Chapter 3: Data and Control Flow in Systematic Structures

Sections:

Notation · · · Two Phase Clocks · · · The Shift Register · · · Relating Different Levels of Abstraction · · · Implementing Dynamic Registers · · · Designing a Subsystem · · · Register to Register Transfer · · · Combinational Logic · · · The Programmable Logic Array · · · Finite State Machines · · · Towards a Structured Design Methodology

The process of designing a large-scale integrated system is sufficiently complex that only by adopting some type of regular, structured design methodology can one have hope that the resulting system will function correctly, and not require a large number of redesign iterations. However, the methodology used should allow the designer to take full advantage of the architectural possibilities offered by the underlying technology.

In this chapter we present a number of examples of data and control flow in regularized structures, and discuss the way in which these structures can be assembled into larger groups to form subsystems, and then these subsystems assembled to form the overall system. The design methodology suggested in this chapter is but one of many ways in which integrated system design may be structured. The particular circuit form presented does tend to produce systems of very simple and regular interconnection topology, and thus tends to minimize the areas required to implement system functions. Arrays of pass transistor logic in register to register transfer paths are used wherever possible to implement system functions. This approach tends to minimize power dissipated per unit area, and, with level restoration at appropriate intervals, tends to minimize the time delay per function. The methodology developed is applied in later chapters to the architecture and design of a data processing path and its controller, which together form a microprogrammed digital computer.

Computer architects, who usually design systems in a rather structured way using commercially available MSI and LSI circuit modules, are often surprised to discover how unstructured is the design within those modules. In principle one can use the basic NAND and NOR logic gates described in Chapter 1 to implement combinational logic, build latches from these gates to implement data storage registers, and then proceed to design integrated systems using traditional logic design methodology as applied to discrete devices. Integrated systems are often designed this way at the present time. However, it is unlikely that such unstructured approaches to system design can survive as the technology scales down towards maximum density VLSI.

There are historical reasons for the extensive use of random logic within integrated systems. The first microprocessors produced by the semiconductor industry were fairly direct mappings of early generation central processor architectures into LSI. A block diagram of the Intel 4004, the earliest microprocessor to see widespread commercial application, is illustrated in figure 1a. The actual chip layout of the 4004 shown in Figure 1b indicates the complexity of the LSI implementation of this simple central processing unit. Such LSI systems, directly mapping data paths and control functions appropriate in earlier component technologies, of necessity contained a great deal of random logic. However, the extensive use of random logic results in chip designs of very great geometrical and topological complexity, relative to their logical processing power.

To deal with such complexity, system design groups have often stratified the design problem into architecture, logic design, circuit design, and finally circuit layout, with specialists performing each of these levels of the design. Such stratification often precludes important simplifications in the realization of system functions.

Switching theory provides formal methods for minimizing the number of gates required to implement logic functions. Unfortunately, such methods are of little value in VLSI systems, since the area occupied on the silicon surface by circuitry is far more a function of the topological properties of the circuit interconnections than it is of the number of logic gates implemented. The minimum gate implementation of a function often requires much more surface area for its layout than does an alternative design using more transistors but having simpler interconnection topology.

There are known ways of structuring integrated circuit designs implemented using traditional logic design methods. A notable example is the *polycell*[1] technique. In this technique, a group of standard cells corresponding to typical SSI or MSI functions are gathered into a library of functions. The logic diagram for the system to be implemented is used to specify which cells in the library are required. The cells are then placed into a chip layout, and interconnections laid out between them by an automatic interconection routing system. The polycell technique provides the logic designer having limited knowledge of integrated systems with a means of implementing modest integrated circuit designs directly from logic equations. However, a heavy penalty is paid in area, power, and delay time. Such techniques, while valuable expedients, do not take advantage of the true architectural potential of the technology, and do not provide insight into directions for further progress.

The Intel 4004 Microprocessor: An Early LSI System

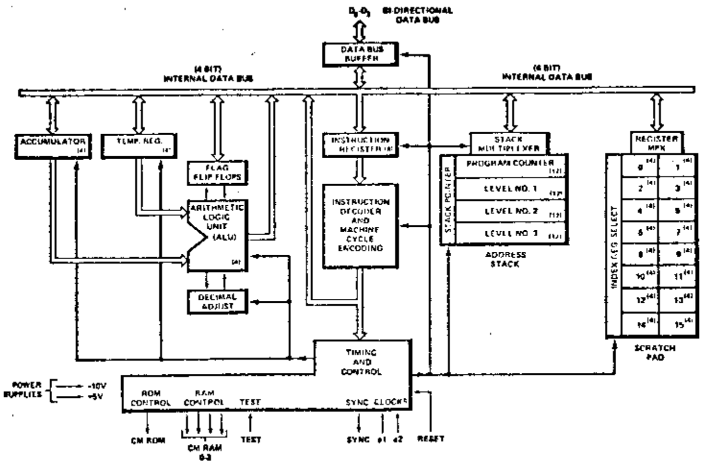


Fig. 1a. 4004 Block Diagram

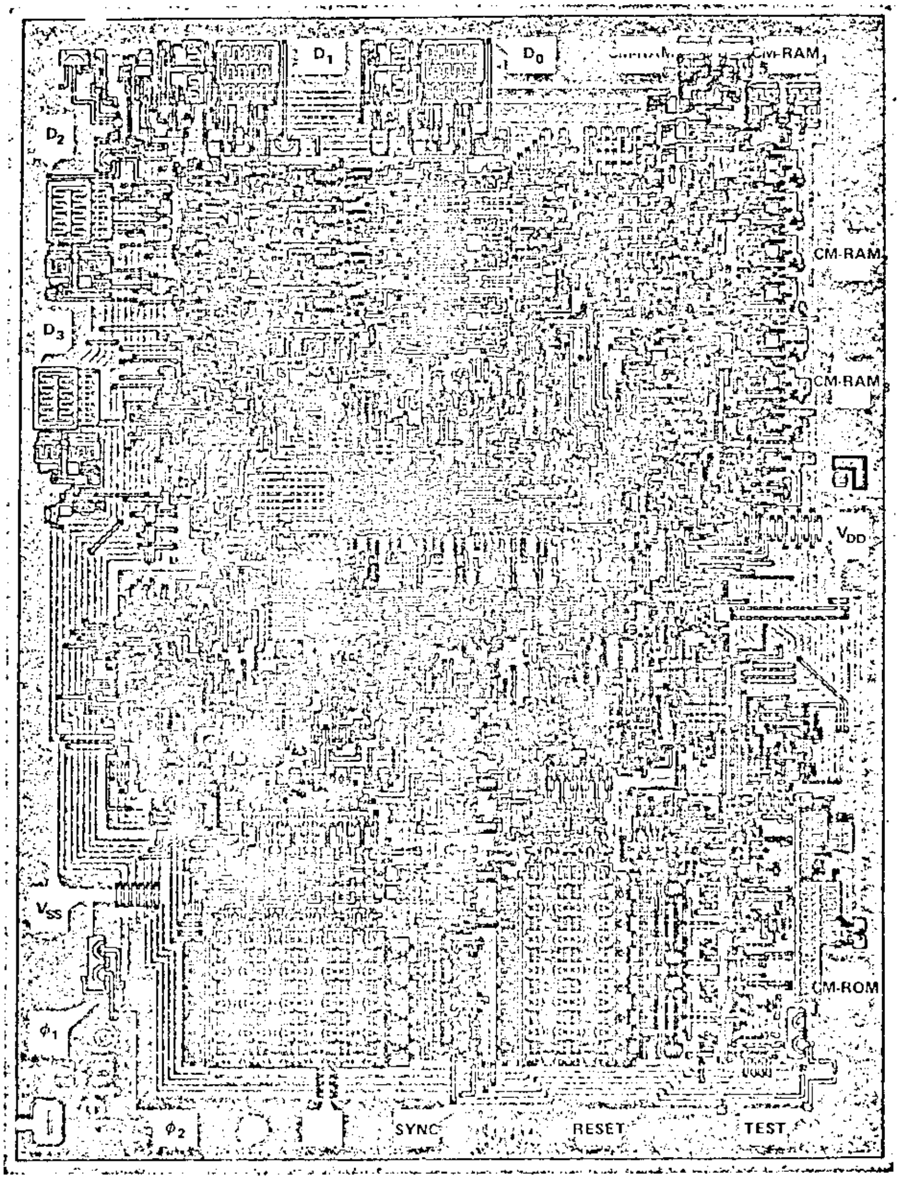


Fig. 1b. 4004 Chip Photomicrograph with Pin Designations

[reprinted with permission of G. Moore, Intel Corp.]

Switching theory not only yields the minimum number of gates to implement a logic function, but it also directly synthesizes the logic circuit design. Unfortunately, at the present time there is no general theory which provides us with a lower bound on area, power, and delay time for the implementation of logic functions in integrated systems. Theoretical lower bounds for certain special structures and algorithms of interest are given in chapter 9.

In the absence of a formal theory, we can at best develop and illustrate alternative design methodologies which tend to minimize these physical parameters. Proposed design methodologies should in addition provide means of structuring system designs so as to constrain complexity as circuit density increases. We hope that the examples and techniques presented in this text will serve to clarify these issues and stimulate others to join in the search for more definitive results[2].

Notation

There are a number of different levels of symbolic representation for MOS circuits and subsystems used in this text. Figures 2a., 2b., 2c., and 2d., illustrate a NAND gate at several such levels. At times it may be necessary to show all the details of a circuit's *layout geometry* in order to make some particular point. For example, a clever variation in some detail of a circuit's layout geometry may lead to a significant compaction of the circuit's area without violating the design rules.

Often, however, a diagram of just the topology of the circuit conveys almost as much information as a detailed layout. Such *stick diagrams* may be annotated with important circuit parameters if needed, such as the L/W ratios shown in figure 2b. Many of the important architectural parameters of circuits and subsystems are a reflection of their interconnection topologies.

Alternative topologies often lead to very different layout areas after compaction. The discovery of a clever starting topology for a design usually provides far better results than does the application of brute force to the compression of final layout geometries. For this reason, many of the important structural concepts in this chapter and throughout the text will be represented for clarity by use of colored *stick diagrams.* The color coding of the stick diagrams is the same as that of layout geometries, and is as follows: *green* symbolizes *diffusion* and *transistor channel reqion*; *yellow* symbolizes *ion implantation* for depletion mode transistors; *red* symbolizes *polysilicon*; *blue* symbolizes *metal*; *black* symbolizes a *contact.*

Later, through a number of examples in chapter 4, we will present the details of procedures by which the stick diagrams are transformed into circuit layouts, and then digitized for maskmaking. Note that if this topological form of representation were formalized, one might consider "compiling" such descriptions by implementing algorithms which "flesh out and compress" the stick diagrams into the final layout geometries[3], according to the constraints imposed by the design rules.

When the details of neither geometry nor topology are needed in the representation, we may revert to the familiar *circuit diagrams* and *logic symbols.* At times we may find it convenient to *mix* several levels in one diagram, as shown in figure 2e. A commonly used mixture is: (i) stick diagrams in portions where topological properties are to be illustrated, (ii) circuit symbols for pullups, and (iii) logic symbols, or defined higher level symbols, for the remaining portions of the circuit or system.

We will define logic variables in such a way that a *high voltage* on a signal path representing that variable corresponds to that variable being *true* (logic-1). Conversely, a *low voltage* on a signal path representing that logic variable corresponds to the variable being *false* (logic-0). Here *high voltage* and *low voltage* mean well above and well below the logic threshold of any logic gates into which the signal is an input. This convention simplifies certain discussions of logic variables and the voltages on the signal paths representing them. Thus when we refer to the logic variable β being *high,* we indicate simultaneously that β is *true* (logic-1) and is represented on the signal path named β by a *high voltage,* one well above the logic threshold. In boolean equations and logic truth tables we use the common notation of 1 and 0 to represent *true* and *false* respectively, and by implication *high* and *low voltages* on corresponding signal paths.

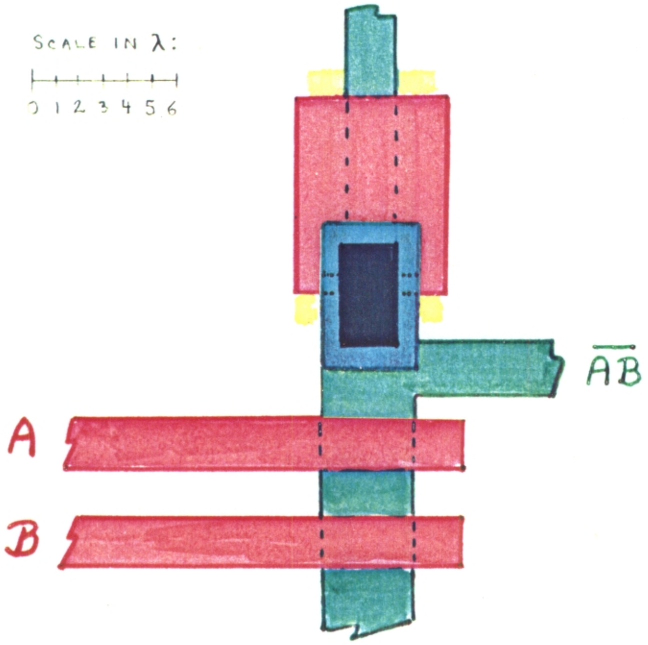


Fig.2a. NAND Gate: Layout Geometry

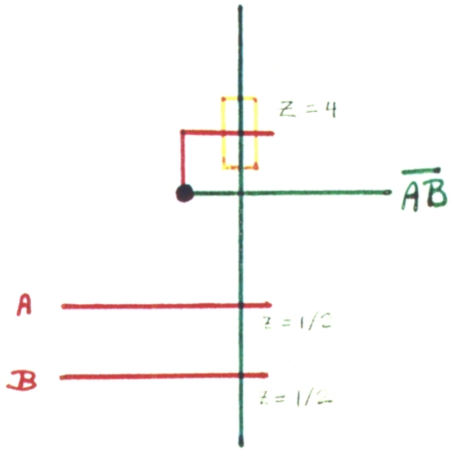


Fig.2b. NAND Gate: Topology (Stick Diagram)

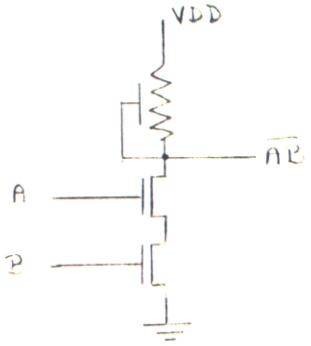


Fig.2c. NAND Gate: Circuit Diagram

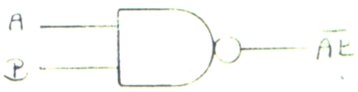


Fig.2d. NAND Gate: Logic Symbol

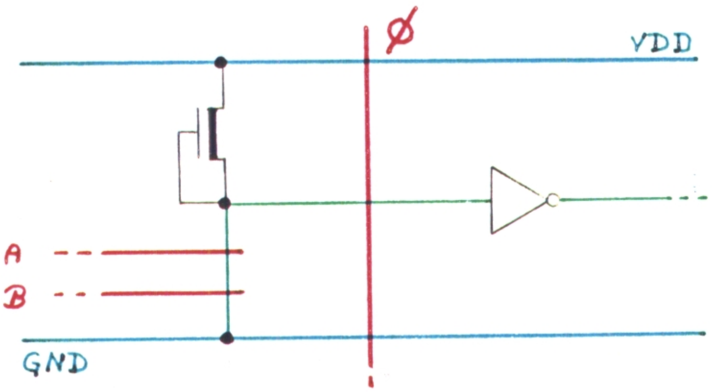


Fig. 2e. Example of Mixed Notation

Two Phase Clocks

We will often make use of a particular form of "clocking" scheme to control the movement of data through MOS circuit and subsystem structures. By clocking scheme we mean a strategy for defining the times during which data is allowed to move into and through successive processing stages in a system, and for defining the intervening times during which the stages are isolated from one another.

Many alternative clocking schemes are possible, and a variety are in current use in different integrated systems[4]. The clocking scheme used in an integrated system is closely coupled with the basic circuit and subsystem structuring, and has major architectural implications. For clarity and simplicity we have selected one clocking scheme, namely *two-phase, non-overlapping clock signals.* This scheme is used consistently throughout the text, and is well matched to the type of basic structures possible in MOS technology.

The two clock signals φ_1 and φ_2 are plotted as a function of time in figure 3. The signals both switch between zero volts (logic-0) and a voltage near VDD (logic-1), and both have the same period, T. Note that both signals are non-symmetric, and have non-overlapping *high* times. The *high* times are somewhat shorter than the *low* times. Thus φ_2 is *low* all during each of those time intervals from when φ_1 rises, nears VDD, and then falls back to zero.

We have adopted a compact convention for transistions of clocking signals. The *rising transition* of a signal φ is symbolized as $\uparrow\varphi$, and the *fall* as $\downarrow\varphi$, we also have a similar rule for φ_1, namely $\varphi_1 = 0$ all during each time interval from $\uparrow\varphi_2$ to $\downarrow\varphi_2$. Therefore, at all times the logic AND of the two signals equals zero: $[\varphi_1(t)] \cdot [\varphi_2(t)] = 0$, for all t. For convenience, we will often use the following equivalence in our descriptions: "*during* φ_i" is equivalent to "*during the time period when* φ_i *is high*". In the next section we will illustrate the use of these two clocking signals to move data through some simple MOS circuit structures. A more detailed discussion of clocking requirements is given in chapter 7.

The Shift Register

Perhaps the most basic structure for movement of a sequence of data bits is the *serial shift register,* shown in circuit diagram form in figure 4a. The shift register is composed of level restoring inverters coupled by pass transistors, with the movement of data controlled by applying clock signals φ_1 and φ_2 to the gates of alternate pass transistors in the sequence.

Data is shifted from left to right as follows. Suppose a logic signal X is present on the leftmost input to the shift register when clock signal φ_1 rises. Then, during the time when φ_1 is *high,* this signal will propagate through the pass transistor and be stored as charge on the input capacitance of the first inverter stage. For example, if the signal X is *low,* then the inverter input gate capacitance will be discharged towards zero volts during the time when φ_1 is *high.* On the other hand, if X is *high,* the inverter input capacitance will charge up towards VDD - V_{th} during φ_1.

When the clock signal φ_1 falls, the pass transistor becomes an open circuit, isolating the charge on the input of the inverter. The second clock phase is now initiated by the rise of φ_2. During the time interval when φ_2 is *high* the logic signal X, now inverted, will flow through the second pass transistor onto the gate of the second inverter. This pattern can be repeated an arbitrary number of times to produce a shift register of any length.

Note that since the clock signals do not overlap, the successive pairs of stages of the shift register are effectively isolated from one another during the transfer of data between inverter pairs. For example, when φ_1 is *low,* and φ_2 is *high,* all adjacent inverters connected by the φ_2 controlled pass transistors are in the process of transferring data from the left to the right members of the pairs. All these pairs of inverters are isolated from each other by the intervening φ_1 controlled pass transistors which are all open circuits when φ_1 is *low.*

It is also important to note that the shortest period, T, we can use for clocks controlling such data transfers is determined by the time required to adequately charge or discharge the inverter input gate capacitance through the pass transistor and the preceding stage pullup or pulldown. To this time must then be added an increment of time sufficient to insure that the clocks do not overlap. For more complex systems, the minimum clock period may be estimated as a function of basic circuit parameters as discussed in Chapter 1.

Figures 4b and 4c illustrate the serial shift register using mixed notations. In figure 4b, each inverter circuit diagram has been replaced by its logic symbol. In figure 4c, the pass transistor

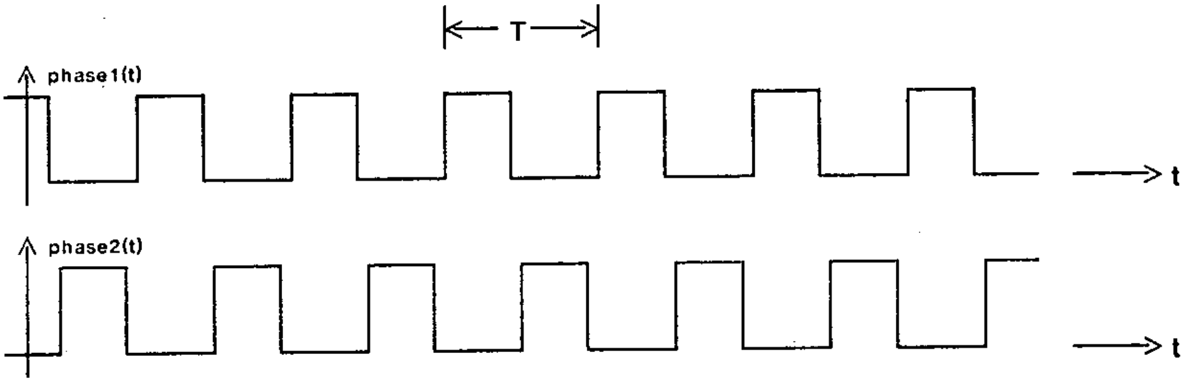


Fig. 3. Two Phase Non-Overlapping Clock Signals

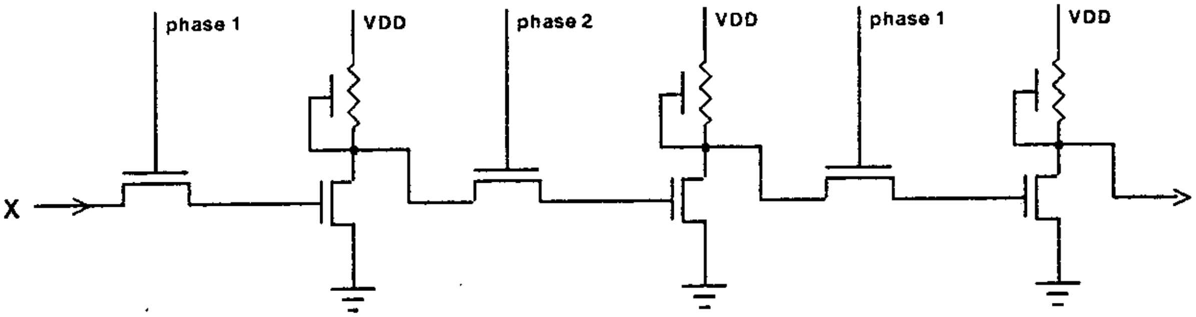


Fig. 4a Shift Register: Circuit Diagram

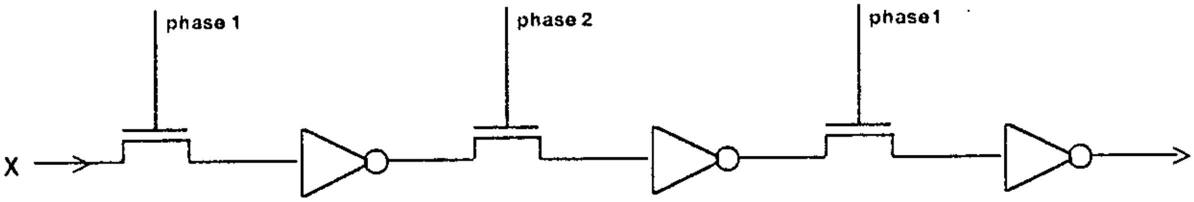


Fig. 4b. Shift Register: In Mixed Notation

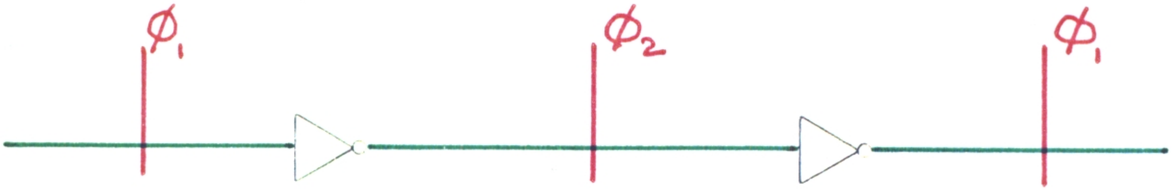


Fig.4c. Shift Register: More Mixed Notation

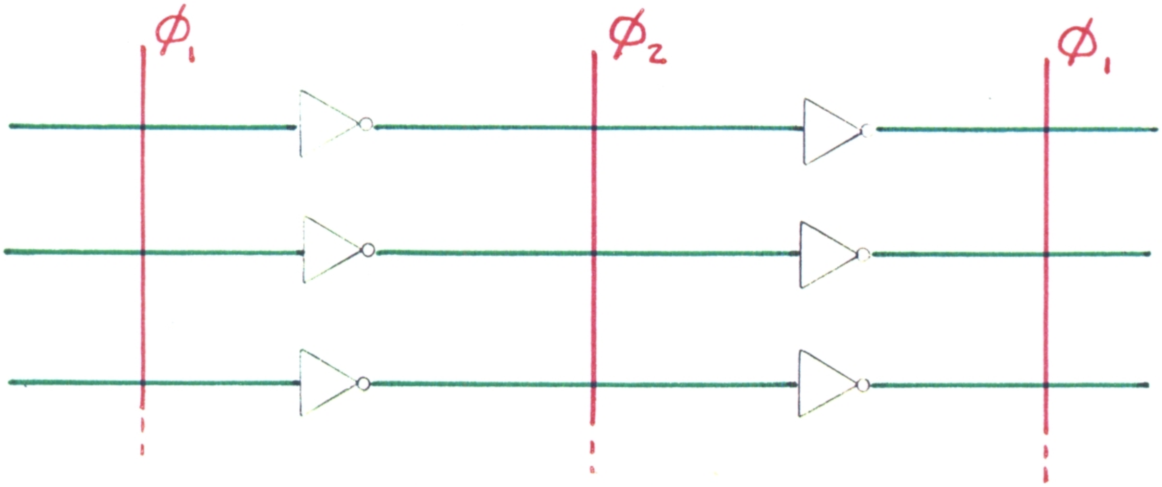


Fig.5a. Array of Shift Registers

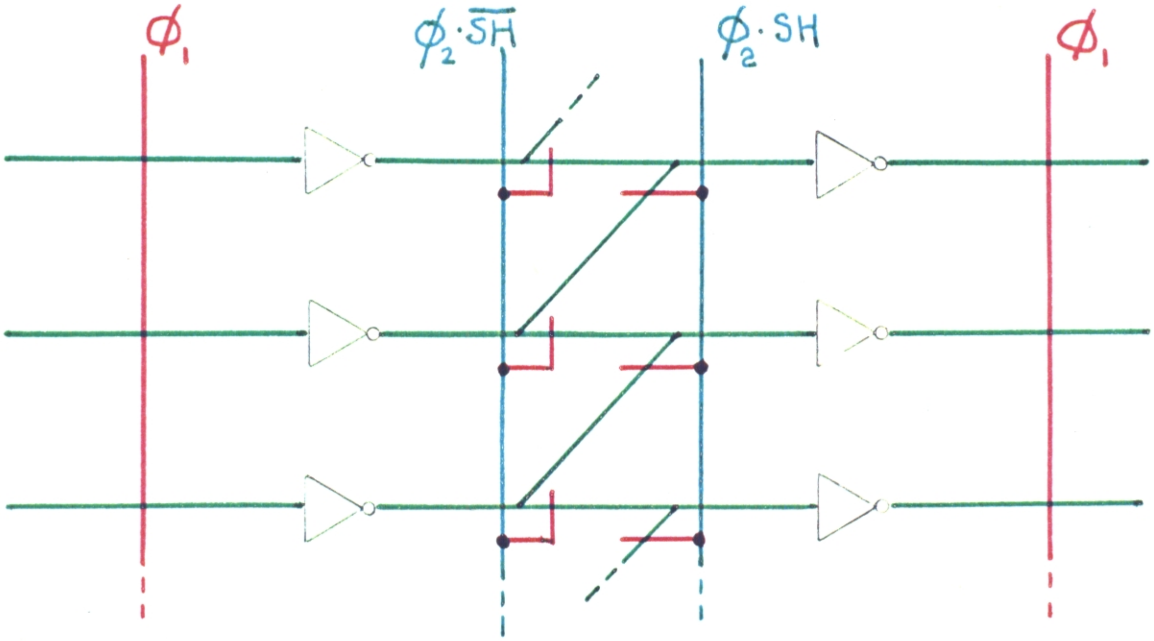


Fig.5b. Shift-Up Register Array

circuit symbols have been replaced by their stick diagrams. When visualizing the inverter, as represented by its logic symbol, in a circuit structure containing mainly stick diagrams, two points should be kept in mind:

(i) The input to the inverter leads directly to the gate, and thus the gate capacitance, of the inverter's pulldown transistor. This input may be used to store a data bit by isolating the charge representing the bit with a pass transistor. Note that the input path will end up on the poly level within the inverter. A contact cut may thus be required to connect the poly gate and the metal or diffusion path on which the signal enters the inverter.

(ii) Since the connection between the source and gate of the inverter pullup transistor requires a connection of all three conducting levels, the inverter output signal may easily be routed out on any one of the three levels.

Identical serial shift registers can be stacked next to each other and used to move a sequence of data *words*, as shown in figure 5a. The simple structure in figure 5a anticipates the elegant topological simplicity of many important MOS integrated system functions. By connecting the successive inverter stages with diffusion paths, the pass transistors controlled by the clock signals are formed by simply running vertical clock lines in poly. The structure in figure 5a also anticipates another important point: topological simplification often results when control signals flow on lines that are at right angles to the direction of data flow. In this way as many bits as necessary can be processed in parallel with the same control signals.

The example in figure 5a is so rudimentary it is perhaps difficult to visualize the two clock signals as actually containing control information. Let us consider a slightly more complex example, the *shift-up register array* shown in figure 5b. In this structure, each data bit moving from left to right during φ_2 has two alternative pass transistor paths through which it can proceed to the next stage: a straight through path, and a path which shifts it up to the next higher row. If the control signal SH is *low*, then $[\varphi_2 \bullet SH']$ is *high*, and the straight through pass transistor paths are used during φ_2. At the same time, $[\varphi_2 \bullet SH]$ is *low*, thus preventing data flow through the shift-up pass transistor paths. On the other hand, if SH is *high*, the straight through pass transistors are off and the shift-up pass transistor paths are used during φ_2, resulting in the entire data word being shifted vertically as well as horizontally. Here the vertical control lines are run in metal, and the pass transistors are selectively formed by crossing the appropriate diffusion paths with short poly lines.

Relating Different Levels of Abstraction

In the discussions in this chapter, we will not have to make extensive calculations of the detailed electrical behavior of the devices and circuits involved in order to analyze the general behavior of digital logic constructed with these devices and circuits. Most of the examples presented in this chapter, and throughout the text, build upon the use of pass transistors coupling inverting logic stages as a means of structuring designs. The general results of chapter one provide the solutions to most device and circuit problems encountered, such as ratio and delay calculations, etc. In most cases, design concepts can be worked out using stick diagrams, and only at the stage of transforming the circuit topology into the detailed circuit layout geometry will these calculations need to be worked out, either by hand or with circuit simulation programs.

It is important to simplify our mental model of integrated circuitry, so as to more quickly and easily analyze or explain the function of a given circuit, and more easily visualize and invent new circuit structures without drifting too far away from physically realizable and workable solutions. Of course, it is a dangerous practice to oversimplify our abstractions of electronic circuit behavior, and there are some nMOS circuits of deceptively simple appearance which have exceedingly complex behavior. However, throughout large portions of digital integrated systems, if the circuit and subsystem design is structured as suggested in this text, an extremely simple mental model of device and circuit behavior will prove adequate to predict circuit and subsystem behavior.

Figure 6a illustrates a simple way of visualizing the operation of successive inverting logic stages coupled by pass transistors. Assume for the moment that any pass transistors in the paths between stages are *on*. To visualize the time behavior of an inverter, and the effect of the pullup L/W to pulldown L/W ratio, imagine the flow of current from VDD to GND as the flow of a fluid, and the inverter's two transistors as valves. The basis for thinking of the transistors in this manner is the fluid model of their internal behavior, as given in chapter 1. Whether a transistor is on or off depends upon the voltage, and thus upon the charge, on its control gate, and also on its threshold voltage. The upper "valve" is always open, since the pullup transistor is always *on*. However, the "valve" corresponding to the pulldown transistor may be either open or closed, depending on the amount of charge on its gate.

In figure 6a, the input to inverter-A is a logic-0, so the pulldown of inverter-A is *off*, and the lower valve is closed. Current is thus diverted to the large charge storage site corresponding to the gate of the pulldown of inverter-B. At this level of diagram we have reverted to the common

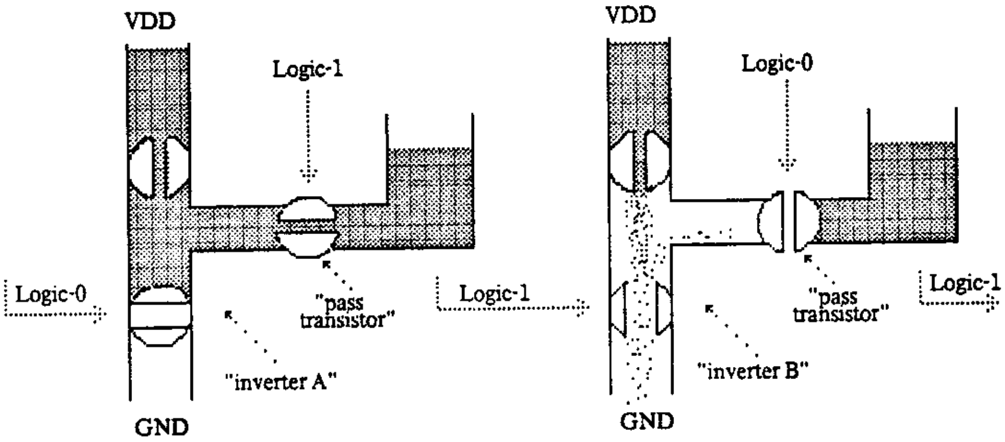


Fig. 6a. A Way of Visualizing the Operation of Successive Inverter Stages

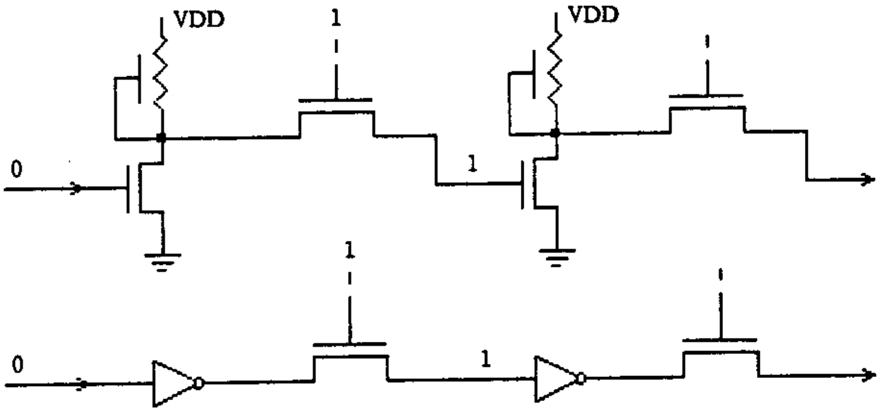
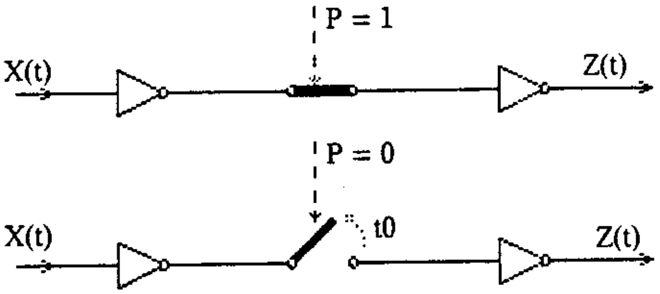


Fig. 6b. Successive Inverter Stages: Circuit Diagram and Logic Diagram



While P = 1,
Z(t) = X(t - delta t)

If P → 0, at t = t0
Z(t) = Z(t0), for t > t0

Fig. 6c. Successive Inverter Stages Connected Through a Pass Transistor

[Illustrating effect of the pass transistor "switch"] convention of positive charge flow from VDD to ground, rather than electron flow from ground to VDD. If sufficient positive charge has flowed onto this gate, corresponding to a high level of fluid in the tank representing the gate capacitance, then the pulldown of inverter-B is turned *on*, and thus the lower valve of inverter-B is open. If the lower valve in inverter-B is much larger than the upper one, corresponding to a practical pullup to pulldown size ratio, then the pulldown of inverter-B can sink all the source current provided by the pullup. Also, if given sufficient time and if the connecting pass transistor is on, the pulldown can drain off any charge stored on the succeeding inverter's input gate. Thus we can visualize the sequence of inversions of a logic signal propagating through successive inverter stages as an alternation between high and low levels of fluid in the storage tanks. We can also visualize some of the time behavior of the signal propagation: the larger the gate capacitance, the longer it takes to build up enough charge to open the next stage, and the longer it takes to drain charge off the next stage to turn it off.

Figure 6b represents the same physical circuit modelled in figure 6a, but on successively higher levels of abstraction. When analyzing circuit or logic diagrams showing successive inverting logic stages, as in figure 6b, one should keep the model of figure 6a in mind. Whether one is a novice or an expert in integrated system design, it is very helpful to compress the details of any given lower level of abstraction, so as to reduce the complexity of the problems presented at the next higher level, and enable the mind to span problems of larger scope.

We are now able to visualize a very simple model for the pass transistor: it is in fact like a valve, or "switch" in the path between an inverter and the next charge storage site, i.e. the input gate of the next inverter. Figure 6c shows two inverters coupled by a pass transistor, with the pass transistor informally symbolized as a "switch". In the upper diagram of figure 6c, the pass transistor input is a logic-1, and so the "switch" is in the *on* position, resulting in the output Z being equal to the input X, after a suitable delay time Δt. Thus during the time the pass transistor gate input $P = 1$, the output $Z(t) = X(t - \Delta t)$. Here Δt is some multiple of the transit time, τ, of the inverter pulldown transistor, as discussed in Chapter 1.

In the lower diagram of figure 6c, the pass transistor "switch" is moved to the *off* position since P is a logic-0. Therefore, according to our model, the valve in the path between the inverters is shut, and the charge, or lack of charge, is isolated in the storage site. Thus, once the pass transistor "valve" is shut, Z remains at a constant value, independent of changes in X. In other words, if $P \rightarrow 0$, at $t = t_0$, then $Z(t) = Z(t_0)$, for $t > t_0$.

These simple visualizations of the inverter and the pass transistor will carry us fairly far into LSI subsystem design. Several logic circuits in this chapter are drawn first in stick diagram form, and then informally sketched with pass transistors replaced with "switches", both to clarify the behavior of the circuits involved, and to further demonstrate the applicability of the model.

Implementing Dynamic Registers

Registers for the storage of data play a key role in digital system design. It is interesting to note that a group of adjacent inverters, with their gates isolatable by pass transistors, can be considered a form of temporary storage register. This arrangement is illustrated in figure 7, which shows two levels of symbolism for this *dynamic register*. Such a register is very simple in structure. It consists of only three transistors per bit position: the pass transistor and the two transistors of the inverter. However, this dynamic form of register will preserve data only as long as charge can be retained on the inverter input gates. Typically dynamic registers are used in situations where the input gate updating control signals are applied frequently. They are ideal in a clocked system in which they are reloaded every clock cycle, as in the shift register.

Suppose we wish to construct a simple register which can be loaded during the appropriate clock phase under the control of a *load* signal, and which will retain its information through an indefinite number of successive clock periods until it is reloaded using the *load* signal. A one bit cell for such a register may be constructed using cross coupled inverters in the configuration shown in figure 8. This register cell is still dynamic in form, since it uses charge storage on the gate of the first inverter to preserve its state. However, it need not be loaded on every successive φ_1 as was the simple register in figure 7. The pass transistor leading to it from the preceding stage is switched on only when *both* φ_1 and LD are *high*. On any following φ_1 when LD is *low*, the cell updates itself by the feedback path through the second pass transistor. Figure 9 illustrates a selectively loadable register composed of such cells. One important feature of this type of register is that it provides as output both the true and complemented forms of the stored data. This feature is often useful when the data is to be processed by a following network of combinational logic.

While there are more elaborate forms of dynamic and static registers, the above two forms are sufficient for many of the required data storage applications within integrated systems.

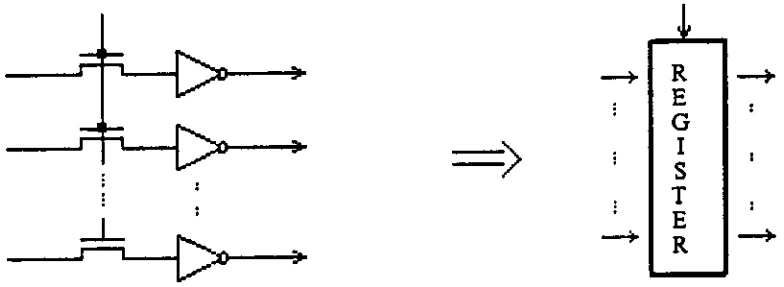


Fig. 7. A Dynamic Register

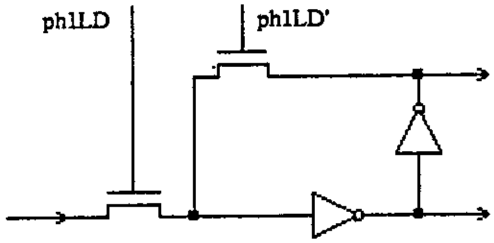


Fig. 8. A Selectively Loadable Dynamic Register Cell

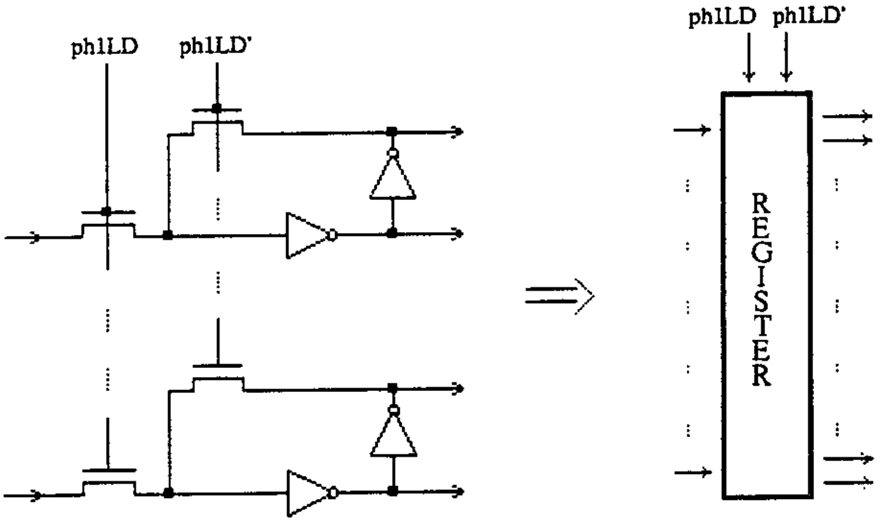


Fig. 9. A Selectively Loadable Dynamic Register

Designing a Subsystem

The ideas used to construct simple dynamic registers in the preceding section may be applied to the construction of more sophisticated and interesting subsystems. In this section we will describe the design of a *stack*. The methodology we use for this specific example we will find appropriate for a wide variety of functional subsystems. We first invent a "cell" which implements the most primitive function of the subsystem. This cell dictates a set of "timing" criteria necessary for its proper operation. The cell geometry together with the timing requirements dictates the design of control "circuits" which will surround an array of the basic cells. Once these control circuits are attached to the cell array, and the necessary "interconnections" are made, the entire assemblage constitutes a functional "module" with a well defined "interface" to the next higher level of design. This interface consists of a functional specification, a geometrical specification, and a set of timing requirements for the control inputs, data inputs, and data outputs.

The stack subsystem is commonly called a *last-in, first-out* (LIFO) stack. It is also known as a *pushdown* stack, although we will diagram it horizontally rather than vertically. It is a shift register array with three basic operations: during each full clock period (1) we can *push* in a new data word at one end of the array, pushing all previously entered words one word position further into the array, or (2) we can leave all words in their current position, or (3) we can *pop* out a word from the end of the array, pulling all previously entered words back out by one word position.

Figure 10a shows the structure of one horizontal row of the stack. Here we have implemented a shift register which can perform the following three operations: shift data left to right, hold data in place, or shift data right to left. There are four control signals used, two of them being active during φ_1 and two of them being active during φ_2. The signals φ_1 and φ_2 are our familiar two phase, non-overlapping clock signals.

In order for data to be shifted from left to right, the shift right control line (SHR) is driven *high* during φ_1, followed by driving the transfer right control line (TRR) *high* during φ_2. The bit of data appearing at the left is thus transferred by this operation onto the gate of the first inverter during φ_1, and thence to the gate of the second inverter during φ_2. In order for data to be held in place, the signal transfer left (TRL) is driven *high* during φ_1 and transfer right (TRR) is driven *high* during φ_2, causing the data to recirculate upon itself without shifting. Note that the data can be obtained at any time from the output of the first inverter. However, since new data may come to the gate of the first inverter during φ_1, the only safe time to take data out to the left is during φ_2. The transfer of data from right to left is caused by driving the shift left control (SHL) line *high* during φ_2, followed by driving transfer left (TRL) *high* during φ_1.

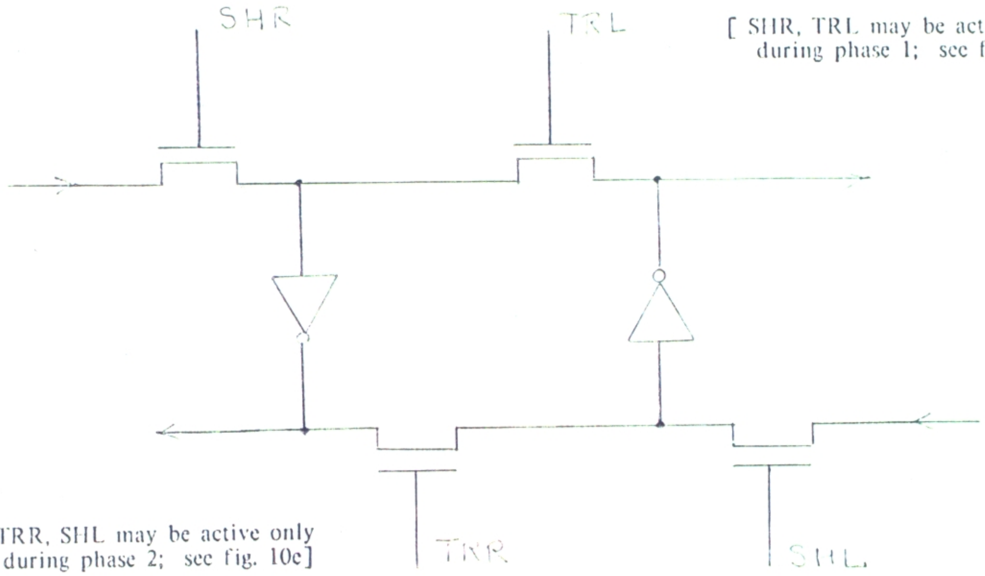
Figure 10b illustrates a possible topological structure of one horizontal row of the stack. There are two horizontal pathways on the diffusion level for shifting bits right or left. The two inverters for one stage of the row are nested between these paths. VDD, GND, and the four control lines run vertically in metal. The four pass transistors required for controlling the movement of data are conveniently implemented by short poly lines which cross the horizontal diffusion tracks at appropriate positions. Note that the entire row is composed of 180° rotations and repetitions of a basic cell containing one inverter.

In a typical implementation of the complete LIFO stack, a number of such rows run parallel to each other in the horizontal direction. The number of rows is equal to the width in bits of the data words involved. The control lines run vertically across the entire stack, perpendicular to the direction of data flow. For data words of any substantial width, the capacitive loading on the control signals would be sufficient to warrant use of super-buffer drivers.

The stack as a whole may be controlled with only two logic signals: one signalling *push*, and the other signalling *pop*. The activation of neither of these two signals causes data to recirculate in place, awaiting the next active instruction.

Let us consider how to derive, from *push* and *pop*, the control signals for driving the four control lines SHR, TRR, SHL, TRL. A possible scheme is shown in Fig. 10c. We use random logic for this purpose since only a few gates are required to control the large, regular array of circuit cells in the stack. The operation which determines what the stack will do during the subsequent clock phase is brought in on the path labeled OP. It is important to note in the following that only one signal path (OP) is required to bring in both *push* and *pop* logic signals, since these are active on mutually exclusive clock phases.

The control scheme is summarized in the timing diagrams in figure 10e. Here we see that holding OP *high* during φ_2, followed by *low* during φ_1, implements *push*. Holding OP *low* during both φ_1 and φ_2 causes the data to recirculate in place. Holding OP *high* during φ_1, followed by *low* during φ_2, implements *pop*. Thus, the single signal path, OP, is sufficient to carry both stack control signals into the stack.



[SHR, TRL may be active only during phase 1; see fig. 10e]

[TRR, SHL may be active only during phase 2; see fig. 10e]

Fig.10a. One Horizontal Row of the Stack

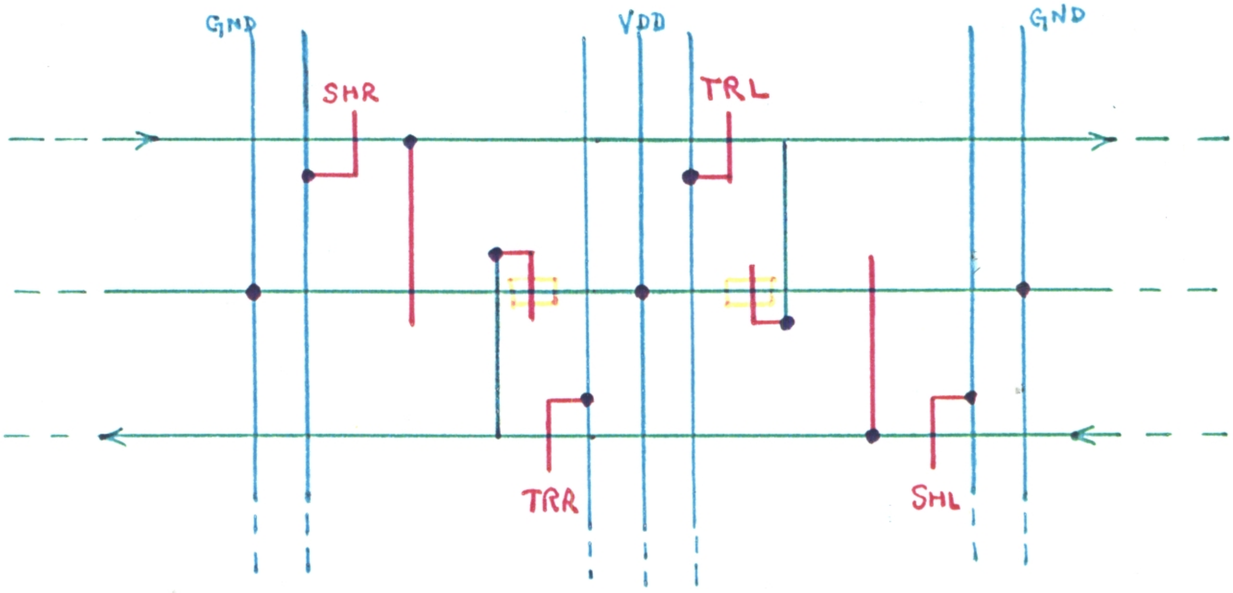


Fig.10b. Topology of One Horizontal Stack Row

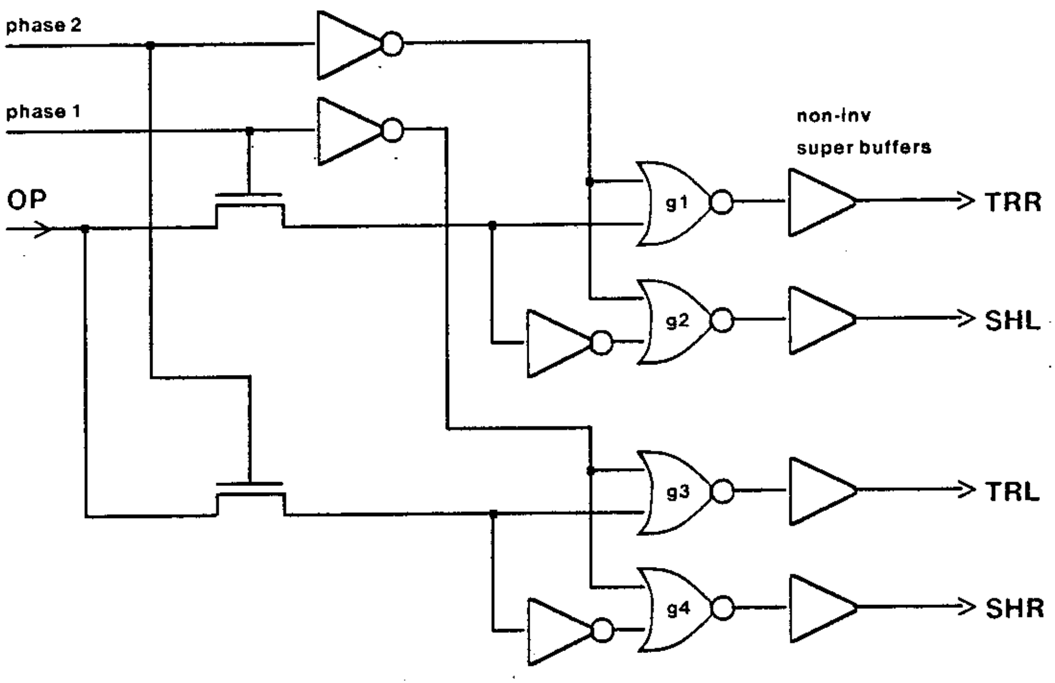


Fig. 10c. Generating the Stack Control Signals

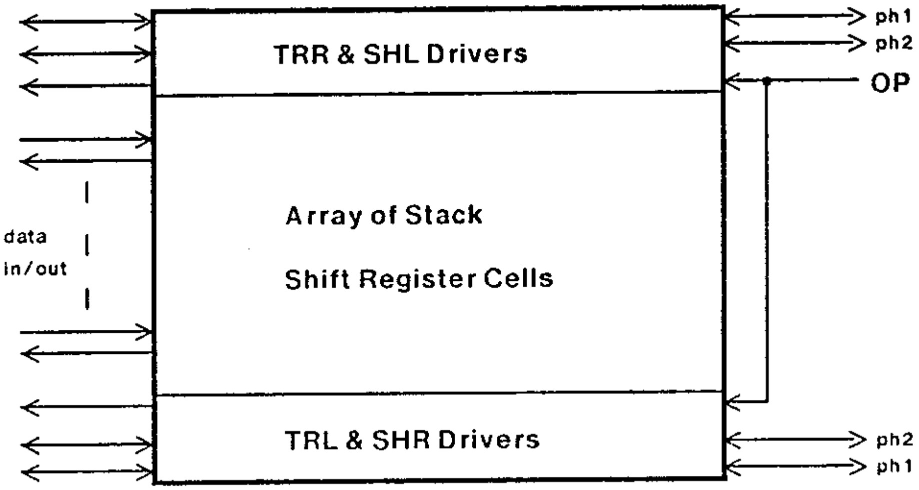
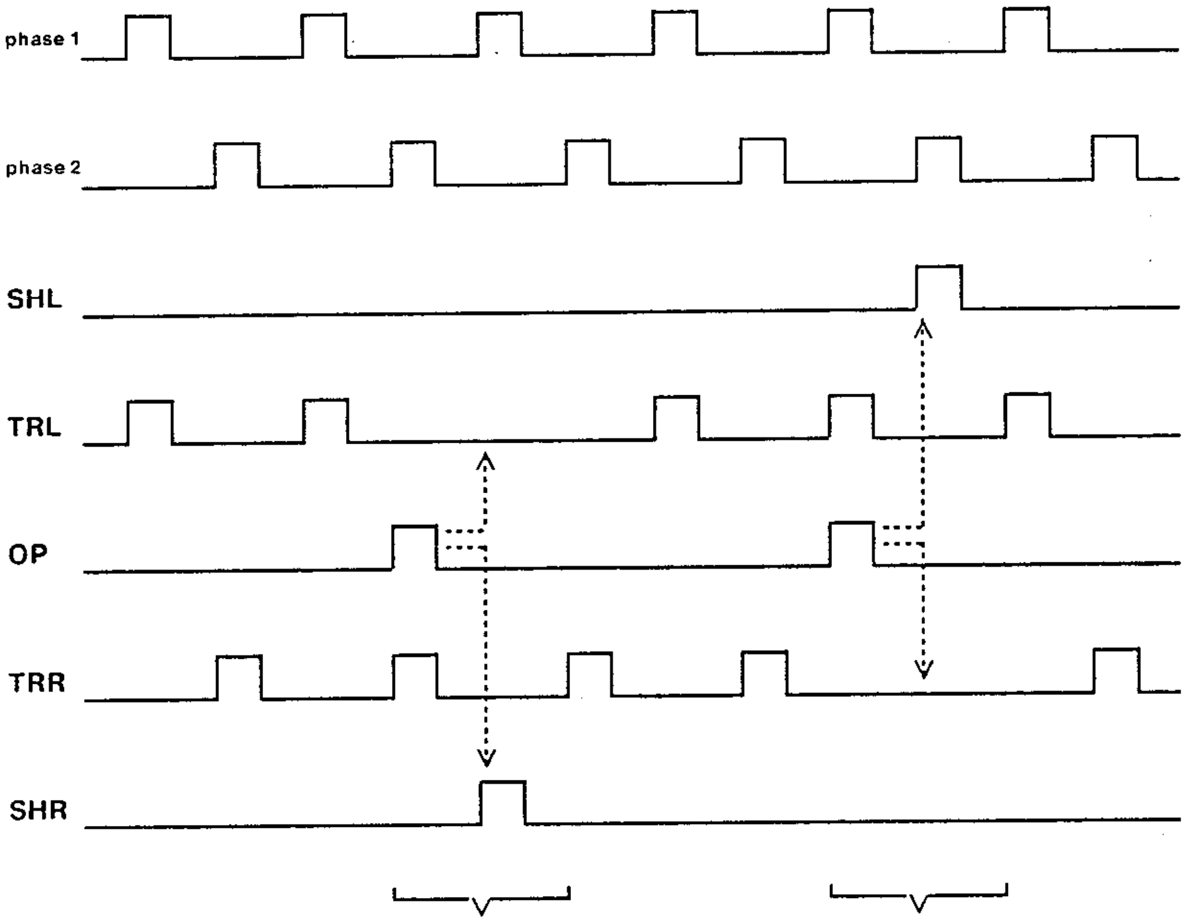


Fig. 10d. Stack Geometry and Interconnect Topology



PUSH:

OP: high in phase 2, and then low in phase 1;

Causes: SHR, not(TRL)

POP:

OP: high in phase 1, and then low in phase 2;

Causes: SHL, not(TRR)

Fig. 10e. Stack control Signal Timing Diagrams

(dcf10e.sil)

During φ_1, the OP signal is fed through the upper pass transistor into the inputs of the two NOR gates g_1 and g_2. The outputs for these NOR gates are *low* during this period, since φ_2' is *high.*

If the incoming OP signal is *high* while φ_1 is *high,* then the lower input of NOR gate g_2 will be *low.* Thus when φ_2' falls *low,* the output of g_2 will go *high,* thereby driving SHL *high.* If the OP signal is instead kept *low* while φ_1 is *high,* then the output of the NOR gate g_1 will go *high* on the fall of φ_2', thereby driving TRR *high* during φ_2.

During the period when φ_2 is *high* and either the shift left (SHL) or the transfer right (TRR) operation is being executed, the signal on the OP line is being stored on the corresponding input gates of the lower two NOR gates, g_3 and g_4. Thus, if OP is *high* while φ_2 is *high,* a logic-0 is stored on the input of the NOR gate g_4, and during the subsequent φ_1 *high* period, SHR will be driven *high.* Conversely, if OP is *low* while φ_2 is *high,* TRL will be driven *high* during the following φ_1 *high* period.

This kind of control scheme recognizes that there must be a lull period between any operation and its next occurrence. Control information is taken in during this period and set up for the subsequent operation. The scheme takes advantage of these lull periods, when possible, to perform other operations which can be done without conflict. It is an example of a fundamental design technique which can be extended to larger system structures.

When planning the overall architecture of a larger system, it is often useful to represent subsystems, such as the stack, using a higher level of symbolism. To be truly useful, such representations should, in addition to a functional definition, include the *topological* factors associated with the interconnection points of the subsystem and the *geometrical* factors of its shape and relative physical dimensions.

A system level sketch of one particular implementation of the stack is shown in figure 10d. Identical driver circuitry is placed along the top and bottom edges of the shift register array. The transfer right and shift left drivers which are set up during φ_1 (and active during φ_2) are placed along the top of the shift register array. The transfer left and shift right drivers which are set up during φ_2 (and active during φ_1) are placed along the bottom of the array. The OP bit and the clock signals are required on both the top and the bottom of the shift register array.

The integration of this subsystem into a larger integrated system design will require that the data in and out paths be matched to those of subsystems to which the array is connected, and that the φ_1, φ_2, and OP signals be available at either the left or right side of the array. By using system level representations that reflect as closely as possible the dimensions and locations of critical signals in all major subsystems, the interactions between topologies and dimensions of the subsystems can be assessed. The feasibility of an overall system architecture can thus be ensured prior to detailed design and layout.

Register to Register Transfer

From an implementation point of view it is often desirable to combine logic steering functions with the clocking of data into registers, since both require pass transistors as their elementary functional unit. An example is the shift-up register array shown in figure 6. From the next higher level system view, however, it is desirable to separate the two functions conceptually. In Fig. 11a we have shown some combination of inputs, X0 through Xn going through some combination of pass transistors, *which may or may not have logic functions attached,* into the input gates of some inverting logic elements. This combination of function is then abstracted into a register clocked on the phase during which the input pass transistors are turned on. Any logic function associated with the input pass transistors is considered part of the preceding combinational logic module. This viewpoint is an extension of the concept of dynamic register previously developed in figure 7.

Using this notation, any processing function can be built up using blocks of the form shown in Fig. 11b. Here we have a clocked input register, a block of strictly combinational logic *with no timing attached,* and an output register clocked on the opposite phase. In this case the inputs are stored in the input register during φ_1. They then propagate into and through the combinational logic (C/L), with the resulting outputs stored in the output register during φ_2. Any single data processing step can be viewed as a transfer from one such register to a second through a combinational logic block.

A sequence of such operations can be performed on a data stream by a series of such combinational blocks separated by registers as shown in Fig. 11c. Since different sets of data words in the stream may be operated upon at the same time, but at different locations, this data path is a type of pipelined processing structure. Such pipelined processing structures offer the opportunity for improved processing bandwidth by performing many different operations concurrently. Notice that the throughput rate of such a pipeline system of register to register transfer operations is limited by the delay time through the slowest of the combinational logic blocks. If no registers had been interposed between the function blocks, and each operand set separately run through the entire sequence of combinational logic modules, the throughput rate would be much lower.

In line with the ideas developed earlier in this chapter, the detailed functions performed by the combinational logic modules may often be implemented in circuit structures of very simple and regular topology. Control signals will in general cross the data path at right angles to the direction of data flow. Figure 11c illustrates sets of such control inputs as n_1 lines carrying the control function OP_1 into the first C/L module, n_2 lines carrying OP_2 into the second, etc.

The idea of data being processed while passing through combinational logic interspersed between register stages in a sequence of register to register transfers is a basic and important concept in the hierarchy of digital system architecture. We have already described the implementation of registers. The next sections will describe some ways to implement combinational logic functions.

Combinational Logic

Combinational logic modules contain no data storage elements. The outputs of a combinational logic module are functions only of the inputs to that module, provided that sufficient time has been allowed for those inputs to propagate through the module's circuitry.

In integrated systems, combinational logic design problems will typically fall within one of three general classes. The first is when a small amount of simple logic is required, for example to derive control signals at the periphery of a system module (as in the stack control signal generation) or to implement a simple function within a single circuit cell (which may then be replicated in a regular array). In these cases, traditional logic design procedures using static NAND and NOR gates can be applied. Such designs involving a few gates are usually rather simple, and can be produced by inspection rather than by use of formal minimization and synthesis procedures. Even in these simple cases, the minimum static logic gate implementation does not necessarily result in either the most regular, the minimum area, the minimum delay, or the minimum power design. In fact, we often find alternative techniques to the use of static logic gates, which in specific instances lead to "better" designs by one of these measures than would minimum gate implementations. For example, figure 12a shows a *selector* logic circuit (I. Sutherland), in which one of the inputs S_1, S_2, S_3, S_4 is selected for output by the control variables A, and B according to the function:

$$Z = S_1A'B' + S_2AB' + S_3A'B + S_4AB$$

This selector circuit is composed simply of poly paths crossing diffusion paths. Where depletion

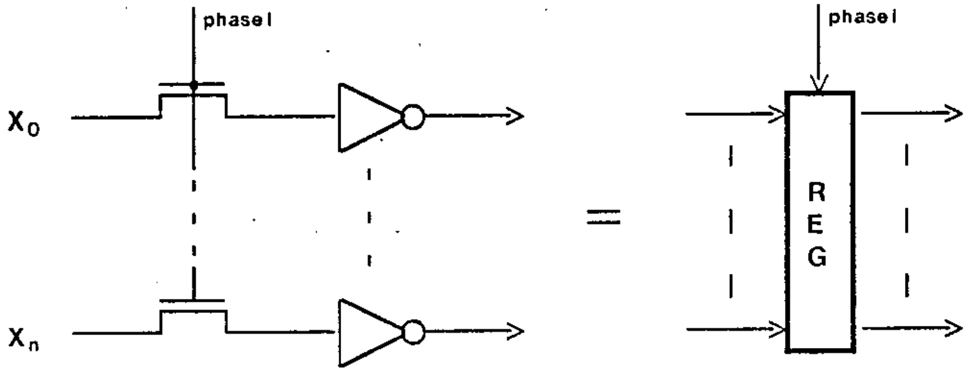


Fig. 11a. A Register

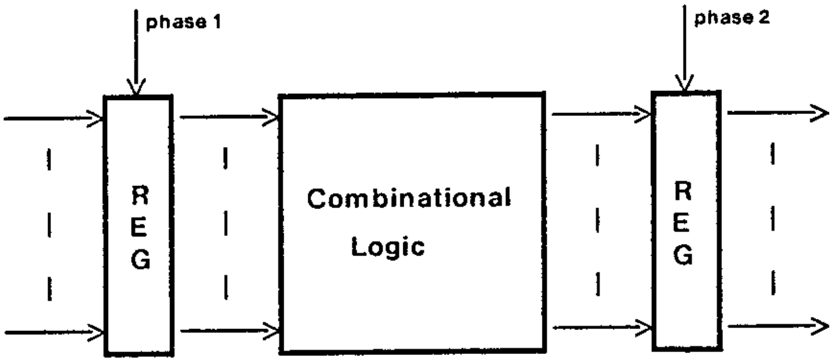


Fig. 11b. A Section of a Data Path

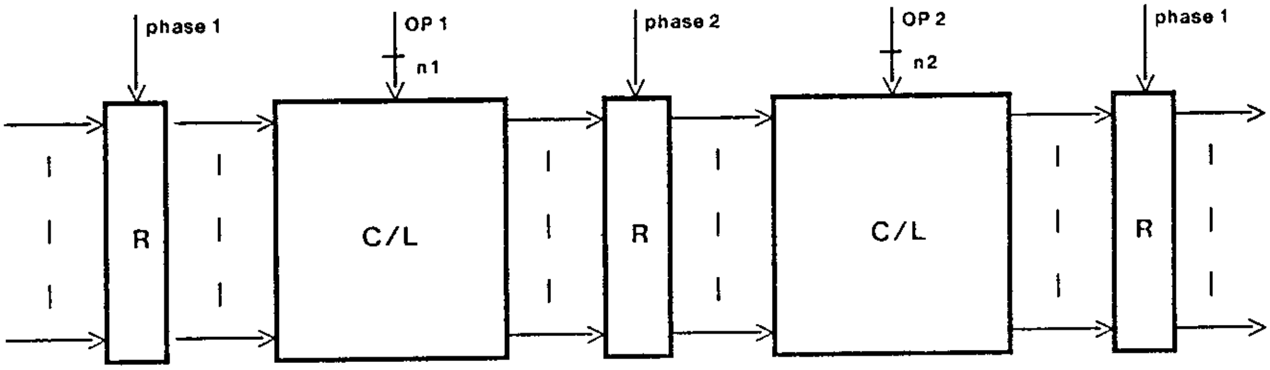


Fig. 11c. General Form for a Data Path

mode transistors are placed, the diffusion level path is always connected, thus placing control in the selectively located enhancement mode pass transistors, which function as simple switches. Figure 12c shows the circuit's paths from inputs to outputs using the "switch" abstraction for each of the pass transistors. For each possible combination of values of A and B, there is a path through the selector to Z from only one of the inputs S_i. For the specific inputs shown in the example in figure 12c, the signal S_2 propagates through to Z since both A and B' are *high.* Note that no static power is consumed by the circuit, and the area occupied by the circuit is minimal since no contact cuts are required within it. In chapter 5 we describe a very general and powerful arithmetic logic unit (ALU) which uses an array of such selector blocks to control a pass transistor carry network.

The second general class of combinational logic design problems are those rather complex functions for which clever ways of structuring topologically regular implementations have been discovered. As an example, consider the implementation of a *tally* function. This function has n inputs and n+1 outputs. The k^{th} output is to be *high,* and all other outputs *low,* if k of the inputs are *high.* The boolean equations representing this function for the simple case of three inputs are:

$$Z_0 = X_1'X_2'X_3'$$

$$Z_1 = X_1X_2'X_3' + X_1'X_2X_3' + X_1'X_2'X_3$$

$$Z_2 = X_1X_2X_3' + X_1X_2'X_3 + X_1'X_2X_3$$

$$Z_3 = X_1X_2X_3$$

If this function were designed with random logic consisting of active pullup, static logic gates, it would result in a topological kludge. Figure 12b shows a topologically regular implementation of the tally function. A major portion of the function is implemented using a regular array of identical cells each containing only two pass transistors. The design is based on the shift-up register idea presented earlier. A *high* signal propagates through the array from the pullup at the lower left. Whenever one of the variables X_i is *high,* the propagating *high* signal moves up to the next higher horizontal diffusion level path. Thus the number of paths it moves up equals the number of inputs X_i which are *high.* Logic-0 signals propagate through the array from the ground points to all other outputs.

Figure 12d shows the paths from inputs to outputs for this tally circuit, using the "switch" abstraction for the pass transistors. The figure shows a specific example of a set of inputs controlling the pass transistors of the circuit. Since two of the inputs are *high*, the logic-1 signal is shifted up two rows and emerges at Z_2.

This tally function design can be easily expanded to handle more than three inputs by simply extending the array structure upwards and to the right. However, remember that the delay through n pass transistors is proportional to n^2. Thus it may be necessary to insert level restoration prior to such extension. Similar comments apply to the extension of the selector circuit previously shown, or other pass transistor logic arrays one might invent.

The electronic logic gates traditionally used in digital design are unilateral elements: they allow a logic signal to propagate in one direction only. It should be noted that the pass transistor is a bilateral circuit element. It permits the flow of current, and thus the passage of a logic signal, in either direction when its gate is *high*. While this property of the pass transistor is not necessarily of fundamental importance in integrated systems, it is an interesting and occasionally useful one.

Early relay switching logic used switching contacts which were bilateral elements. Interesting discussions of relay switching logic are contained in both references R4 and R5. The tally array example just given is a basic *symmetric network* mapped directly into nMOS from relay switching logic (see R5, p.241). The mathematics of switching universally used in digital systems today was proposed by Claude Shannon (R7) in 1938. Shannon demonstrated that the calculus of propositions, based on the algebra of logic developed by Boole (R8), was directly applicable to relay switching circuits.

A third combinational logic design situation occurs when a complex function must be implemented for which no direct mapping into a regular structure is known. Methods for handling this situation are the subject of the next section.

In the design methodology developed in this text, the combinational logic between stages in the register to register transfer paths is often done by operations on the *charge* moving between stages, using pass transistors to perform these operations. Many researchers at the present time are searching for alternative structures and techniques for performing elementary logic functions, including the use of charge transfer devices[5].

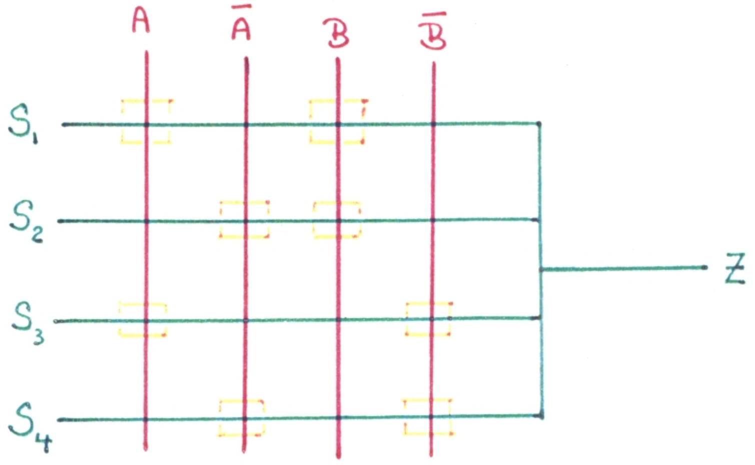


Fig.12a. Selector Logic Circuit

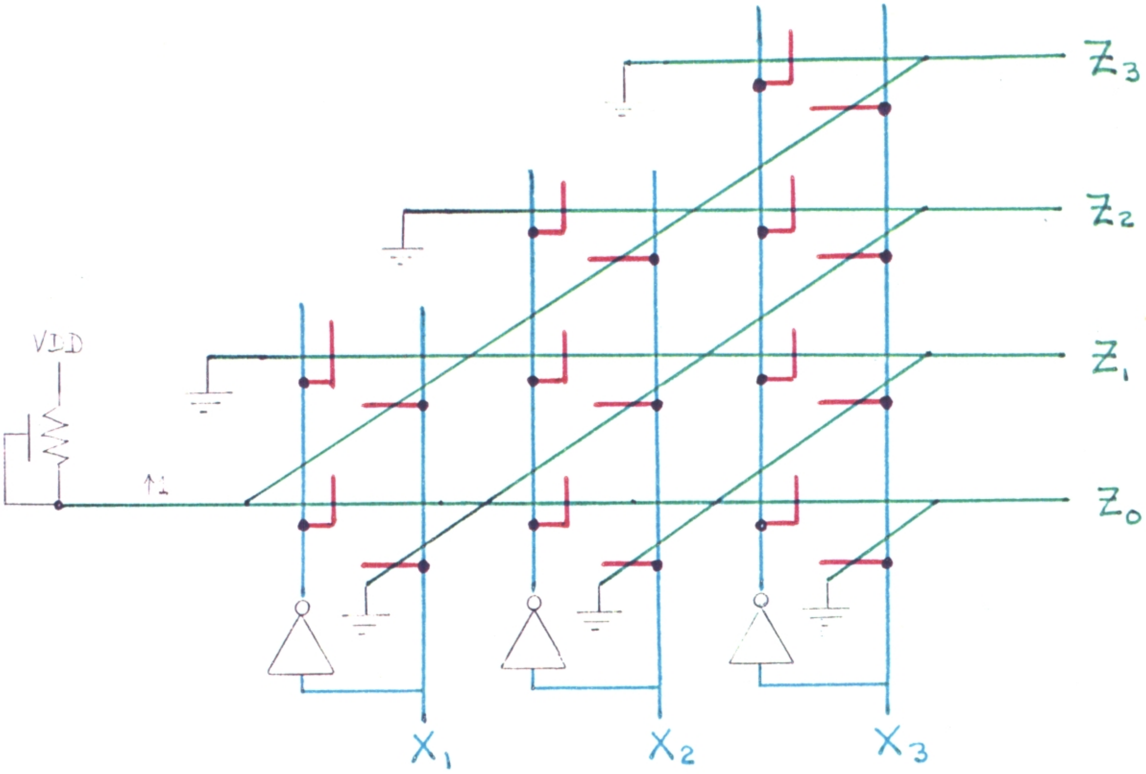


Fig.12b. A Tally Circuit

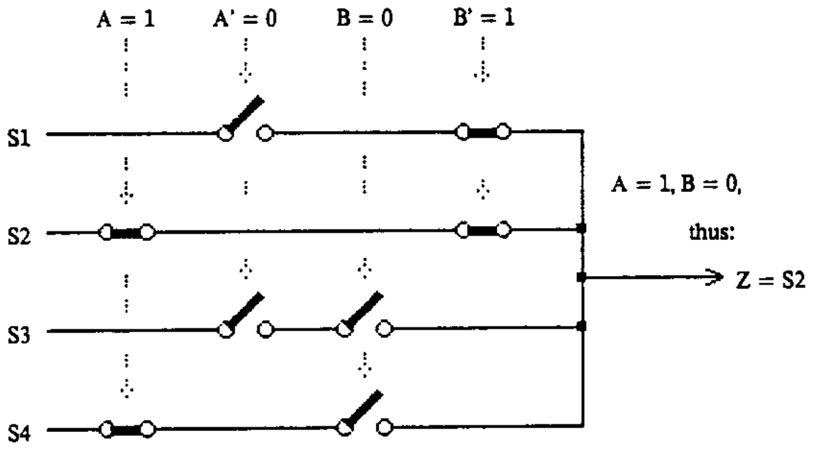


Fig. 12c. Example of Operation of Selector Circuit

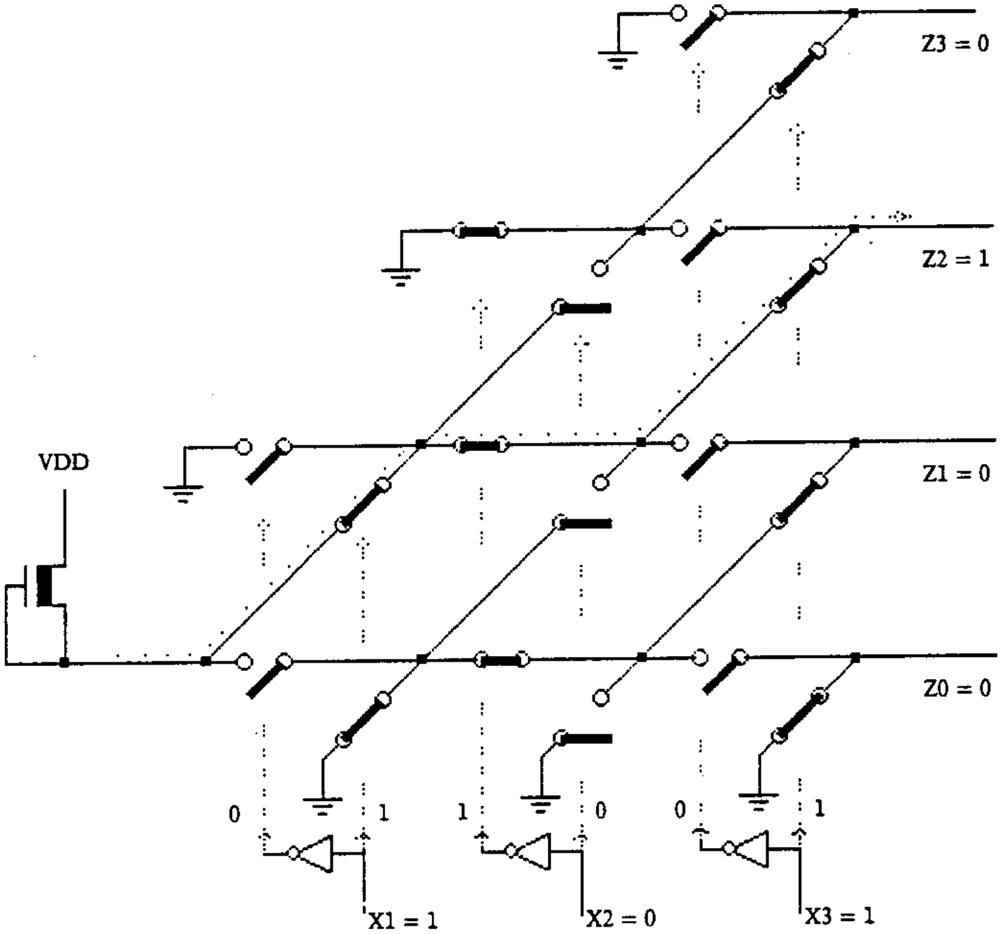


Fig. 12d. Example of Operation of Tally Circuit
(visualizing where the switches are)

(dcf12cd.press)

The Programmable Logic Array

On many occassions it is convenient to implement the combinational logic interspersed between register stages with regular structures of pass transistors. However, we will often encounter important combinational logic functions which do not map well into such regular structures. In particular, combinational logic used in the feedback paths of finite state machines is often highly complex and inherently irregular. Also, we may wish to delay binding the details of the logic functions used in finite state machine sequencing until most of the design is complete. If the combinational logic were implemented in an irregular structure, such changes could require a major redesign.

Fortunately, there is a way to map irregular combinational functions onto regular structures, using *programmable logic arrays* (PLA's) as described in this section. This technique of implementing combinational functions has a great advantage: functions may be significantly changed without requiring any major design or layout changes of the PLA structure.

One very general and regular way to implement a combinatorial logic function of n-inputs and m-outputs is to use a memory of 2^n words of m-bits each. The n-inputs form an address into the memory, and the m-outputs are the data contained in that address. Such a memory implements the full truth table for the output functions. Many systems are in fact built using memories as combinational logic elements. A common form of memory for this purpose is the *read-only memory* (ROM) where the data is permanently placed in the memory by a mask pattern, or by electrically altering the individual bit positions. There is one major difficulty with this approach: it is often the case that most of the possible input combinations cannot occur, due to the nature of the specific problem. Stated another way, many combinational logic functions require only a small fraction of all 2^n product minterms for a canonical sum of products implementation. In such cases, a ROM is very wasteful of area.

The *programmable logic array* (PLA) is a structure which has all the generality of a memory for implementing combinational logic functions. However, any specific PLA structure need contain a row of circuit elements only for each of those product terms that are actually required to implement a given logic function (see R4, Ch.4). Since it does not contain entries for all possible minterms, it is usually far more compact than a ROM implementation of the same function. To achieve full compaction, the various output functions must be jointly minimized before the PLA layout pattern can be defined. However, such minimization is not essential. Less than full compaction increases the independence of the different entries, so that changes in function may require only local changes in the PLA.

An illustration of the overall structure of a PLA is shown in figure 13a. The diagram includes the input and output registers, in order to show how easily these are integrated into the PLA design. The inputs, stored during φ_1 in the input register, are run vertically through a matrix of circuit elements called the AND-plane. The AND-plane generates specific logic combinations of the inputs and their complements. The outputs of the AND-plane leave at right angles to its inputs and run horizontally through another matrix called the OR-plane. The outputs of the OR-plane then run vertically and are stored in the output register during φ_2.

The circuit diagram of a specific programmable logic array is shown in figure 13b. This diagram will help to clarify the structure and function of the AND and OR-planes of the PLA. The input register bit for each input path is formed by a pass transistor clocked on φ_1 leading to both inverting and non-inverting super buffers. These buffers drive two lines running vertically through the AND-plane, one for the input term and one for its complement. The outputs of the AND-plane are formed by horizontal lines with pull-up transistors at their leftmost end. The function of the PLA's AND-plane is then determined by the locations and gate connections of pull-down transistors connecting the horizontal lines to ground.

Each output running horizontally from the AND-plane carries the NOR combination of all input signals which lead to the gates of transistors attached to it. For example, the horizontal row labelled R_3 has three transistors attached to it in the AND-plane, one controlled by A, one by B and one by C'. If any of these inputs is *high*, then R_3 will be pulled down towards ground and will be *low*.

Thus, $R_3 = (A + B + C')' = A'B'C$. Similarly, $R_4 = (A + B' + C)' = A'BC'$.

The OR-plane matrix of circuit elements is identical in format to the AND-plane matrix, but rotated 90 degrees. Once again, each of its outputs is the NOR of the signals leading to the gates of all transistors attached to it. In figure 13b for example. both R_3 and R_4 lead to the gates of transistors leading from the output line Z_4' to ground. If either R_3 or R_4 is *high*, Z_4' will be *low*. Thus, $Z_4' = \mathrm{NOR}(R_3,R_4) = (A'B'C + A'BC')'$. Up to this point the PLA implements the *NOR-NOR canonical form* of boolean function of its inputs.

The output lines of the OR-plane matrix are run into an output register formed by pass

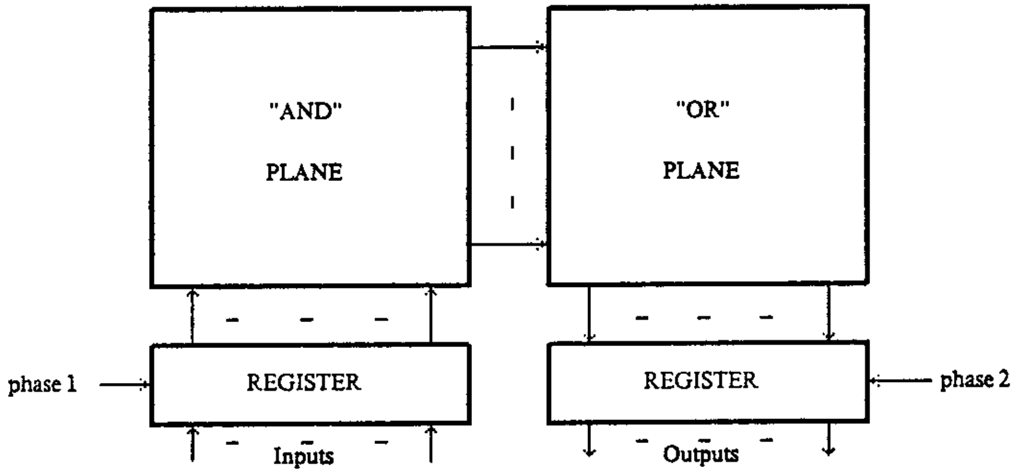


Fig. 13a. Overall Structure of the PLA

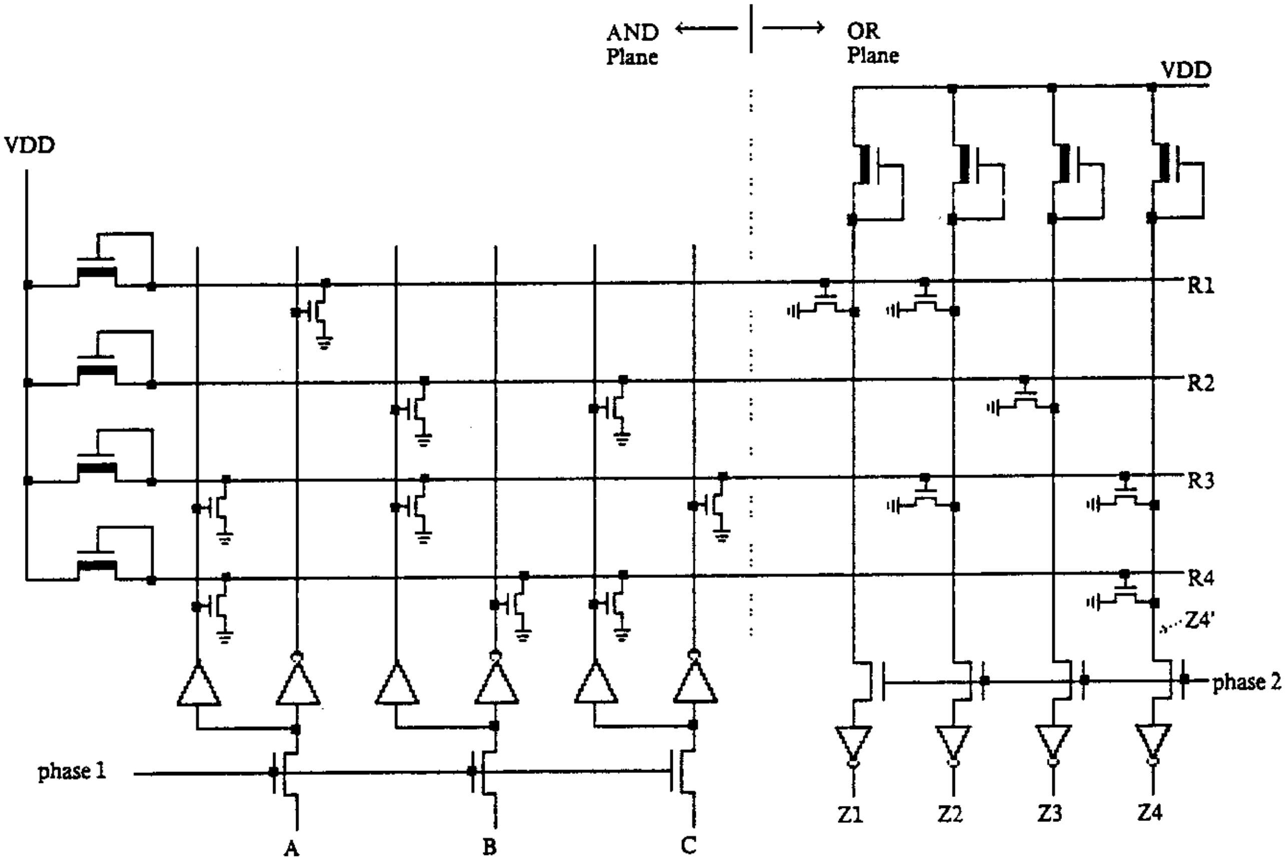


Fig. 13b. Circuit Diagram of PLA Example

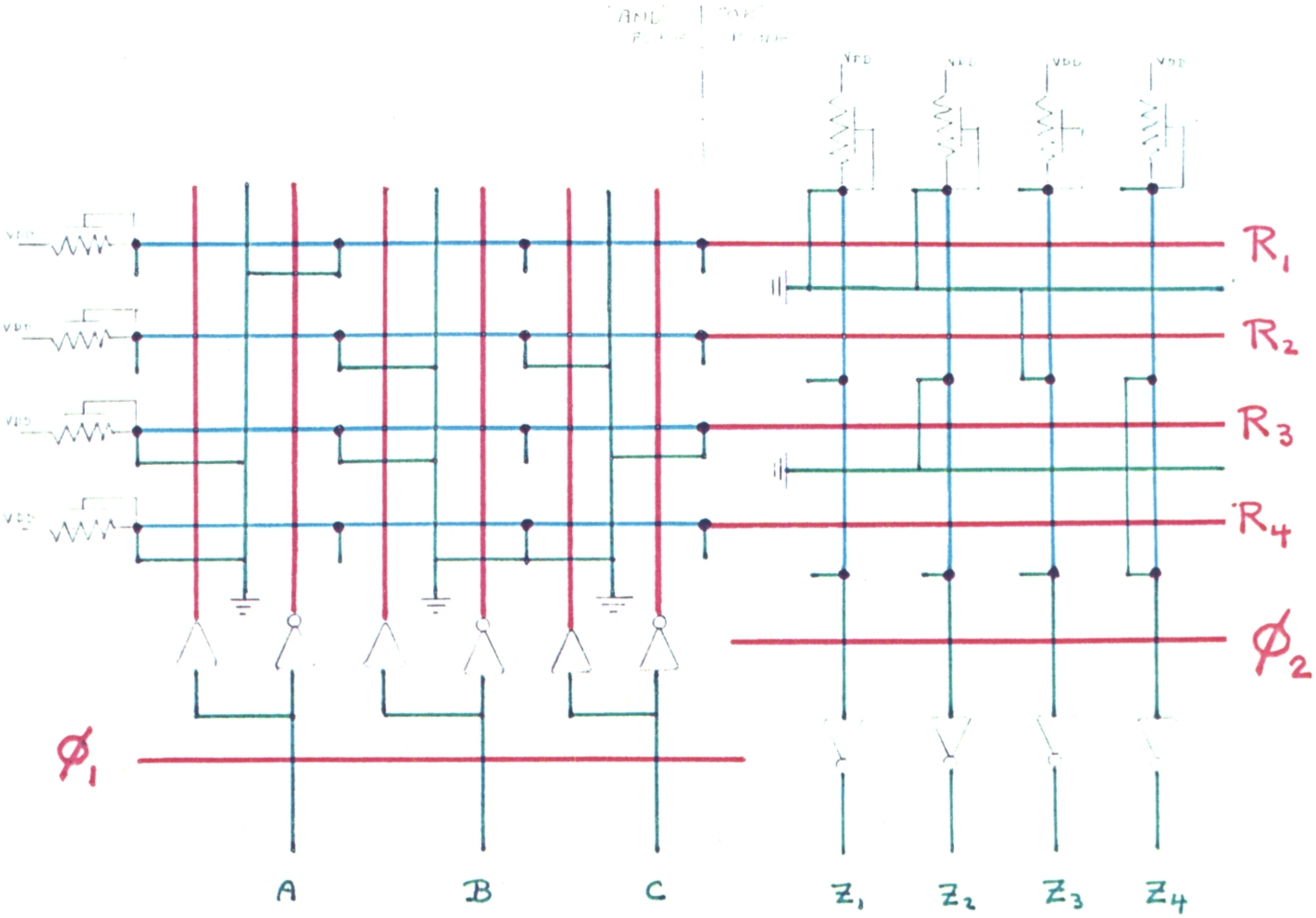


Fig.13c. Stick Diagram of PLA Example

Product Terms:

R_1 = (A')' = A

R_2 = (B+C)' = B'C'

R_3 = (A+B+C')' = A'B'C

R_4 = (A+B'+C)' = A'BC'

Outputs:

Z_1 = A

Z_2 = A + A'B'C

Z_3 = B'C'

Z_4 = A'B'C + A'BC'

transistors (clocked on φ_2) leading into inverting drivers. Note that the output Z_4 at this point is: Z_4 = A'B'C + A'BC. This expression illustrates why the two PLA planes, each implementing the NOR function, are usually referred to as the AND and OR-planes. Following the output register, the outputs appear directly as the *sum of products canonical form* of boolean functions of the PLA inputs, that is, the OR of AND terms. Each horizontal line of the PLA carries one *product term.*

Figure 13c shows one possible layout topology for implementing the PLA in nMOS circuitry. The example is the same circuit illustrated in figure 13b. The input lines crossing each plane are run in poly. The output lines from each plane are run in metal. Paths running to ground are placed between alternate poly lines, on the diffusion level. It is then a simple matter to form the pulldown transistors connecting the metal output lines to ground. They are selectively located diffusion lines under the appropriate input poly lines.

Although the PLA may implement a very irregular combinational function, the irregularity is confined to the irregular locations of pulldown transistors which "program" the function. The overall structure and topology of the PLA are very regular. Note that its overall shape and size is a function of the parameters: (i) the number of inputs, (ii) the number of product terms, (iii) the number of outputs, and (iv) the length unit λ.

Finite State Machines

In many cases in the processing of data, it is necessary to know the outcome of the current processing step before proceeding with the next. Results of the current step may be used as inputs in the next step. The configuration shown in figure 14a can be used to implement a processing stage having this requirement. A typical register to register transfer stage has been modified by simply feeding back some of its outputs to some of its inputs. This structure implements a form of sequential machine known as a *finite state machine.*

The feedback signals form a binary number which may be regarded as identifying the *state* of the machine. The value of this number is stored, along with the external inputs, in the first register during φ_1. These combined inputs then propogate through the combinational logic. The resulting outputs are stored in the second register during φ_2. The falling edge of φ_2 must occur a sufficient time later to insure that all signals have propagated through the combinational logic.

Each complete machine cycle, consisting of φ_1 followed by φ_2, results in two new sets of outputs: (i) the external outputs which are typically used for controlling other units of the system, and (ii) a new feedback number, which defines the *next state* of the machine. This process repeats during each clock period. The number of possible states is determined by the number of bits in the feedback path, and is *finite.*

There are a number of ways of abstractly representing the states, the required state transitions, and the outputs of sequential machines under given input sequences. Possible representations include state diagrams, transition tables, boolean or numerical difference equations, etc. A large body of theory has been developed concerning sequential machines. The serious reader will benefit from a further study of the results of switching theory on this subject (R3, R4).

Implementations of simple finite state machines are used to produce the very lowest level of system control sequencing, since they can autonomously generate control sequences. The sequential machine having a finite number of states is a very important element in the hierarchy of fundamental concepts used in integrated system architecture.

The configuration shown in figure 14a implements a *synchronous* machine, since the feedback loop is only activated at times determined by the clock signals. In any clock period k, the output terms Z_j and the next state terms Y_f are valid during $\varphi_1(k)$. They are functions of the external inputs X_i and feedback terms Y_f which were valid during $\varphi_1(k-1)$.

If a sequential machine contains a feedback loop which is continuously active, then it may begin a response to a change in inputs or state at any time, rather than just at fixed clock times. Such a sequential machine is referred to as an *asynchronous sequential machine.* The analysis of asynchronous sequential machines and their implementation is far more complex than that of synchronous ones. Great care must be exercised to avoid any difference in state sequencing and outputs under arbitrary differential delays of signals through the circuit paths of such machines (R3. Ch.5). There will be only a few special cases where we use the asynchronous form of sequential machine (Chapter 7), and these will be subject to detailed analysis.

Where sequential machines are required within integrated systems, we will generally implement them in synchronous form. Synchronous machines are rather easy to implement correctly, and fit naturally into the two phase clocking scheme used for moving data around within our systems.

However, the reader should carefully note that an implementation of a synchronous sequential

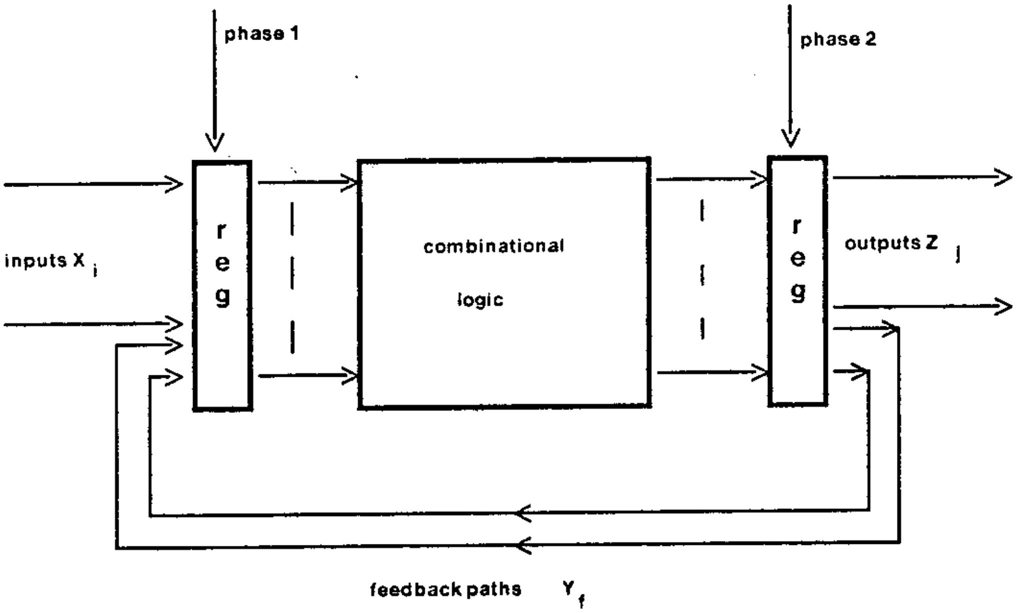


Fig. 14a. Feedback in Register Transfer Path, Implementing a Finite State Machine

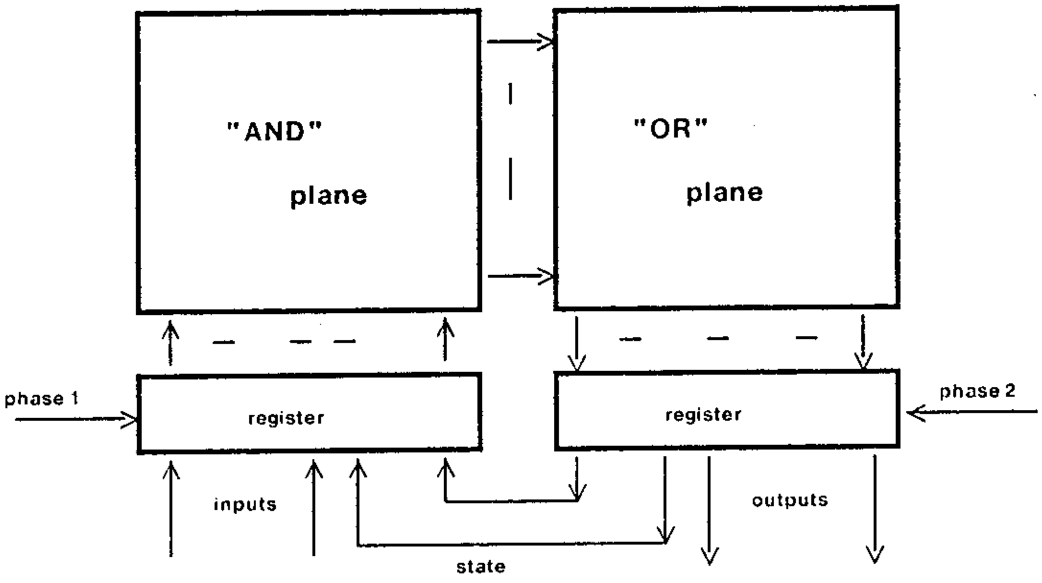


Fig. 14b. PLA Implementation of a Finite State Machine

machine functions correctly only if the delays in the circuit paths are sufficiently short compared to the clock period. If we were to implement many copies of a particular machine, the probability of correct function for any given copy is thus a function of (i) the clock period used, and (ii) the distribution of differential delays in that copy's signal paths. Our estimate that a particular copy will function correctly is thus based in part on assumptions about the ratio of likely deviations in circuit delays to the clock period. A discussion of delays in MOS circuits is given in chapter 1.

There is a very straightforward way to implement simple finite state machines in integrated systems: we use the PLA form of combinational logic and just feed back some of the outputs to the inputs, as illustrated in figure 14b. The circuit's structure is topologically regular, has a reasonable topological interface as a subsystem, and is of a shape and size that are functions of the appropriate parameters. The function of this circuit is determined by the "programming" of its PLA logic. If, for example, early in a design cycle there is some uncertainty in the details of the desired sequencing of such a circuit, it is easy to provide layout space for extra, unused inputs, minterms, or outputs as contingencies.

The following simple example will help illustrate the basic concepts of finite state machines and their implementation in nMOS circuitry. A busy highway is intersected by a little used farmroad, as shown in figure 15a. Detectors are installed which cause the signal C to go *high* in the presence of a car or cars on the farmroad at the positions labelled C. We wish to control traffic lights at the intersection, so that in the absence of any cars waiting to cross or turn left on the highway from the farmroad, the highway lights will remain green. If any cars are detected at either position C, we wish the highway lights to cycle through caution to red, and the farmroad lights then to turn green. The farmroad light is to remain green only while the detectors signal the presence of a car or cars, but never longer than some fraction of a minute. The farmroad light is then to cycle through caution to red, and the highway light then to turn green. The highway light is not to be interruptible again by the farmroad traffic until some fraction of a minute has passed.

A state diagram model of a finite state machine to control the lights is sketched in figure 15b. This diagram identifies four possible states of the machine, and indicates the input conditions which cause all possible state transitions. A block diagram of the PLA circuit implementing the machine is shown in figure 15c. The circuit uses the signal C as an input, and provides outputs HL and FL which encode the colors of the highway and farmroad lights it controls. Note that a timer is used to provide, as controller inputs, the short and long timeout signals (TS, and TL), at appropriate times following a start timer (ST) signal output from the controller. This timer could be implemented as a digital counter in the same nMOS circuitry. Another abstract model describing the desired function of the controller is given in the state transition table in figure 15d, which contains similar information to that in the state diagram.

The detailed sequencing of the machine under various input sequences is described by both the state diagram and transition table models of the controller. Consider starting in the state HG, where the highway lights are green. The machine remains in state HG as long as either no cars are detected or the long timeout has not occurrred, in other words as long as (C)AND(TL) = 0. After the long timeout occurs, if any cars are detected, the machine restarts the timer and changes state to HY, where the highway lights are yellow. It remains in state FY only until the short timeout occurs, and then restarts the timer and changes to state FG, where the farmroad lights are green. It remains in state FG until either no cars are detected or the long timeout occurs, i.e. (C)'OR(TL) = 1. Then it restarts the timer and changes to state FY, where the farmroad light is yellow. It remains in state HY only until the short timeout occurs. It then restarts the timer and changes to state HG, the starting state.

The locations of transistors in the PLA light controller circuit can be determined by "hand assembling" the "program" specified in the "symbolic" transition table in figure 15d, resulting in the encoded state transition of figure 15e. First we assign codes to the states. In the example: state HG is encoded as (Y_0,Y_1) = (0,0); HY as (0,1); FG as (1,1); and FY as (1,0). Next, we assign codes to the output light control signals: green is encoded as (0,0), yellow as (0,1), and red as (1,0). We now form the encoded state transition table by constructing one row for each product term implied by the symbolic table of fig. 15d. A row in 15d specifying a state transition as a function of a single input variable or single product term of input variables produces a single row in table 15e. A row in table 15d specifying a state transition as a function of a sum or sum of products of input variables, leads to a corresponding number of rows in table 15e.

Placement of the transistors within the PLA matrices follows directly from the encoded state transition table:

(i) For each logic-1 in the next state and output columns in the table, we run a diffusion path *from* the corresponding next state or output line in the PLA OR-plane, *under* the corresponding product term line, *to* ground. This creates a transistor controlled by the product term line. Then,

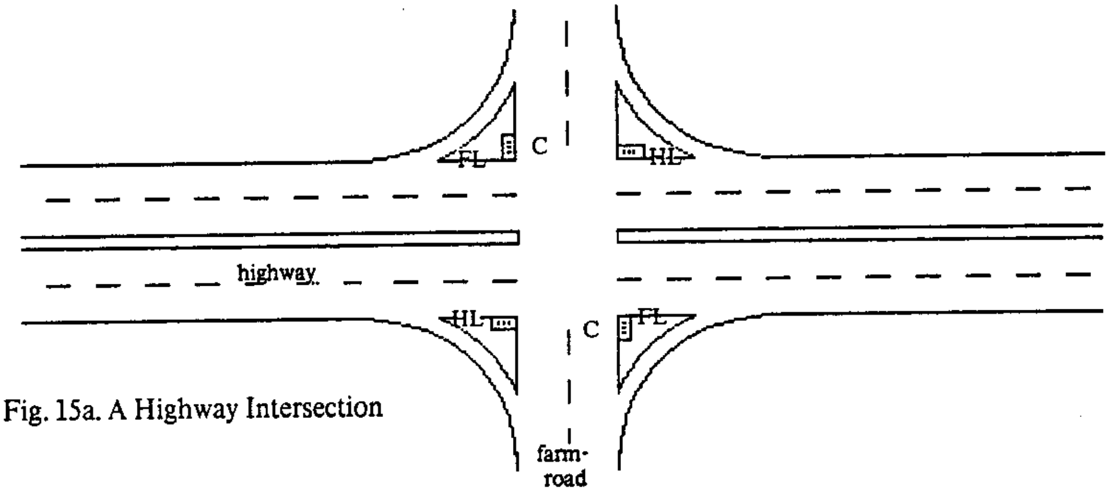


Fig. 15a. A Highway Intersection

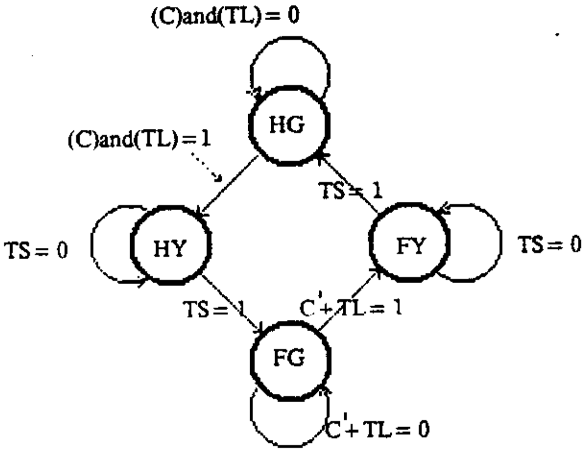


Fig. 15b. Light Controller State Diagram

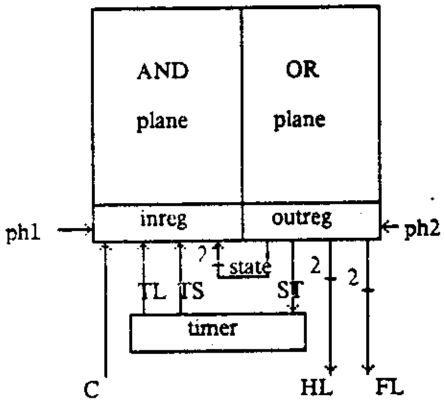


Fig. 15c. Controller Block Diagram

In Present State:	If Inputs are: *	Next State will be:	and Outputs are:		
			HL	FL	ST
Highway Green	(Cars)and(TimeoutL) = 0	Highway Green	Green	Red	No
	(Cars)and(TimeoutL) = 1	Highway Yellow	Green	Red	Yes
Highway Yellow	TimeoutS = 0	Highway Yellow	Yellow	Red	No
	TimeoutS = 1	Farmroad Green	Yellow	Red	Yes
Farmroad Green	(Cars)'or(TimeoutL) = 0	Farmroad Green	Red	Green	No
	(Cars)'or(TimeoutL) = 1	Farmroad Yellow	Red	Green	Yes
Farmroad Yellow	TimeoutS = 0	Farmroad Yellow	Red	Yellow	No
	TimeoutS = 1	Highway Green	Red	Yellow	Yes

Fig. 15d. Transition Table for the Light Controller

* Inputs not listed = don't cares

Stored during φ_1 in INREG				*Stored during φ_2 in OUTREG*						
Inputs:			*Present State:*	*Next State:*	*Outputs:*					*Product terms:*
C	TL	TS	Y_{p0},Y_{p1}	Y_{n0},Y_{n1}	ST	HL_0	HL_1	FL_0	FL_1	
0	X	X	0, 0 (HG)	0, 0 (HG)	0	0	0	1	0	R1
X	0	X	0, 0 (HG)	0, 0 (HG)	0	0	0	1	0	R2
1	1	X	0, 0 (HG)	0, 1 (HY)	1	0	0	1	0	R3
X	X	0	0, 1 (HY)	0, 1 (HY)	0	0	1	1	0	R4
X	X	1	0, 1 (HY)	1, 1 (FG)	1	0	1	1	0	R5
1	0	X	1, 1 (FG)	1, 1 (FG)	0	1	0	0	0	R6
0	X	X	1, 1 (FG)	1, 0 (FY)	1	1	0	0	0	R7
X	1	X	1, 1 (FG)	1, 0 (FY)	1	1	0	0	0	R8
X	X	0	1, 0 (FY)	1, 0 (FY)	0	1	0	0	1	R9
X	X	1	1, 0 (FY)	0, 0 (HG)	1	1	0	0	1	R10

Fig.15e. Encoded State Transition Table for the Light Controller

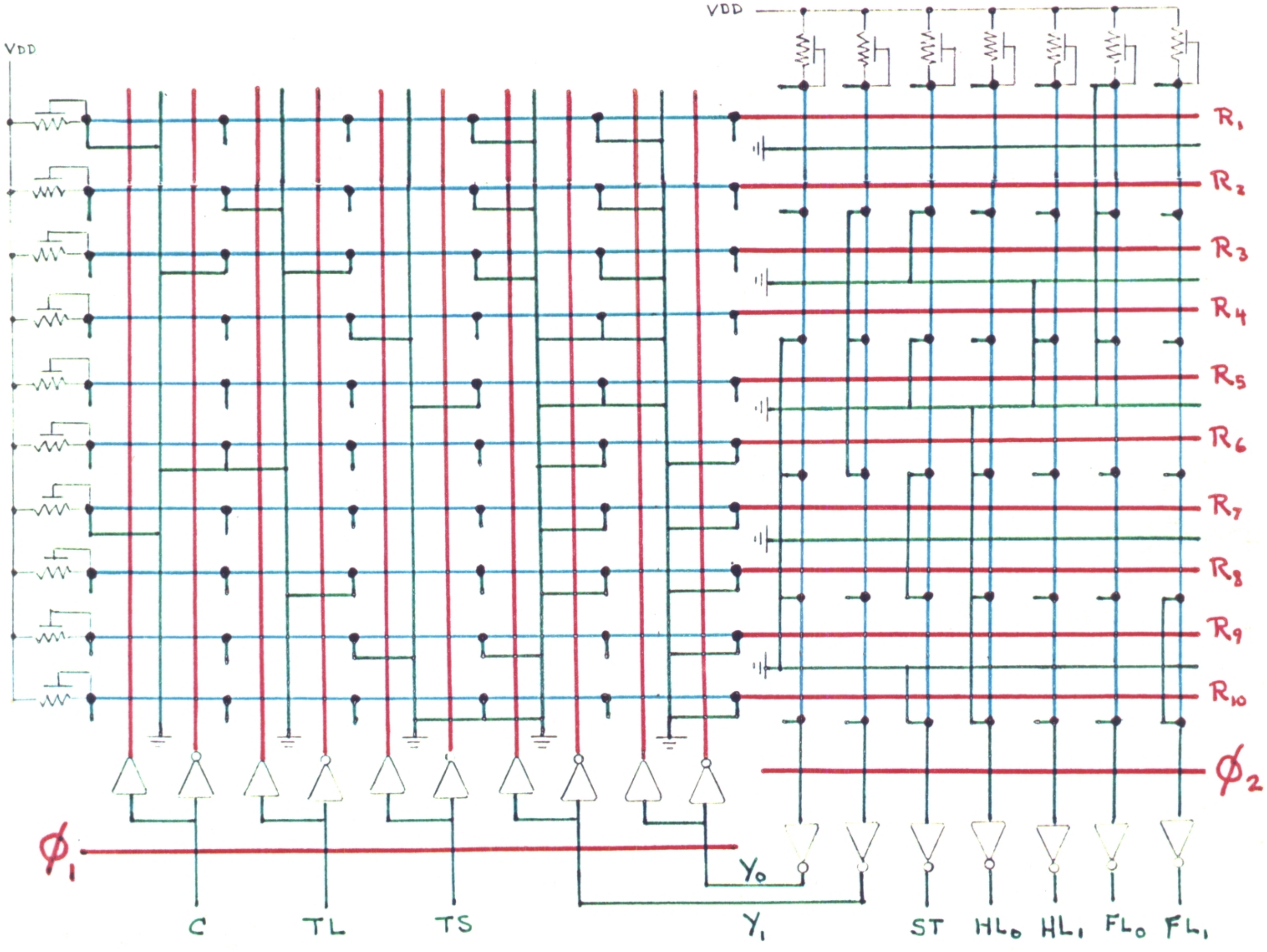


Fig.15f. PLA Sequential Circuit Implementing the Light Controller

if that controlling product term line is ever *high*, the path to the output inverter will be *low*, and the output will be *high*. The output line will be *low* unless some product term line controlling it is *high*.

(ii) For each logic-1 in the input and present state columns in the table, we run a diffusion path *from* the corresponding product term line, *under* the corresponding *inverted* input or state line in the PLA AND-plane, *to* ground. The transistor thus created is controlled by the *inverted* input or state line. Whenever that controlling line crossing the AND-plane is *high*, the product term line will be *low*.

(iii) For each logic-0 in the input and present state columns in the table, we run a diffusion path *from* the corresponding product term line, *under* the corresponding *non-inverted* input or state line in the PLA AND-plane, *to* ground. The transistor thus created is controlled by the *non-inverted* input or state line. Whenever that controlling line crossing the AND-plane is *high*, the product term line will be *low*.

Note that if all lines which control the transistors connecting a given product term line to ground are *low*, then that product term line will be *high*. Otherwise it will be *low*.

The PLA circuit in figure 15f is programmed from the transition table in figure 15e, according to the rules above, and implements the traffic light controller. Note that this LSI implementation does not exactly strain itself to meet the time response requirements of the control problem: it can run at a clock rate at least 10^7 times as fast as required. Also, note that the PLA controller is roughly $(150\lambda)^2$ in area. Using the 1978 value of $\lambda = 3\mu m$, this controller is $(450\mu m)^2 \sim 0.002$ cm^2 in area. A PLA controller this size may contain over 150 transistors, but occupies only 1/125th of the area of a typical 0.25 cm^2 silicon chip in 1978. By the late-80's, as λ scales down towards its ultimate limits, such a controller will require only ~ 1/25,000th of the area of such a chip.

As we will see in later chapters, a data processing machine of any desired complexity can be created by interconnecting register to register data processing paths constructed along the lines of that shown in figure 11c, such paths being controlled by finite state machines implemented as shown in figure 14b. The data paths form the "highways" for the movement of data, under control of the finite state machine "traffic controllers".

Towards a Structured Design Methodology

The task of designing very complex systems involves managing, in some highly structured way, the space and time relationships between the various levels of system building blocks so that the entire system will function as intended when it is finished. The beginnings of a structured design methodology for VLSI systems can be produced by merging together in a hierarchy the concepts presented in this chapter. Designs are then done in a "top down" manner, but with a full understanding by the architect of the successive lower levels of the hierarchy.

To begin, we plan our digital processing systems as combinations of register to register data transfer paths, controlled by finite state machines. Then the geometric shapes, relative sizes, and interconnection topologies of all subsystem modules are collectively planned so all modules will merge together snugly, with a minimum of space and time wasted by random interconnect wiring. Storage registers are typically constructed by using charge stored on input gates of inverting logic. The combinational logic in the data paths is typically implemented using steering logic composed of regular structures of pass transistors. Most of the combinational logic in the finite state machines is typically implemented using PLA's. All functioning is sequenced using a two-phase, non-overlapping clock scheme.

When viewed in its entirety, a system designed in this manner is seen as a hierarchy of building blocks, from the very lowest level device and circuit constructs, on up to and including the high level system software and application programs in which the intended functions of the system are finally expressed. Individuals who understand the key concepts of each level in this hierarchy will recognize that the boundaries between levels are rather elastic ones. Each level of activity might best be optimized not on its own as a specialty, but as it fits into an overall systems picture. For example, the activity "logic design" in integrated systems might best be conceptualized as the search for techniques and inventions which best couple the physical, topological, and geometric properties of integrated devices and circuits with the desired properties of digital VLSI systems. The search for alternative components for any given design hierarchy, and the search for alternative hierarchies, will be done best by those who span more than one specialty.

A particularly uniform view of such a system of nested modules emerges if we view every module at every level as a finite state machine or data path controlled by a finite state machine. At the lowest level, elements such as the stack and register cells may be viewed as state machines with one feedback term (the output), two external inputs (the control signals), and a one bit state register. These rudimentary state machines are grouped in a structured manner to form portions of a state machine, or data path controlled by a state machine, at the next level of the hierarchy. Structured arrays of identical state machines often provide a mechanism for distributing processing among memory cells[R6], thus enabling vast increases in processing bandwidth. Although in some cases the feedback paths are used in rather specialized ways, the state machine metaphor still provides a precise description of module behavior. The entire system may thus be viewed as a giant hierarchy of nested machines, each level containing and controlling those below it. A detailed quantitative treatment of certain hierarchically organized machines is given in chapter 9.

In chapters 5 and 6 we will apply the design methodology developed in this chapter to the design of a digital computer system. A one chip implementation of the data path portion of this computer system is illustrated in the frontispiece. Consistent use of the described design methodology resulted in a design of great regularity, short delay times, low power consumption, and high logical processing capability. As we will see in chapter 4, regular designs, with small numbers of basic circuit cell types replicated in two dimensions to form subsystems, also have significant implementation advantages over less structured designs.

References

1. - - - *polycell reference:* - - -

2. I. E. Sutherland, C. A. Mead, "Microelectronics and Computer Science", *Scientific American*, September 1977, pp. 210-228.

3. J. D. Williams, "Sticks -- A New Approach to LSI Design", M.S.E.E. Thesis, Dept. of Electrical Engineering, M. I. T., June, 1977.

4. W. M. Penney, L. Lau, Eds., "MOS Integrated Circuits", Van Nostrand, 1972. Chapter 5.

5. C. H. Sequin, M. F. Tompsett, "Charge Transfer Devices", Academic Press, 1975, Ch. VIII.

Reading References

R1. C. G. Bell, A. Newell, "Computer Structures: Readings and Examples", McGraw-Hill, 1971, contains an excellent discussion of the levels in the hierarchy of computer architecture, and many specific examples of computer structures.

R2. B. Soucek, "Microprocessors and Microcomputers", John Wiley, 1976, is a good introductory reference containing sections on basic digital design, and on the interfacing and programming of a number of present day microprocessors.

R3. D. L. Dietmeyer, "Logic Design of Digital Systems", Allyn and Bacon, 1971, is a comprehensive text on switching theory and logic design.

R4. Z. Kohavi, "Switching and Finite Automata Theory", McGraw-Hill, 1970, is another good text on switching theory.

R5. S. H. Caldwell, "Switching Circuits and Logical Design", John Wiley, 1958, is an early text containing interesting material on relay contact networks.

R6. S. H. Unger, "A Computer Oriented Towards Spatial Problems", Proc. of IRE, vol. 46, no. 10, pp. 1744-1750, Oct. 1958, is an early paper describing a spatially distributed processor, anticipating present strategies for commingling processing and memory.

R7. C. E. Shannon, "Symbolic Analysis of Relay and Switching Circuits", *Trans. of AIEE*, Vol. 57, 1938, pp. 713-723, is the classic paper proposing a method for the mathematical treatment of switching circuits.

R8. G. Boole, "An Investigation of the Laws of Thought", London, 1854, reprinted by Dover Publications, contains a presentation of the algebra of logic on which Shannon based his switching algebra.