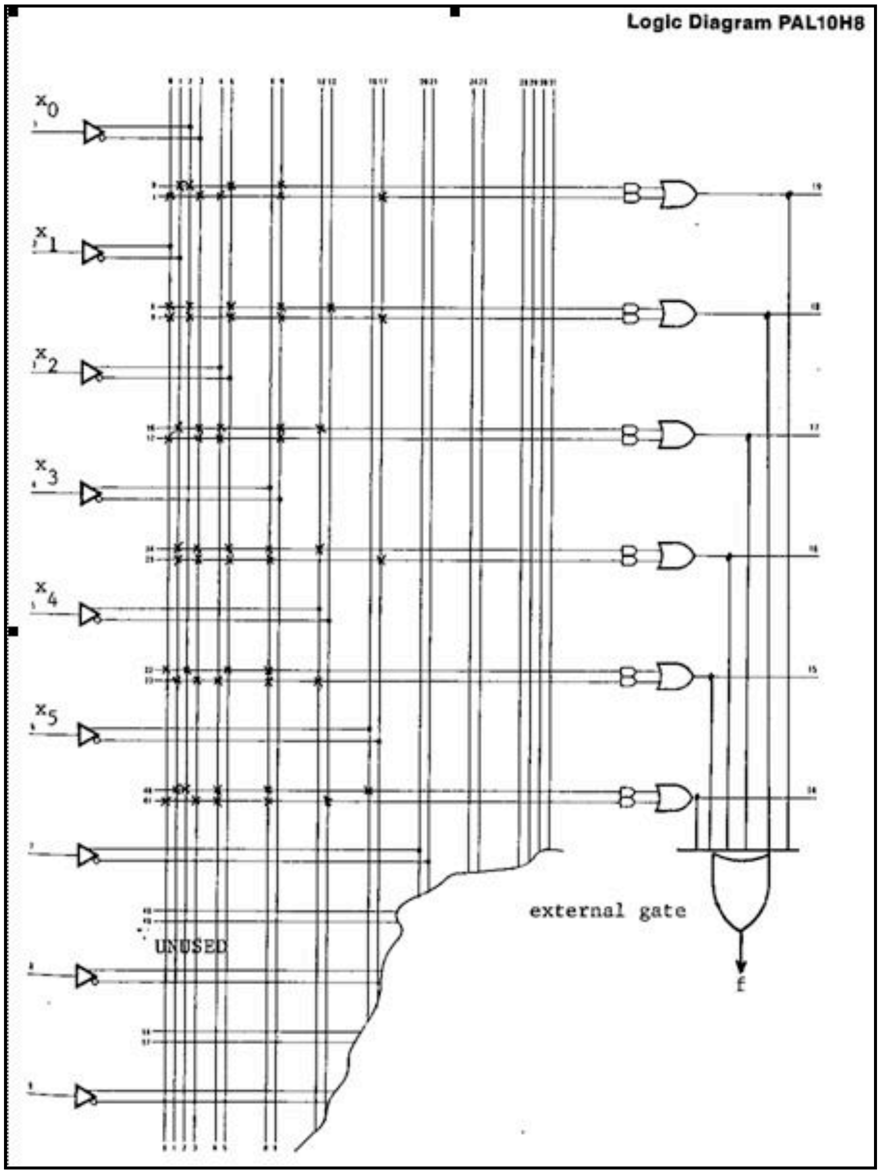


Figure 11-11 Implementation of the Six-Variable Problem with a PAL

11.5.4 Design Example

Using Baucham's modified Hamming Code for high-speed single-error correct, double- and some triple-error detection(*Figure 11-12*), which is for use with 16-word memories, most of the parity equations of the code can be implemented via simple SSI and MSI devices.

$$\begin{array}{cccccccccccccccccccccc}
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&0&0&0&0&0&0&1&1&1&1&1&1&0&0&0&1&0&0&0&0\\
0&0&1&1&1&0&0&0&1&1&1&0&0&0&1&0&0&0&1&0&0&0\\
1&0&1&0&0&1&1&0&1&0&0&1&1&0&0&1&0&0&0&1&0&0\\
0&0&0&1&0&1&0&1&0&1&0&1&0&1&1&1&0&0&0&0&1&0\\
0&1&0&0&1&0&1&1&0&0&1&0&1&0&1&1&0&0&0&0&0&1\\
I_{15}&I_{14}&I_{13}&I_{12}&I_{11}&I_{10}&I_9&I_8&I_7&I_6&I_5&I_4&I_3&I_2&I_1&I_0&\bar{C}_5&C_4&\bar{C}_3&C_2&\bar{C}_1&\bar{C}_D
\end{array} = H$$

C_D, C_1, C_3, C_5 are all complements (odd parity)

C_2, C_4 are even parity sums

Syndrome bits are computed by:

$S_i = C_i^1 + C_i$ C_i^1 is computed upon receipt of the data

C_i is computed and stored and transmitted with the data

$$S_D = P_e\ (\ C_1, \ldots, C_5, C_D\ ) + P_o\ (\ D_0, \ldots, D_7\ ) + P_o\ (\ D_8, \ldots, D_{15}\ )$$

P_e = even parity

P_o = odd parity sum

C_i = check bit

D_i = data or information bit

Figure 11-12 Basham's Modified Hamming Code

The parity equations for the generation of check bits may be implemented with six 74S280 parity generators (*Figure 11-13*a).

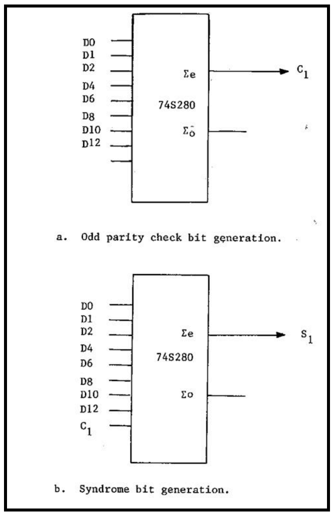


Figure 11-13 Parity and Check Bit Generation

Decode logic is also similarly based on parity generators to provide the syndrome check bits (*Figure 11-13*b). The overall syndrome bit SD is computed using all data bits and all check bits. Using Pe for even parity, and Po for odd parity:

$$S_D = P_e (C_1, C_2, C_3, C_4, C_5, C_6) + P_0 (D_0, \ldots, D_7) + P_o (D_8, \ldots, D_{15})$$

However, Syndrome Bit 5 is found from:

$$S_5 = P_o (D_8, \ldots, D_{15,} C_5)$$

To keep part count down, redefine:

$$S_D = S_5 + P_0 (D_0, \ldots, D_7) + P_o (C_1, C_2, C_3, C_4, C_D)$$

Then use a multiplexer to generate S_D (see *Figure 11-14*).

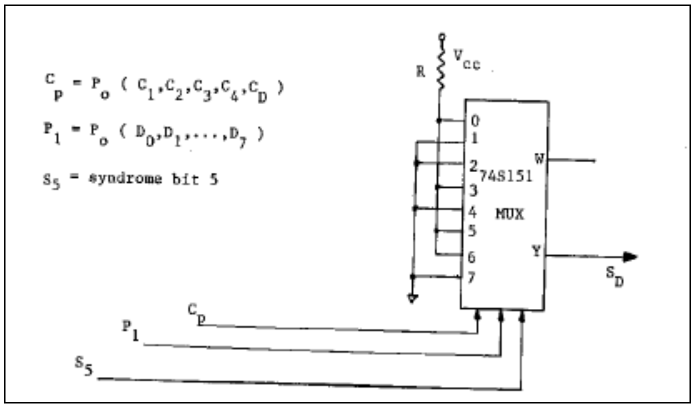

$C_p = P_o (C_1, C_2, C_3, C_4, C_D)$
$P_1 = P_o (D_0, D_1, \ldots, D_7)$
S_5 = syndrome bit 5
R
V_{cc}
0
1
2
3
4
5
6
7
74S151
MUX
W
Y
S_D
C_p
P_1
S_5

Figure 11-14 Syndrome Bit Generation Using a Multiplexer

Given the syndrome bits, a decode operation is necessary to determine error status as:

1) No error has occurred
2) One, and, therefore, a correctable error has occurred
3) Or, more than one error has occurred (uncorrectable)

The decode matrix is specified via a Marquand Map in *Figure 11-15*.

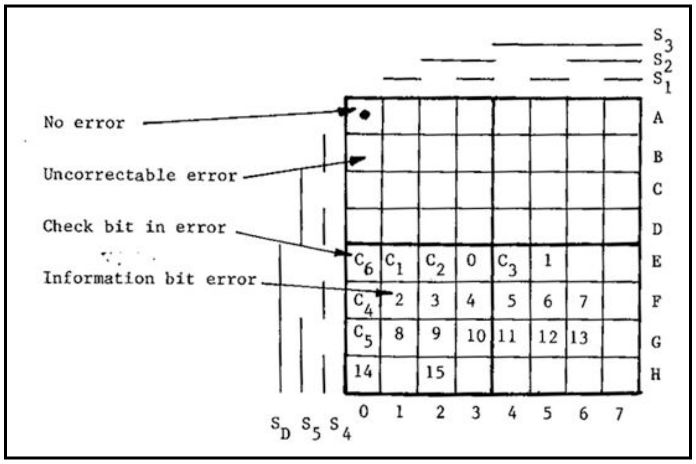


Figure 11-15 Syndrome Bit Decode

The status signals may be generated using two 1-of-16 multiplexers (Figure 11-15). Demultiplexers can decode the individual bit to be corrected and correction accomplished by inverters and XOR gates.

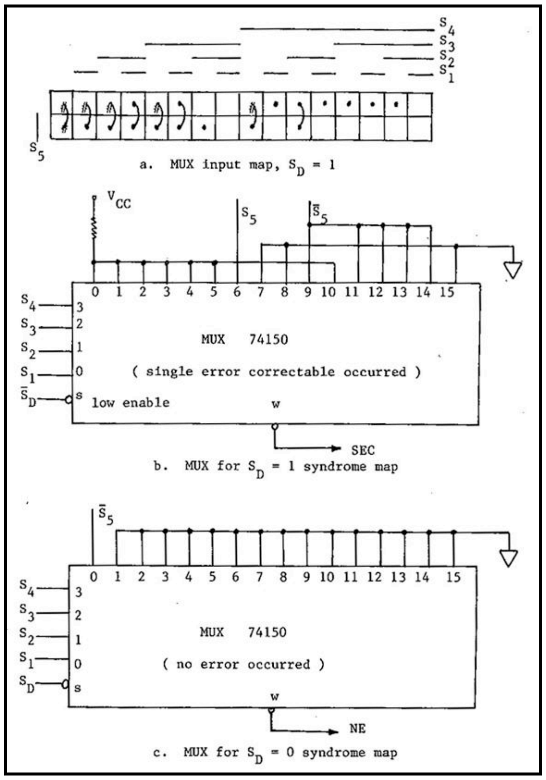


Figure 11-16 Control Signal Generation

The syndrome bit decode could also be implemented using a single PAL. Here, Marquand Maps can provide the necessary documentation and can be used as a tool for minimization. A PAL such as the PAL16L8 can be used to replace the two 1-of-16 MUX chips and an AND gate. The coding for the PAL16L8 is shown in *Figure 11-17*.

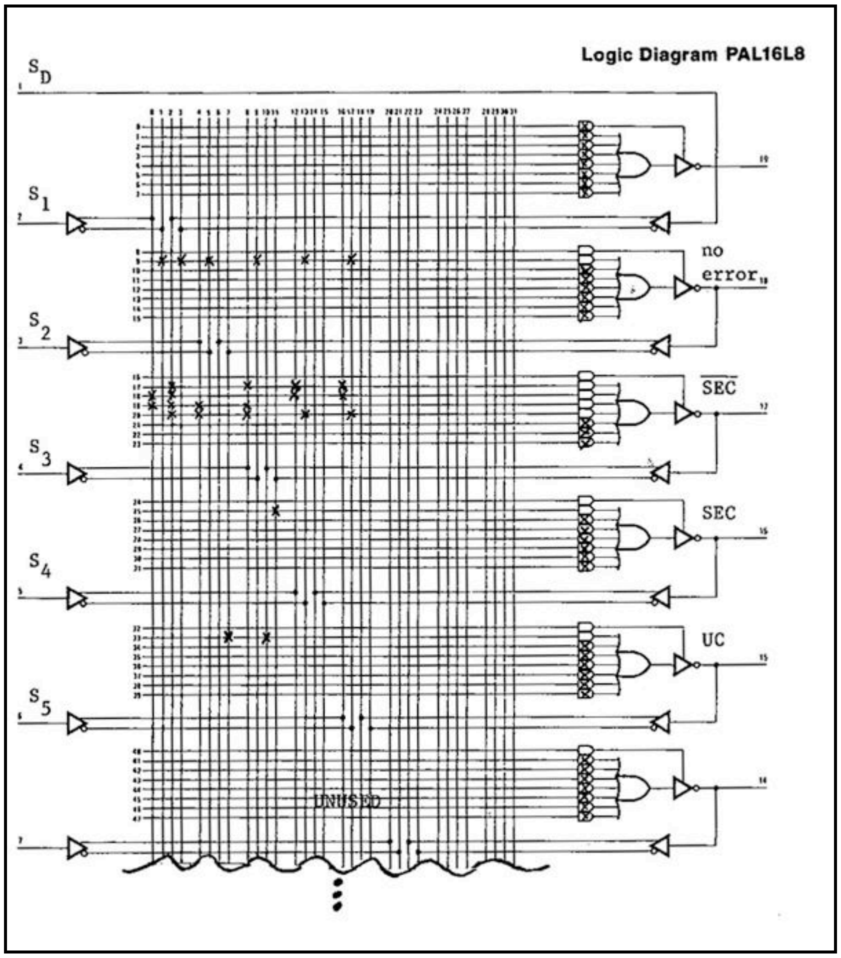


Figure 11-17 Programmed PAL for Control Signal Generation

11.5.5 Read-Only Memories

Read-Only memories or ROMs (*Figure 11-18*) contain a bit position for every combination of input variable validities for each of its multiple outputs. A ROM may be thought of as containing the information from η Marquand Maps, where η is the number of outputs. A ROM is characterized by:

1) Its programmability (field, PROM, or factory, ROM)
2) Its re-programmability (erasable, EPROM, or not)
3) Whether it has output latches or registers (registered-PROM)
4) Its size, expressed as the number of addresses (2^n, where n is the number of inputs)
5) The number of outputs
6) Whether it is all “1”s or all “0”s initially
7) Its speed (access time if unregistered; set-up time and clk to output time if registered)
8) Its output (Open-Collector or tri-state)
9) Its enable structure

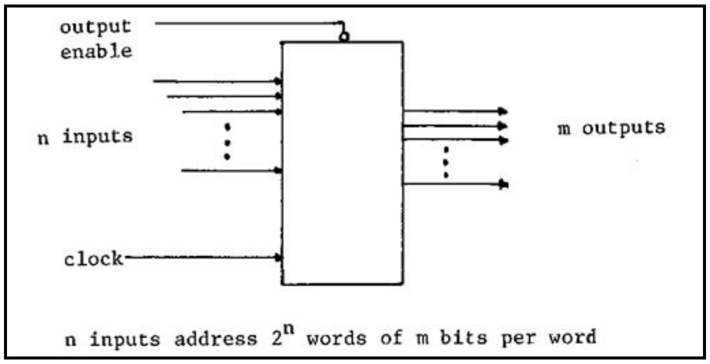
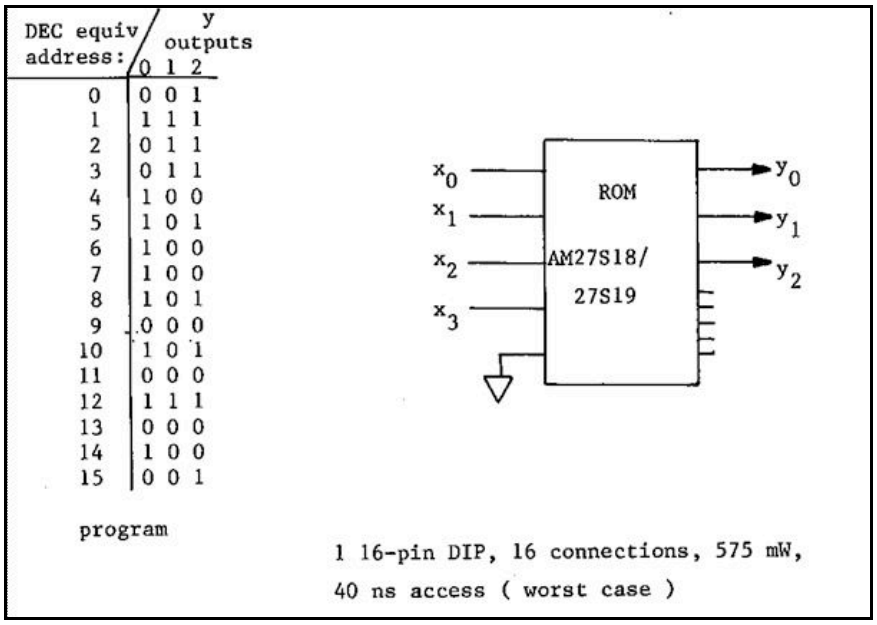


Figure 11-18 ROM Logic Block

ROM family elements (EPROM, PROM, etc.) are useful, for example, as decoders and as sequential controllers. Using a ROM represents a shift from high-speed, custom, hardwired SSI logic to possibly slower, flexible, LSI firmware design.

A ROM plus a sequencer, such as the Am2910, represents a structured approach to sequential design. It may be said the microprogramming is to hardware design what structured programming is to software design. There are parallel trade-offs which must be considered in each case.

Unlike the PGA, PAL, PLA and PMUX devices, a ROM can be erased (EPROM). Both PALs and ROMs have development system assemblers available to assist in their programming. A ROM is documented by the assembly program which prepares its tape or by a manually-prepared mnemonic program listing. [11] Where the circuit is sensitive to glitches in signal lines, registered PROMs or external latches are preferred to "naked" ROMs.



DEC equiv address:	y outputs 0	1	2
0	0	0	1
1	1	1	1
2	0	1	1
3	0	1	1
4	1	0	0
5	1	0	1
6	1	0	0
7	1	0	0
8	1	0	1
9	0	0	0
10	1	0	1
11	0	0	0
12	1	1	1
13	0	0	0
14	1	0	0
15	0	0	1

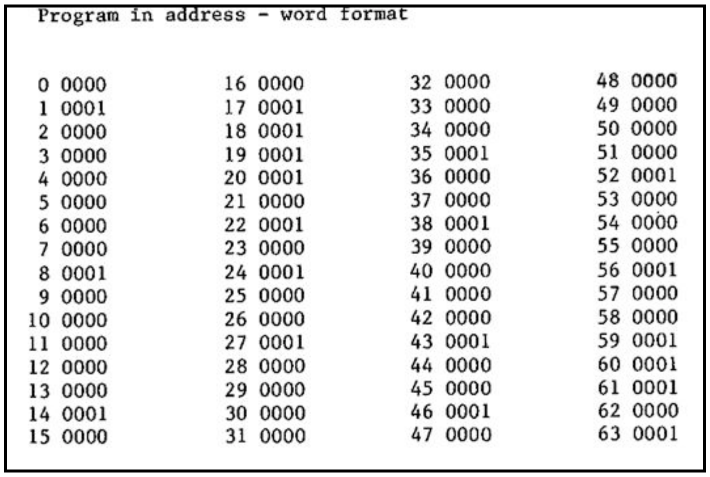
program

1 16-pin DIP, 16 connections, 575 mW,
40 ns access (worst case)

Figure 11-19 Multiple Output Problem Implemented in ROM

Figure 11-19 demonstrates the multiple-output problem implemented in a 32 x 8 bipolar PROM. Svoboda's six-variable example is given in *Figure 11-20* using a 256 x 4 bipolar PROM. In any design, the fact that there are unused areas in the PROM is not a problem. Unused areas allow for minor modifications with a major redesign. They also allow for patches to be made to the "microprogram".

[11] Designing with the bit-slice architecture (Am2900 family) and microprogramming in general are not covered in this book.



```
Program in address - word format

 0 0000        16 0000        32 0000        48 0000
 1 0001        17 0001        33 0000        49 0000
 2 0000        18 0001        34 0000        50 0000
 3 0000        19 0001        35 0001        51 0000
 4 0000        20 0001        36 0000        52 0001
 5 0000        21 0000        37 0000        53 0000
 6 0000        22 0001        38 0001        54 0000
 7 0000        23 0000        39 0000        55 0000
 8 0001        24 0001        40 0000        56 0001
 9 0000        25 0000        41 0000        57 0000
10 0000        26 0000        42 0000        58 0000
11 0000        27 0001        43 0001        59 0001
12 0000        28 0000        44 0000        60 0001
13 0000        29 0000        45 0000        61 0001
14 0001        30 0000        46 0001        62 0000
15 0000        31 0000        47 0000        63 0001
```

Since this PROM is all "0"s before programming, all Don't' Cares (#) are taken as "0".

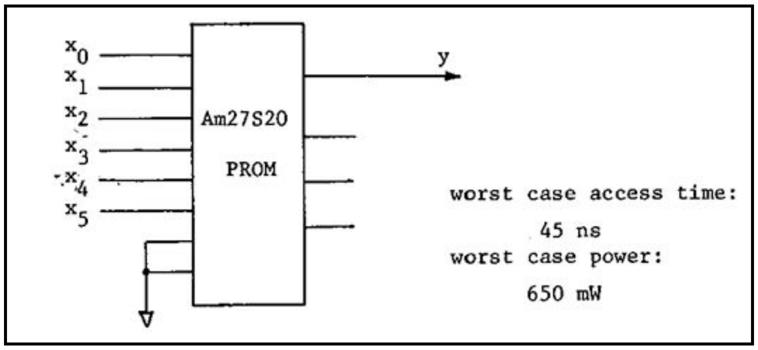


Figure 11-20 Svoboda's Six-Variable Example Done with a PROM

11.5.6 Sequential Design Example

To demonstrate microprogramming as a sequential design tool, a traffic light controller is included here. Note that this is NOT a valid application for the Am2910, but it is sufficient to demonstrate the use of this device in addition to being a problem of manageable size for discussion.

A sequencing schema is given in *Figure 11-21*. The starting point is main-street-green (MG). During main-street-yellow (MY), a test is made to determine if a protected left turn (MLT) is desired. If it is, main-left-turn-green and main0-left-turn-yellow are cycled.

Next, the side street-turns-green (SG) for its time period. During the side-street-yellow (SY), tests are made for:

1) Manual override, where the lights will all flash RED (accident control).
2) For night, where the lights will all flash RED (STOP), except
3) For night, for the main street lights, which will all flash YELLOW (CAUTION).
4) For night, for the side-protected left turn, in this priority.

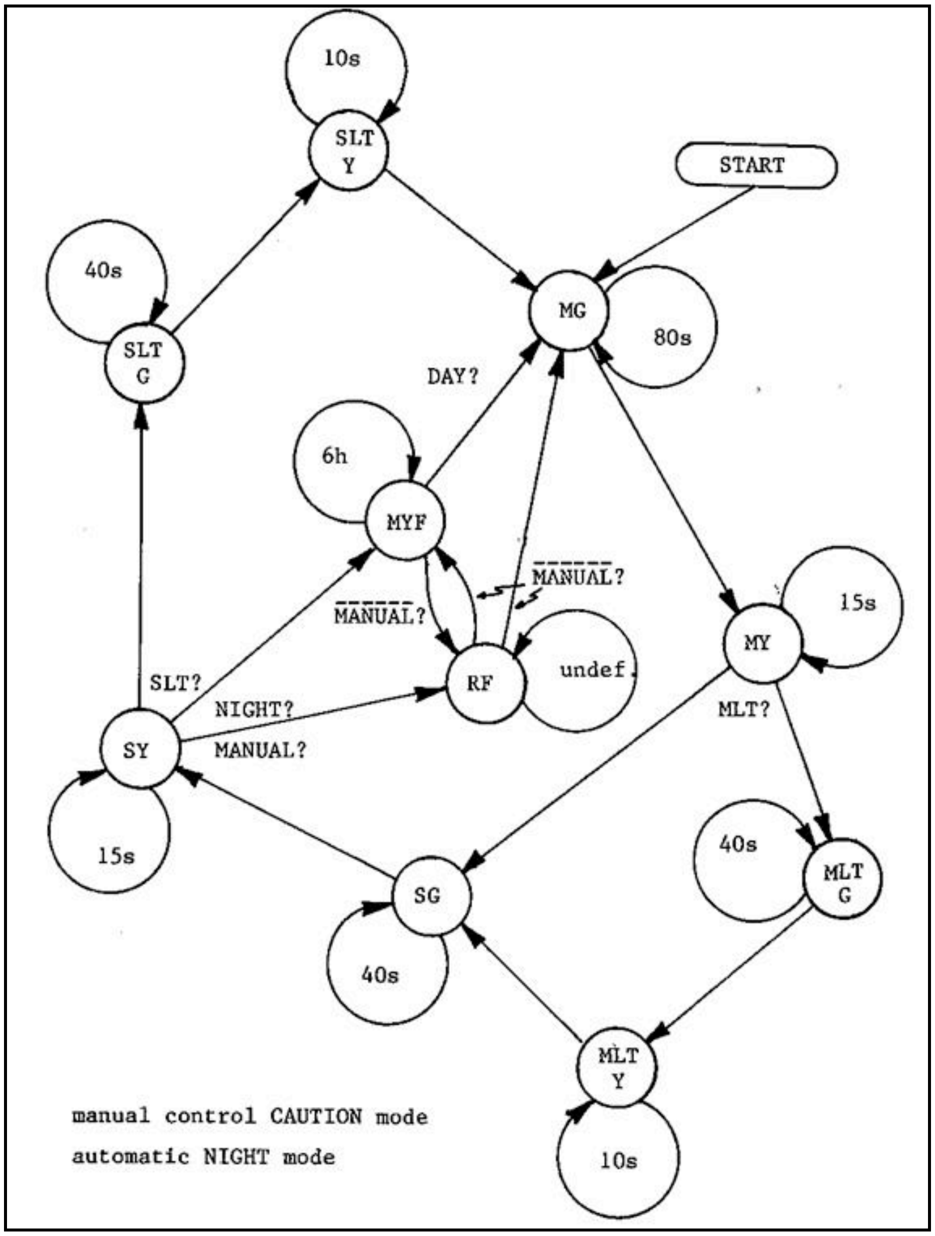


Figure 11-21 Traffic Light Controller[12]

If MANUAL is present, set all lights RED and loop here until $\underline{\text{MANUAL}}$ occurs.

If $\underline{\text{MANUAL}}$ occurs, return to normal cycle (or go to night mode if NIGHT is present).

If NIGHT is present, set the lights and loop until DAY (= $\underline{\text{NIGHT}}$) or MANUAL occurs.

The times are in seconds and a 5-second clock is assumed.

The lights are controlled by three signal lines, each, to provide signals RED, YELLOW, GREEN, RED FLASH and YELLOW FLASH.

[12] Used in the ED2000A Seminar – Bit-Slice Design: Controllers and ALUs; and in the subsequent textbook.

The state sequenced paths for the light controller are given in *Figure 11-22* and are considered a fixed constraint for the purpose of the problem. (That may seem arbitrary, but remember that this is for demonstration only.)

Name	Actual Pattern	(Main lights)
R0	000	
R2	010	red
R4	100	
Y	010	yellow
G	011	green
YF	101	yellow flash
RF	110	red flash

Name	Actual Pattern	(Side lights)
Ro	000	
R2	010	red
R4	100	
Y	010	yellow
G	011	green
RF	101	red flash

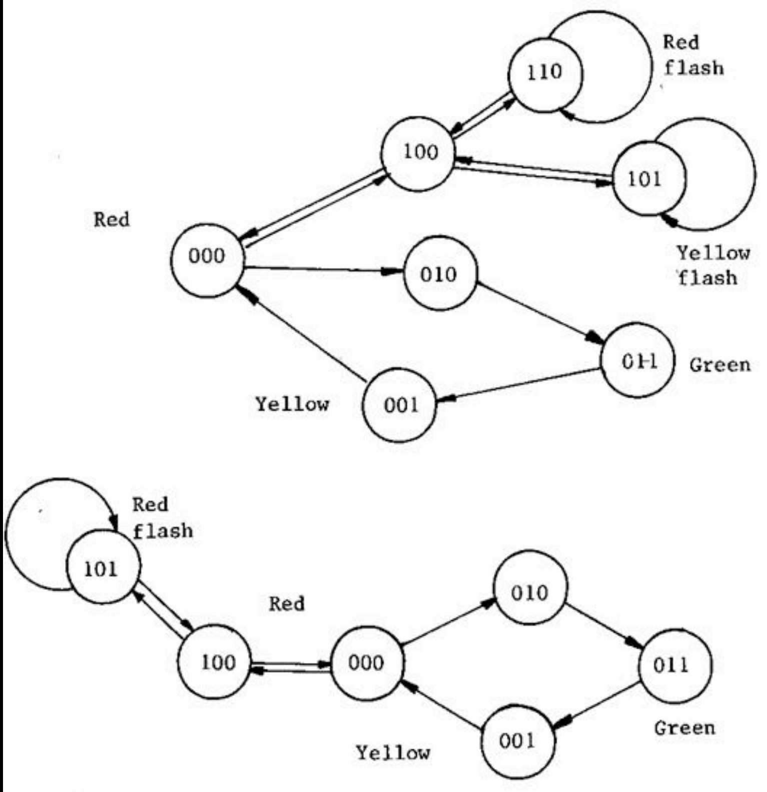


Figure 11-22 Light Control Signal State Sequencing

The control portion of the system is shown in *Figure 11-23* and uses one Am2910 sequencer, three 32 x 8 PROMs (registers not necessary here; the pipeline is shown only to demonstrate output enable), and one MUX.

MLT, <u>SLT</u>, NIGHT, MANUAL and START are assumed to be available.

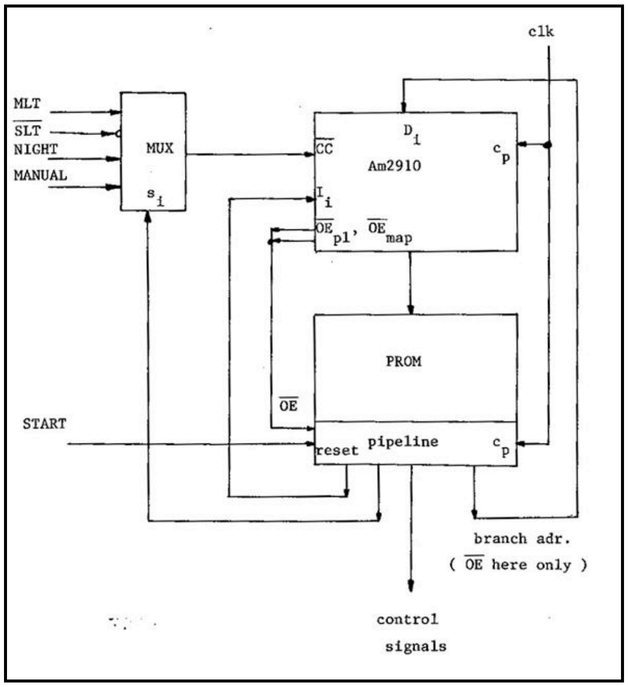


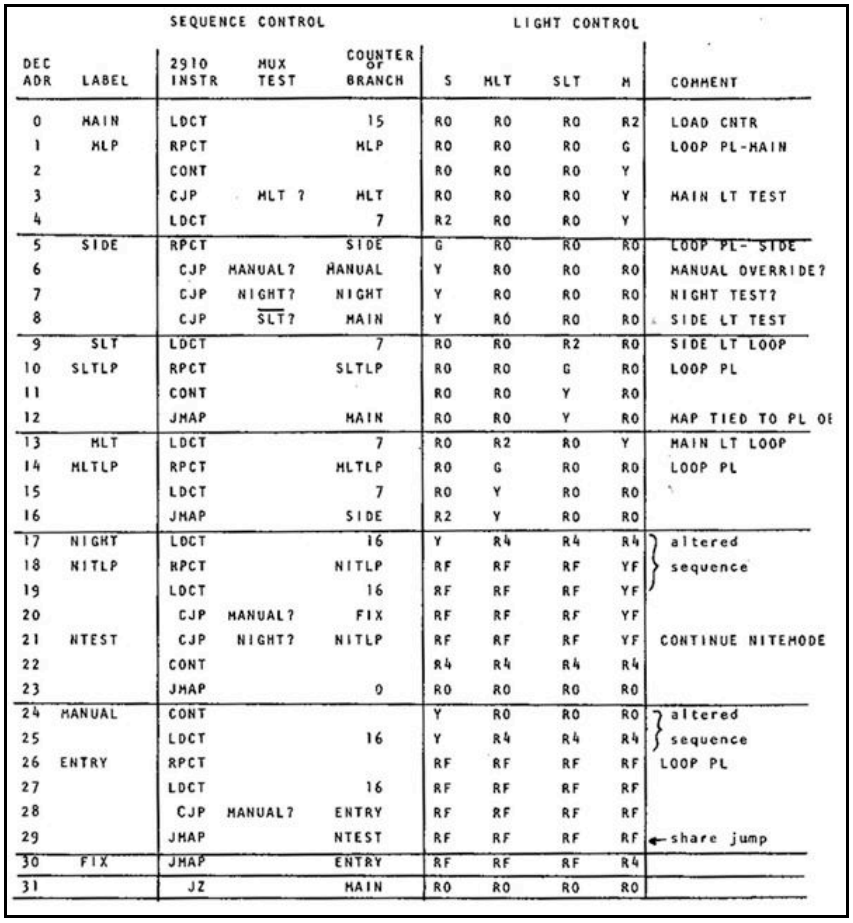
Figure 11-23 Traffic Light Control Design with the Am2910

A microprogram for the controller is given in *Figure 11-24* using mnemonics rather than bit-patterns. (A ***System 29*** development system can be used to generate the bit pattern for the PROMs.)[13]

The Am2910 has 16 instructions which facilitate programming. A summary of those used are given in *Table 11-7*

A direct similarity can be drawn to FORTRAN-type or assembly-level languages.

[13] See www.Donnamaie.com and look for Am2900 vintage publishing for information on the System 29.



		SEQUENCE CONTROL			LIGHT CONTROL				
DEC ADR	LABEL	2910 INSTR	MUX TEST	COUNTER or BRANCH	S	MLT	SLT	M	COMMENT
0	MAIN	LDCT		15	RO	RO	RO	R2	LOAD CNTR
1	MLP	RPCT		MLP	RO	RO	RO	G	LOOP PL-MAIN
2		CONT			RO	RO	RO	Y	
3		CJP	MLT ?	MLT	RO	RO	RO	Y	MAIN LT TEST
4		LDCT		7	R2	RO	RO	Y	
5	SIDE	RPCT		SIDE	G	RO	RO	RO	LOOP PL- SIDE
6		CJP	MANUAL?	MANUAL	Y	RO	RO	RO	MANUAL OVERRIDE?
7		CJP	NIGHT?	NIGHT	Y	RO	RO	RO	NIGHT TEST?
8		CJP	$\overline{SLT}$?	MAIN	Y	RO	RO	RO	SIDE LT TEST
9	SLT	LDCT		7	RO	RO	R2	RO	SIDE LT LOOP
10	SLTLP	RPCT		SLTLP	RO	RO	G	RO	LOOP PL
11		CONT			RO	RO	Y	RO	
12		JMAP		MAIN	RO	RO	Y	RO	MAP TIED TO PL OE
13	MLT	LDCT		7	RO	R2	RO	Y	MAIN LT LOOP
14	MLTLP	RPCT		MLTLP	RO	G	RO	RO	LOOP PL
15		LDCT		7	RO	Y	RO	RO	
16		JMAP		SIDE	R2	Y	RO	RO	
17	NIGHT	LDCT		16	Y	R4	R4	R4	altered
18	NITLP	RPCT		NITLP	RF	RF	RF	YF	sequence
19		LDCT		16	RF	RF	RF	YF	
20		CJP	MANUAL?	FIX	RF	RF	RF	YF	
21	NTEST	CJP	NIGHT?	NITLP	RF	RF	RF	YF	CONTINUE NITEMODE
22		CONT			R4	R4	R4	R4	
23		JMAP		0	RO	RO	RO	RO	
24	MANUAL	CONT			Y	RO	RO	RO	altered
25		LDCT		16	Y	R4	R4	R4	sequence
26	ENTRY	RPCT			RF	RF	RF	RF	LOOP PL
27		LDCT		16	RF	RF	RF	RF	
28		CJP	MANUAL?	ENTRY	RF	RF	RF	RF	
29		JMAP		NTEST	RF	RF	RF	RF	← share jump
30	FIX	JMAP		ENTRY	RF	RF	RF	R4	
31		JZ		MAIN	RO	RO	RO	RO	

Figure 11-24 Microprogram for the Controller[14]

Table 11-7 Table of Am2910 Instructions Used in the Sample Design

Instruction	Function
CONT	Continue, address = address + 1
LDCT	Load counter and Continue
RPCT	Repeat starting at given address until counter = 0 (Do-loop)
JMAP	GO TO branch address
CJP	Conditional jump (IF – THEN)
JZ	Initialize jump zero

[14] The use of a Meta-Assembler was the means to program the Am2900 Family – these days, support assemblers and compilers are generated with C++. There is, however, still a table of bit-level values that must be equated to the defined mnemonics of the higher-level language.