A Design Methodology for Self-Timed Systems

by

Narinder Pal Singh

February 1981

This work was supported in part by the National Science Foundation under the research grants MCS75-04060 A01 and MCS79-15255.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge Massachusetts 02139

A Design Methodology for Self-Timed Systems

by Narinder Pal Singh

Submitted to the Department of Electrical Engineering and Computer Science on 16 January in partial fulfillment of the requirements for the degree of Master of Science.

Abstract

This thesis presents a design methodology for self-timed systems which will be extremely attractive for implementing systems in VLSI. Self-timed systems are characterized by the absence of a timing reference to which all operations are synchronized. Currently most systems are implemented using a synchronous design methodology where all operations are synchronized to a global clock. However, this approach will not be attractive in the future for implementing systems in VLSI due to the high communication costs in VLSI and the prohibitive task of managing timing constraints global to a VLSI integrated circuit.

The methodology proposed in this thesis defines a set of modules which form the building blocks for implementing an arbitrary self-timed system. The various module types are based on familiar programming constructs such as iterations, conditionals and constructs that aid in the activation and synchronization of parallel processes, such as, forking and joining. The modules of a self-timed system communicate with each other via an asynchronous communication protocol, and the correct behavior of the system is independent of the delays in the communication medium. This methodology simplifies the design effort by restricting the timing constraints to be local to the modules of the system.

Thesis Supervisor: Jack B. Dennis
Title: Professor of Computer Science and Engineering

Keywords: self-timed systems, speed independent logic, asynchronous design methodology, VLSI.

Acknowledgments

I am indebted to Professor Jack B. Dennis for his guidance and support which has greatly improved the contents of this thesis and made all of this possible. I am also thankful to Clement Leung, Andy Boughton, Dean Brock and Bill Ackerman for reading the many drafts of this thesis and providing helpful comments. I would specially like to thank my wife Sue for sacrificing so much on her part for the sake of my education. Finally I would like to thank the Laboratory for Computer Science for allowing me to use its facilities in preparing this thesis.

Table of Contents

1. Introduction

The design methodology employed in the realization of an Information Processing system is driven to a large extent by the technology of the times. At present, almost all digital systems are implemented using a *Synchronous* design methodology. Synchronous systems have been popular in the past due to the low level of integration available for logic devices. However, these systems will not be as attractive in the future if they are realized using VLSI (Very Large Scale Integration).

Before the invention of the transistor, vacuum tubes were used primarily in the realization of logic-elements. Due to the poor reliability and high cost of vacuum tubes, the most appropriate design methodology was the one that used the fewest logic-elements. Although the reliability of the logic devices is greatly improved in present LSI (Large Scale Integration) systems, the number of logic devices still contributes significantly to the cost of the system.

Presently, integrated circuit technology is experiencing an exponential growth in the number of *transistors* that can be put on a single silicon chip [10]. With such an explosive growth rate, the cost of the logic-elements will not be the significant consideration in the realization of a system. System reliability and performance, on the other hand, will play a more important role in choosing between the synchronous and asynchronous approaches to system realization.

This thesis presents a design methodology for self-timed systems. The term *Self-timed* is used to denote systems whose operation is not synchronized to any reference timing source. One of the earliest forms of digital self-timed systems can be found in the relay-logic of the pre-electronic era [15]. There have been a number of other self-timed design methodologies proposed [16,11,1,4,8], but these have either not been complete, or the realization of a system using these methodologies has not been straightforward.

The methodology proposed in this thesis will present a few basic self-timed modules from which it will be possible to implement an arbitrary system. The choice of modules has been influenced by familiar programming constructs such as conditionals, iterations, sequencing,forking, synchronizing, etc. The philosophy behind the design methodology is to present a set of self-timed modules which can be used as building blocks for implementing systems. The methodology is the rules by which more complicated building blocks can be constructed from the set of self-timed modules and more primitive building blocks.

1.1 Requirements for the Realization of a System

In designing any system, it is always useful to consider the design process as a hierarchy of levels. The design is first specified at the highest level in the hierarchy. Each level in the hierarchy abstracts away the details that are not relevant to that level. To complete the design process, one must translate the design from the highest level in the hierarchy to the next lower level, and so on till we are at the lowest level. The lowest level in the design hierarchy must necessarily include every aspect of the physical realization of the system. For example, Figure 1.1 shows a possible hierarchy of design for a system realized in VLSI.

The top level in the design hierarchy will generally be an algorithmic representation of the system. At this level of the design, the notion of the *time metric* has been abstracted away completely. The algorithmic representation of the system specifies the sequence constraints for the operations of the system. A Petri-net [12] can be considered as an abstraction of the design of a system at the algorithmic level.

The algorithmic design must eventually be translated down to the lowest level in the design hierarchy. At the lowest level the sequence constraints of the system must necessarily be bound to the *time metric*. It is essential that the sequence constraints of the system are not violated under any normal operating conditions in the realization of the system.

There are two basic approaches to realizing a digital system, these are: 1) the synchronous approach to system realization, and 2) the self-timed or asynchronous approach to system realization. These design approaches differ in the way they maintain the sequence constraints of the system and how they bind these constraints to the *time metric*. The next two sections present a brief description of these two approaches to system realization.

1.2 Synchronous Approach to System Realization

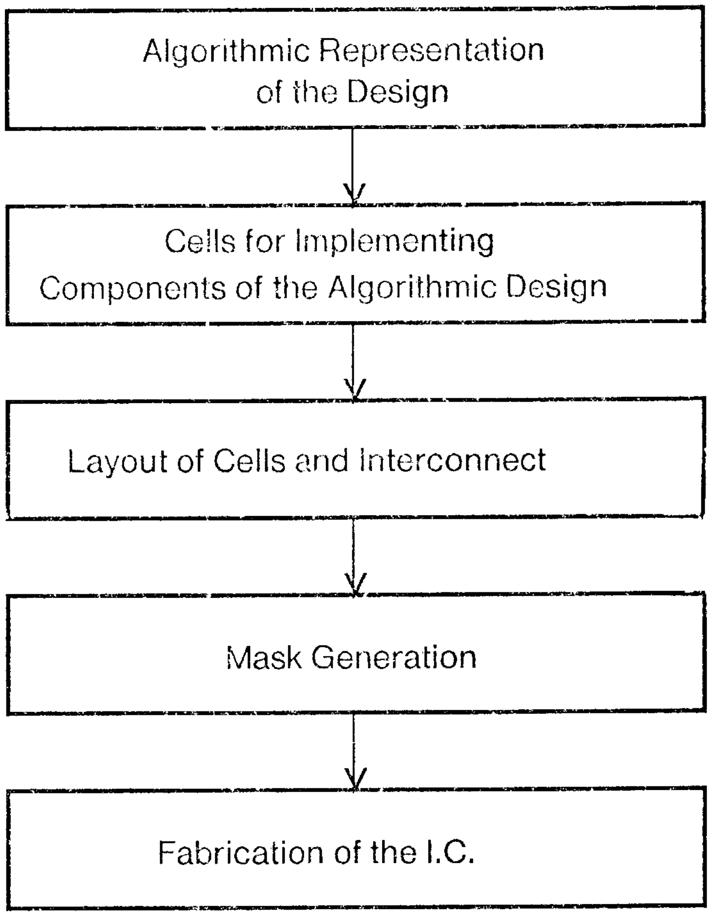

Algorithmic Representation of the Design
Cells for Implementing Components of the Algorithmic Design
Layout of Cells and Interconnect
Mask Generation
Fabrication of the I.C.

Figure 1.1 Hierarchical Design in VLSI

Synchronous systems have the common characteristic that their operations are synchronized to a system clock. The clock in most synchronous systems is either *single-phase* or *two-phase*. The single-phase clock is a single periodic waveform, while the two-phase clock consists of two independent periodic waveforms that do not overlap. The basic requirement for the design of a synchronous system is that all feedback paths must be broken with clocked storage elements.

The simplest form of a synchronous system is a synchronous sequential machine. There are two important varieties of these: the single-phase clocked synchronous sequential machine, and the two-phase clocked synchronous sequential machine. Figure 1.2a shows a single-phase synchronous sequential machine. Such a machine has a set of zero or more inputs, outputs, and feedback paths. The feedback paths are used to store the state information of the machine. If the machine does not have any feedback paths, it is completely combinatorial. In general, the machine consists of a set of registers that are clocked with the single-phase clock, and combinatorial logic. The outputs of the machine can be obtained from two sources; they can either be a subset of the present state variables, or they can be a function of the present state and the present inputs.

A single period of the clock defines the basic unit of operation of a single-phase synchronous sequential machine. Figure 1.2b shows a single cycle of operation for a single-phase synchronous sequential machine, and the timing requirements for the clock. When the clock becomes *true* the next state information is copied from the input of the registers. The new present state and the new inputs to the machine define the new next state of the machine. The *true* level of the clock must be long enough to load the outputs of the registers from their inputs. Also, the true level of the clock must not be so long that the new next state propagates to the inputs of the registers. The entire clock period must be long enough to allow the next state variables at the output of the combinatorial logic to be valid before the next cycle of the clock begins. Also, if the registers are dynamic, the clock period must be less than the refresh period of the dynamic logic. Therefore, there is a two-sided timing requirement on both the true level of the clock, and the clock period itself for the sequential machine to function correctly (Figure 1.2b). In general, it may not be possible to satisfy these timing requirements, and a system built using the single-phase synchronous scheme will not be reliable with variations in circuit parameters. Many existing synchronous systems give the illusion of being single-phase synchronous systems

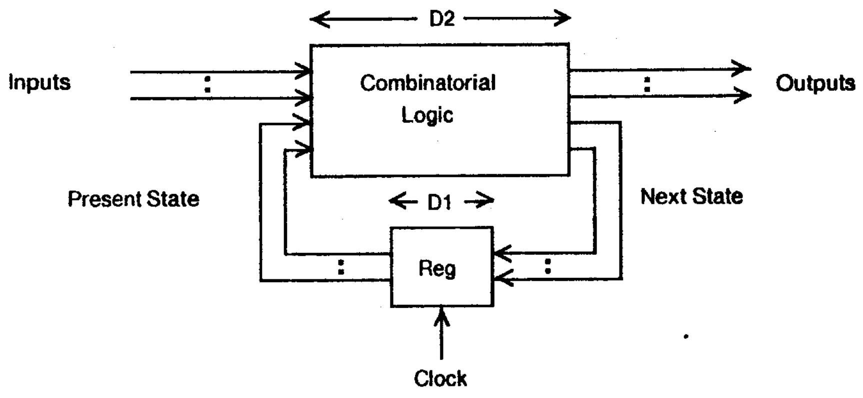
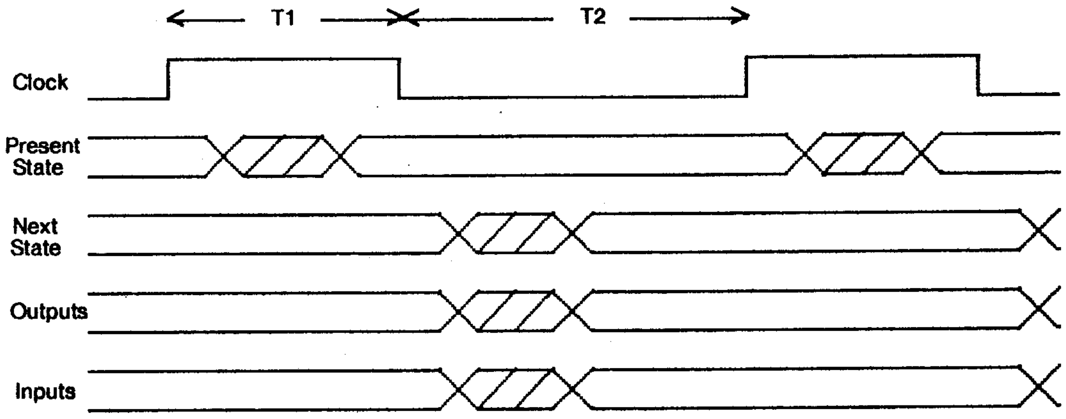


Figure 1.2a Single Phase Synchronous Sequential Machine



1) Max(D1) < T1 < Min(D1) + Min(D2)

2) Max(D1) + Max(D2) < T1 + T2 < Refresh Period

3) 'D1' is the register delay while the clock is true

4) 'D2' is the combinatorial delay

Figure 1.2b Timing Diagram for a Single Clock Period

because they have a single system wide clock. In actuality, the clocked logic devices in these systems convert the single phase system clock into two non-overlapping local clocks.

The correct behavior of the machine is guaranteed by satisfying the timing relationships mentioned above, and by correctly programming the combinatorial logic. The sequence constraints are bound to time-metric by the single-phase clock. The rising edge of the clock defines the transition of the system from one state to the next; and the actual amount of time that the system is in a particular state is equal to the period of the clock. Single-phase synchronous systems are attractive in that a single periodic waveform is used both to define transitions from one state to another, and also to bind the state of the system to the time-metric.

Figure 1.3a shows a two-phase synchronous sequential machine. It is similar to the single-phase synchronous sequential machine except that it has two periodic non-overlapping clocks and two independent register sets that are clocked with opposite phases of the clock. As a result of the two register sets, the two-phase synchronous sequential machine has the additional *intermediate state* variables. In the two-phase synchronous systems there is never any closed feedback path due to the non-overlapping constraints on the clocks. Figure 1.3b shows the operation of a two-phase synchronous sequential machine for a single clock cycle and the timing requirements for the clock. This scheme provides for high reliability since the timing constraints on the clock are completely one-sided, and hence guaranteed to be satisfiable, if the registers are not dynamic storage elements. If the registers are dynamic storage elements, there is a two-sided timing requirement on the clock period. The period of the clock labeled 'T4' in figure 1.3b acts as a buffer to accommodate clock skew in a two-phase synchronous system. Due to the one-sided timing requirements for the clock period (for static registers), 'T4' can be made sufficiently long to ensure correct operation of the system for a given clock skew. Single-phase synchronous systems, on the other hand, do not have the freedom to increase the clock period to reduce their sensitivity to clock skew. Most synchronous systems are built using the two-phase clocking scheme because of the increased reliability compared to the single-phase clocking scheme, or due to the inability to satisfy the two sided timing relationships that are required for the single-phase clocking scheme. The price for the increased freedom in two phase clocked systems is a reduction in performance. The clock period in a two-phase synchronous system must be greater than the clock period in a single-phase synchronous system by the time required to load the 'intermediate state' registers.

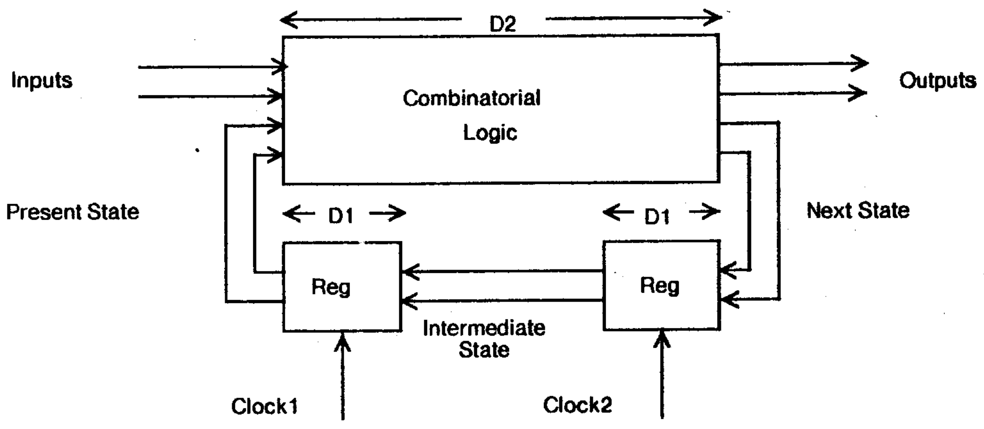
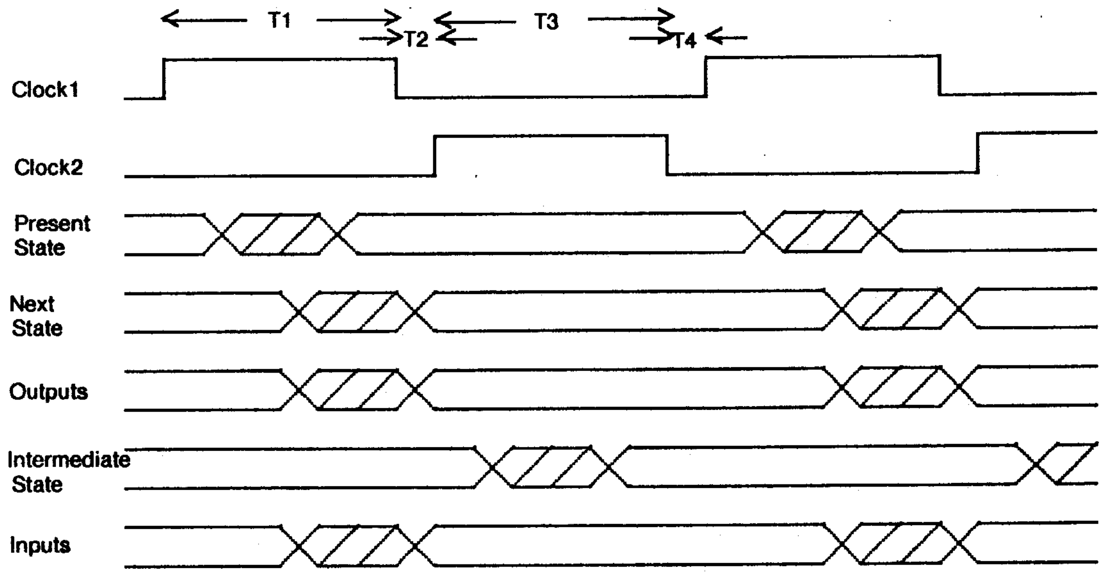


Figure 1.3a Two Phase Synchronous Sequential Machine



1) Max(D1) < T1 2) Max(D1) < T3

3) Max(D1) + Max(D2) < T1 + T2

4) T1 + T2 + T3 + T4 < Refresh Period

5) 'D1' is the register delay while the clock is true

6) 'D2' is the combinatorial delay

Figure 1.3b Timing Diagram for a Single Clock Period

The correct behavior of a two-phase synchronous system is also guaranteed by meeting the timing requirements on the clocks and by correctly programming the combinatorial logic. The rising edge of the first phase of the clock defines the transition of the system from one state to the next. The states are bound to the time metric by the clock period. This scheme of synchronous logic is attractive due to its high reliability and the way in which two non-overlapping clocks are used to define the state transitions and to bind these state transitions to the time metric.

1.2.1 Interconnection of Synchronous Sequential Machines

It is possible to realize an arbitrary system using only a single synchronous sequential machine, but such a methodology would complicate the understanding and design of the system. Also, such an approach inherently restricts the parallelism in a system since it does not allow two or more activities to execute simultaneously. It is more economical to structurally decompose the entire system into an interconnection of smaller subsystems. The exact degree to which the system should be decomposed into smaller subsystems is an engineering decision that depends on the objectives of the design. These objectives can include trade-offs between minimizing the area of the system, minimizing the number of devices in the system, maximizing the speed of the system, etc. These subsystems can each be realized as a single synchronous sequential machine. It is allowable to connect a set of synchronous sequential machines in a cycle if the outputs of each machine in the cycle are a subset of their present state variables. If the outputs of each synchronous machine are a function of their present state and their present inputs, then it is not allowable to connect a series of such machines in a cycle.

The wiring delays associated with the interconnect of the sequential machines are extremely important in designing such a system. In designing the individual sequential machines the timing constraints to be satisfied were only local to the registers and the combinatorial logic of the sequential machine. In building a system that consists of a network of interconnected sequential machines, the timing constraints are no longer local. If it is assumed that the interconnect delays between the sequential machines are zero, then any network of sequential machines will function correctly as long as their topological interconnection is correct, and each sequential machine functions correctly. A *path* in a synchronous system is defined to be a sequence of wires and combinatorial logic, where the start and end of the path are either register storage elements or inputs/outputs of the system. Also, the signal at the end of the path must be dependent on the signal at the start of the path. If the interconnect delays between the sequential machines are are not zero, the additional timing constraints to be satisfied are that all signal paths must have a delay less than a single clock period. If the clock skew between all interconnected sequential machines is not zero, the maximum value for the clock skew between any two interconnected sequential machines must also be included in the timing requirements mentioned above. If the two-phase synchronous clocking scheme is used, it is further assumed that the the registers at the two ends of every path will be clocked with opposite phases of the clock. The correct operation of the entire system is then guaranteed by satisfying the timing requirements for each individual machine, the timing requirements for all paths in the system and by correctly interconnecting these sequential machines.

The state of such a system is a function of the state variables of all the synchronous sequential machines in the system. The transitions of the periodic global clock define the transitions in the sequence domain of the state of each sequential machine, which in turn define the transitions in the state of the entire system. The clock period also binds the duration of a state of the system to the time metric.

1.3 Self-timed Approach to System Realization

Self-timed systems are characterized by the absence of any global clock to which all operations are synchronized. The basic building block of self-timed systems is the self-timed module. This is analogous to the synchronous sequential machine of synchronous design methodologies. The operation of the self-timed modules themselves is also not synchronized to any local clock. A basic requirement for self-timed systems is that they must operate correctly independent of the delays in the interconnect wires between the modules of the system. The wires that connect self-timed modules are called channels.

Figure 1.4 shows the general structure of a self-timed module. At its terminals, it consists of a set of 0 or more input and output ports. In order for a self-timed system to function correctly independent of the channel delays, the modules must obey an asynchronous communication protocol. This has been commonly referred to as *handshaking*. Data flow in a channel is required to be from a source module to a destination module, but the synchronization signals will in general be bi-directional on a channel.

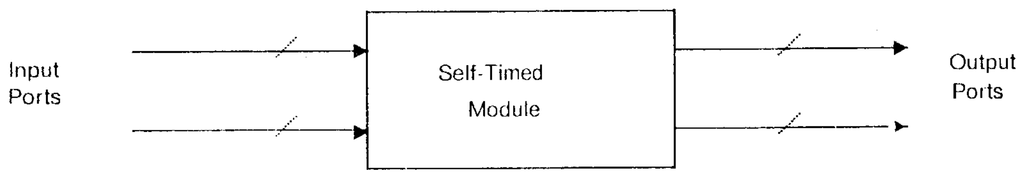


Figure 1.4a Terminal Structure of a Self-Timed Module

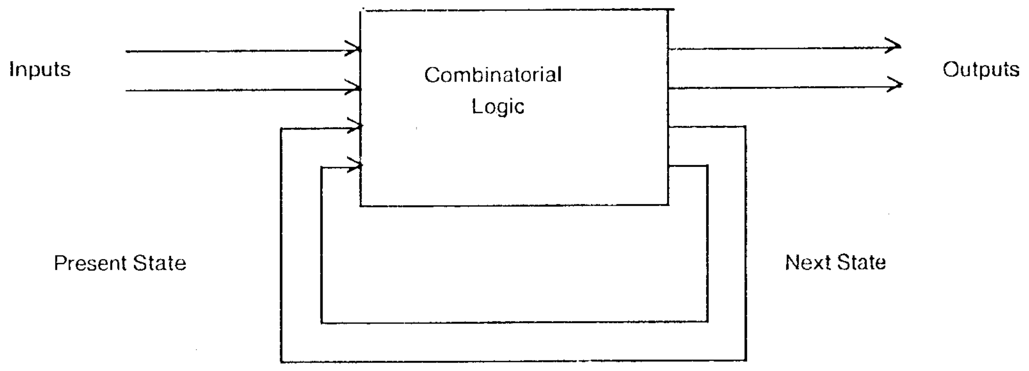


Figure 1.4b Asynchronous Sequential Machine

The interior of the self-timed modules will in general consist of an asynchronous sequential machine. The inputs to this asynchronous sequential machine are the data signals of the input ports and the synchronization signals of the output ports. Similarly, the outputs of the asynchronous sequential machine are the data signals of the output ports and the synchronization signals of the input ports. The asynchronous sequential machine itself consists of combinatorial logic and feedback paths.

The present state variables of the asynchronous sequential machine are the next state variables delayed by the delays of the feedback wires. As in the synchronous case, the outputs of the asynchronous sequential machine can be obtained from two sources; the outputs can either be a subset of the present state variables of the system, or they can be a function of the present state variables and the inputs to the module. If the outputs are a subset of the present state variables of the asynchronous sequential machine, then they can be guaranteed to follow the communication protocol. If the outputs are a function of the present state and the inputs to the system, then the terminal behavior will be dependent on the delays of the combinatorial logic of the module. The designer will have to guarantee the correct behavior at the outputs in this case by carefully analyzing the delays in the interconnect and logic elements of the module.

To guarantee the correct behavior of a self-timed module it is allowable to make assumptions about the delays internal to the module. But it is the task of the designer to ensure that these assumptions are valid under all normal operating conditions. The motivation in dividing a self-timed system into self-timed modules is to make the timing constraints local to the interiors of the modules for the correct operation of the system. This is in contrast to the synchronous design methodologies where the timing constraints to be satisfied are global to the entire system.

The simplest form of a self-timed system is a single self-timed module. The correct operation of the system is guaranteed by ensuring that the module follows the communication protocol under all normal operating conditions and by correctly programming the asynchronous sequential machine internal to the module. The state of the system is a function of the state variables in the asynchronous sequential machine of the module. The transitions from one state to another of the system are bound to the transitions of the state of the asynchronous sequential machine. These transitions are dependent only on the delays internal to the module, and the delays in the input and output channels connected to the module.

It is possible to implement an arbitrary system using only a single self-timed module. But as in the case of a single synchronous sequential machine, such a methodology would complicate the design and understanding of the system. In addition, the designer of the system will have to meet timing constraints that are global to the entire system. This would lose the primary advantage of self-timed systems over synchronous systems. It would be more appropriate to structurally decompose the system into the interconnection of a number of self-timed modules. The exact degree to which a system is broken up into self-timed modules is an engineering decision depending on the objectives of the design.

The correct operation of such a system is guaranteed by guaranteeing the correct functional operation of each module in the system and by interconnecting these modules correctly. The state of such a system is a function of the state of all the modules in the system. The transitions of the state of the system correspond to the transitions of the state of any module in the system. The state transitions do not occur at discrete time periods; in general, the time between state transitions can be arbitrary.

1.4 System Realization in VLSI

The most explosive growth in electronics has been in the VLSI technology. This has been both in the number of devices that can be built on a single silicon chip, and also in the reduction of the switching delays of the devices. The complexity of VLSI devices approximately doubles every year [9]. This trend has been going on since the early 1970's, and is expected to go on well into the mid-1980's before decreasing. It is expected that by the end of 1980 it will be possible to have 100,000 devices on a single silicon chip. The switching time of the elementary devices has also been steadily decreasing. Based on past trends, it is expected that the switching time of the most elementary devices will be about 0.2 nanoseconds by the end of 1980 [10].

The self-timed design methodology proposed in this thesis is motivated in part by the rapid advances that have been made, and that are expected to be made in the future in VLSI. Due to the availability and low cost of VLSI, it is appropriate to come up with a design methodology applicable to this technology. As will be seen shortly, synchronous systems are not well suited for the high density VLSI technology. In the past, the VLSI technology had been profitable only for applications with extremely high volume. Now however, the process for realizing a VLSI circuit has been split up into a number of disjoint activities. These are: 1) logic design, 2) mask generation, and 3) fabrication. This has helped reduce the cost of realizing VLSI circuits for the low volume user. For example, it is possible to put 15 separate designs on a single silicon wafer [6]. The cost for generating the masks for a single wafer are about 7,000 dollars, and about 2,000 dollars for processing a batch of about 15 such wafers [6]. With a total cost of about 10,000 dollars, the cost per design can be reduced to under 1,000 dollars.

The relationships between the parameters of a device, and the physical geometries of the device are extremely important as the geometries are scaled down. For example, as the feature sizes of the devices are reduced, the resistance per unit length of the wires increases by the square of the reduction. The capacitance per unit length, however, remains constant [14]. Also, the switching times of the elementary devices decreases linearly. In analyzing the delays in a system it is important to consider both the delays of the switching elements and the delays in the interconnect wires. The wires in Silicon Gate nMOS devices act like transmission lines, which is similar to other high density technologies. The delay in a transmission line is proportional to the product of: 1)the square of the length of the line, 2) the capacitance per unit length, and 3) the resistance per unit length. If device geometries are reduced by a factor of 10, then the switching times of the devices can be expected to reduce by a factor of 10 also. Since the capacitance per unit length remains constant, and the resistance per unit length increases by the square, the delay in sending a signal the same distance goes up quadratically also. This means that the delay in sending a signal from one part of the chip to another will go up by a factor of 100. More importantly, the ratio of the wire delays to the switching times of the devices goes up by the cube, or 1000 in this case.

If a synchronous system is implemented on a single chip, the clock period must be long enough to allow a signal to be transmitted from any one part of the system to any other. There is no incentive to localize the communication in a synchronous system. Even if a pipelined architecture is used, the clock must be distributed to all the pipelined elements. The clock in a synchronous system can be considered as a signal from every element in the system to every other element. The information that the clock carries is the mutual consent of all the elements as to when they will change their state. The efficiency of a synchronous system is defined to be the ratio of the switching delays of the elementary devices to the clock period. For a synchronous system to be efficient, the clock period must be a few multiples of the switching time of the devices [14]. As mentioned earlier, the ratio of the wire delay to the switching time of the devices increases cubically with the reduction in the geometries. If feature sizes are reduced by a factor of 10, the efficiency of a synchronous system communicating over the same distance will go down by a factor of 1000. This will clearly be an unacceptable design methodology in the future.

Any design methodology that successfully takes advantage of VLSI must allow for, and push towards the localization of communication. The self-timed approach to system realization seems most appropriate at meeting these objectives. Synchronous systems will not be able to fully exploit the VLSI technology due to their poor efficiency and reliability considerations. To increase efficiency, a synchronous system can be realized as a collection of independently clocked communicating synchronous machines. Each synchronous machine in such a system will require logic to synchronize its external asynchronous inputs. The probability of synchronization failures in any synchronizing scheme can never be reduced to zero [2]. Self-timed systems are not subject to synchronization failures since the modules in these systems function independent of any reference timing source. From a performance standpoint self-timed systems have the advantage that their operations are limited by the average delays of the system, while the performance of synchronous systems are limited by the worst case delays.

1.5 Synopsis of Thesis

Chapter 2 gives the formal definition of the self-timed design methodology proposed in this thesis. The model of a self-timed module is given, along with the communication protocol between modules. Chapter 2 also defines the behavior of the various module types in a self-timed system. Also, the assumptions made on the underlying technology used to realize the systems are defined. Chapter 3 gives the general design methodology for realizing control modules in a self-timed system. The chapter concludes by presenting the design of the control modules defined in the methodology of this thesis. Chapter 4 is similar to Chapter 3, except that it gives a general design methodology for realizing data transformation modules in self-timed systems. Chapter 5 contrasts the design of a 2X2 router using a synchronous design methodology, and the methodology presented in this thesis. Finally, chapter 6 gives a summary of the methodology and the areas for further improvement and research.

2. Definition of Self-Timed Systems

In the design methodology proposed in this thesis, every self-timed system is a network of self-timed modules. A self-timed module is a finite state machine with a set of input and output ports. Figure 2.1 shows the general structure of a self-timed system in terms of self-timed modules. The modules of a self-timed system fall into one of the following classes: 1) control modules, or 2) 'user' modules. The control modules are used to perform the data routing functions, while the 'user' modules are used to perform the data transformation functions in a self-timed system. Self-timed modules are restricted to communicate with other modules via channels. A channel is a bundle of wires that connects the output port of a module to an input port of some other module. The channels of a self-timed system are assumed to have unknown delays. One of the primary goals of this design methodology is to define self-timed systems whose behavior is independent of the delays of the interconnect. However, if a self-timed system has the potential for deadlocking then its behavior can be influenced by the channel delays of the system. In certain cases it may be possible to avoid or cause deadlocks with certain values for the channel delays.

It is important to make the distinction between self-timed modules and a self-timed system.The simplest form of a self-timed system is a self-timed module. If two self-timed modules are connected, then their connection defines a new self-timed system. Self-timed modules form the atomic building blocks of self-timed systems. It is not possible to combine two self-timed modules and create a larger self-timed module. Conversely, it not possible to decompose a self-timed module into a connection of two or more smaller self-timed modules.

In the proposed design methodology a self-timed system consists of a single network of interconnected modules. The is no separation of the control and data flow sections of a self-timed system. The data flow and control flow sections of a self-timed system correspond to the data flow and control flow sections of every channel and module of the system.

All modules in a self-timed system fall into one of the categories listed below. The input/output behavior of these modules will be given in a later section of this Chapter, but informally, their behavior closely corresponds to their names.

1) user
2) conditional
3) iteration

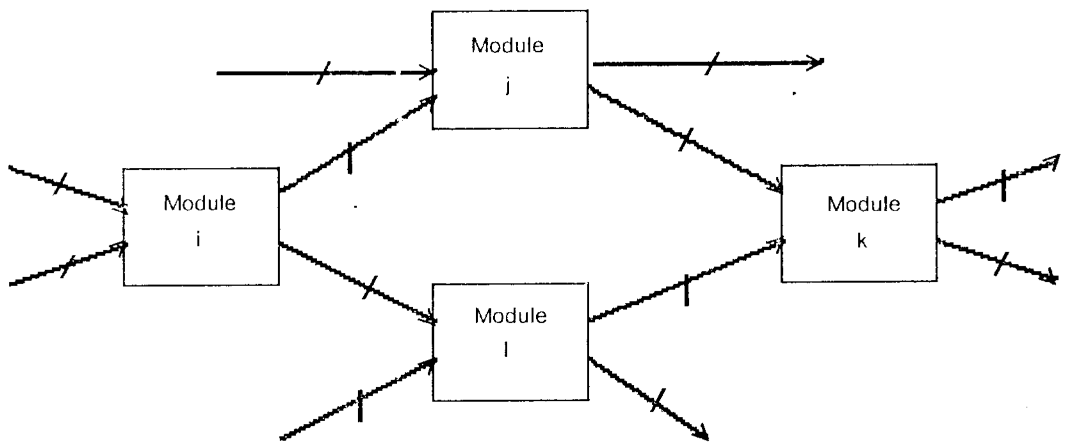


Figure 2.1 Structure of a Self-Timed System

4) register
5) union
6) arbitration
7) fork
8) join
9) predicate
10) source
11) sink
12) switch

Modules of type 2 through 12 are called the control modules, these modules route the data at their input ports to the output ports according to the defined sequencing constraint for the particular control module. The functionality of the control modules is fixed and cannot be changed by the designer of the system. The only exception to this is the 'predicate' module, where the predicate is specified by the designer. The only parameter at the control of the designer is the width of the data paths at the input and output ports of the control modules. The functionality of the 'user' modules is under the complete control of the designer.

2.1 Communication Protocol

The communication channels of a self-timed system transmit: 1) data signals, and 2) a synchronization signal. Note that the signals on a channel do not directly correspond to wires. Each data signal is physically transmitted on a pair of wires, and each synchronization signal is physically transmitted on a single wire. The data signals are directed from the source module to the destination module of the channel, while the synchronization signal is directed from the destination module to the source module of the channel. A channel in a self-timed system can be used to transmit an arbitrary (but finite) number of values at one time.

Every channel in the self-timed system must transmit one synchronization signal. The only exception to this is the predicate channel, which has two synchronization signals. In addition, every channel must transmit at least one data signal. Communication over each channel is assumed to be unbuffered, that is, direct source and destination module synchronization is required for information transfer over a channel.

All data signals in a self-timed system are encoded using the dual rail signaling protocol. Figure 2.2a shows the interpretation of a signal using the dual rail signalling protocol. A signal is transmitted on a pair of wires, *line1* and *line2*. Each wire can be in one of two states, either true or false. Throughout this thesis, the symbol "1" will be used to represent true, and the symbol "0" to represent false. Figure 2.2b shows the *signal constraint* for each data signal on a channel. Once the signal is in a "0" or "1" state, it must pass through the undefined "-" state before changing to a new value of "1" or "0." Under normal operation, the signal should never be in the defined but unknown "?" state.

The communication protocol chosen for this thesis is the *reset signaling* protocol. Figure 2.3a shows the sequence constraints for a single transaction on a channel with this protocol. This is defined as the *channel constraint* of a self-timed system. A single *transaction* on a channel is defined to be the sequence of actions on a channel by which a set of data is transferred from the source to the destination module of the channel. If the synchronization signal of a channel is false, it indicates that the destination module of the channel is waiting for a new set of data from the source module. However, if the synchronization signal of a channel is true, it indicates that the destination module of the channel has accepted the current data on the channel. The synchronization signal of the channel is called the *acknowledge* signal. As mentioned earlier, the only exception to this is the predicate channel, which has two synchronization signals. The synchronization signals of the predicate channel are labeled "t" and "f." For a given channel transaction of the predicate channel, only one of the two acknowledge signals goes through the reset signaling protocol shown in figure 2.3a. The request signal is not explicitly transmitted on a channel but is generated at the input port of the destination module of the channel from the data signals. Figure 2.3b shows the sequence constraints between the data signals of a port and the request signal. A channel is defined to be *reset* if all the data signals of a channel are in the undefined "-" state and the acknowledge signal of the channel is "0."

2.2 Input/Output Behavior of Self-Timed Modules

This section will present the input/output behavioral for the 12 module types mentioned in the previous section. Examples of the use of the various modules can be found in Chapter 5 which presents a design example for a 2X2 router. The only exceptions are the 'iteration,' 'source,' and 'sink' modules for which simple design examples will be included. The input/output behavior of each module type will be given for a single *module transaction.* A single module transaction

Line1	Line2	Signal Definition
0	0	Undefined '-'
0	1	Zero '0'
1	0	One '1'
1	1	Defined but Unknown '?'

Figure 2.2a Dual-Rail Signaling Protocol

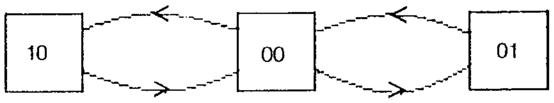


Figure 2.2b 'Signal Constraint' for Dual-Rail Data

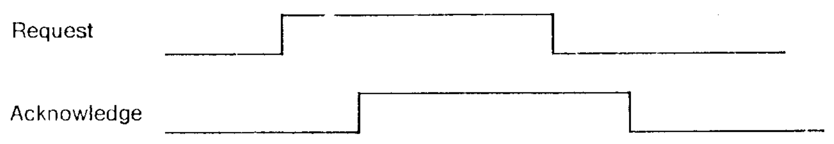
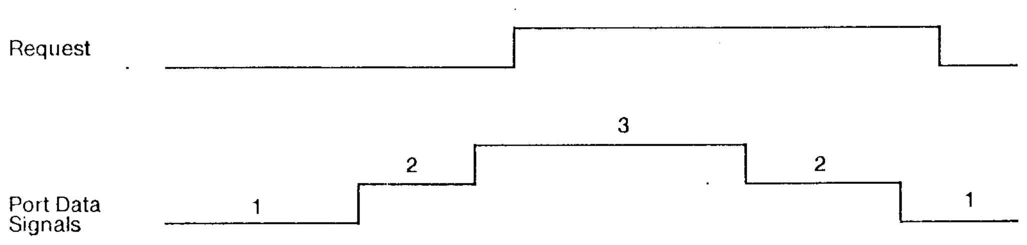


Figure 2.3a 'Reset Signaling' Protocol



1) All Signals Undefined

2) Some Signals Defined

3) All Signals Defined

Figure 2.3b Request Signal Generation for an Input Port

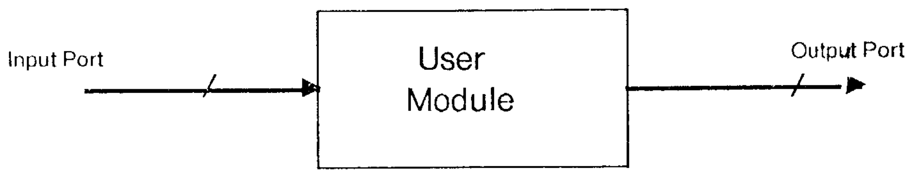
corresponds to a single transaction on one or more input ports of the module followed by a single transaction on one or more output ports of the module. The only exception to this is the 'iteration' module, where a single transaction at the input port can lead to 0 or more transactions on the iteration output port. A module is said to be reset if every input and output port of the module is reset. A single module transaction starts with the module in the reset state, and ends with the module in the reset state.

In presenting the logic symbols and timing constraints for the various modules the following conventions are observed.

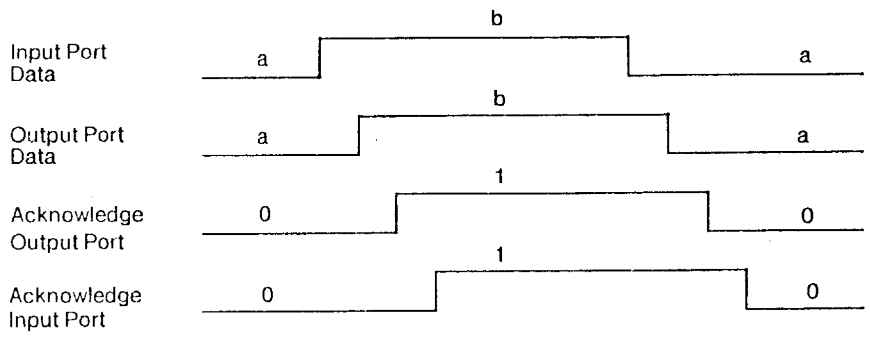
1) all channels are drawn as bold lines with a slash.
2) the data signals of a channel are drawn as bold lines without a slash.
3) the acknowledge signals of a channel are drawn as normal lines.
4) all transitions of the data signals of a channel from the completely undefined to the completely defined state (and vice-versa) are assumed to pass through the unlabeled intermediate state where some data signals are defined and others are undefined.
5) if the signal transitions are shown to occur at the same time then their order can be arbitrary.

The 'user' modules have exactly one input port and one output port. The number of data signals on the input and output ports of a 'user' module are arbitrary (but not 0). A single module transaction for the 'user' module is given in Figure 2.4. The data flow section of a 'user' module defines the data at the output ports as a combinatorial function of the data at the input port. The 'user' modules can be used in conjunction with the 'register' modules to build an arbitrary finite state sequential machine, or they can be used by themselves in implementing arbitrary data transformation functions.

The 'conditional' module is used to conditionally transfer control to one of two subsystems in a self-timed system. The module has one input port and three output ports. Figure 2.5 gives the terminal structure of a conditional module, and the sequence constraints between the input and output ports. Every data signal at the input port must be connectable to at least one of the output ports. After the input port gets defined, the predicate port data signals are defined with a subset of the data of the input port. Only one of the two acknowledge signals of the predicate port should now change to "1." If the "t" acknowledge becomes a "1," a subset of the data at the input port is switched to the data at the "true" output port of the module. Similarly, if the "f" acknowledge signal becomes a "1," a subset of the data at the input port is switched to the data at the "false" output port. Once the acknowledge signal from the output port that was selected becomes true,



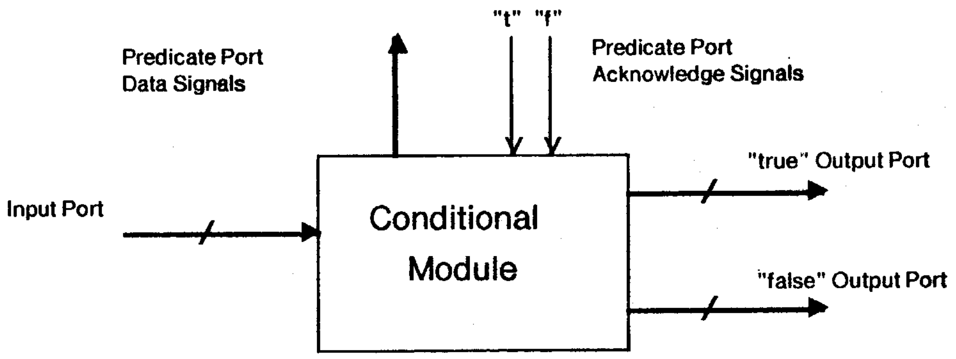
(a) 'User' module logic symbol



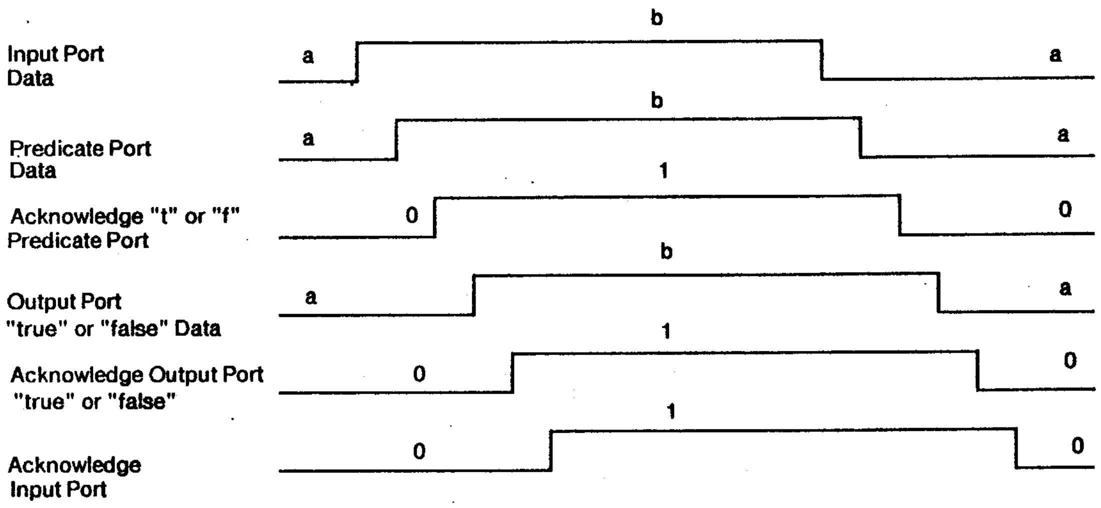
1) "a" - All Data Signals Undefined
2) "b" - All Data Signals Defined
3) "0" - Signal False
4) "1" - Signal True

(b) Input/output behavior of the 'User' module

Figure 2.4



(a) 'Conditional' module logic symbol



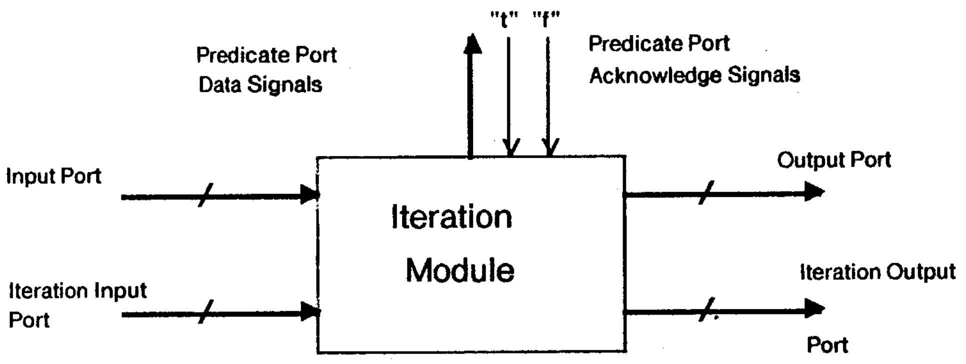
1) "a" - All Data Signals Undefined
2) "b" - All Data Signals Defined
3) "0" - Signal False
4) "1" - Signal True

(b) Input/output behavior of the 'Conditional' module

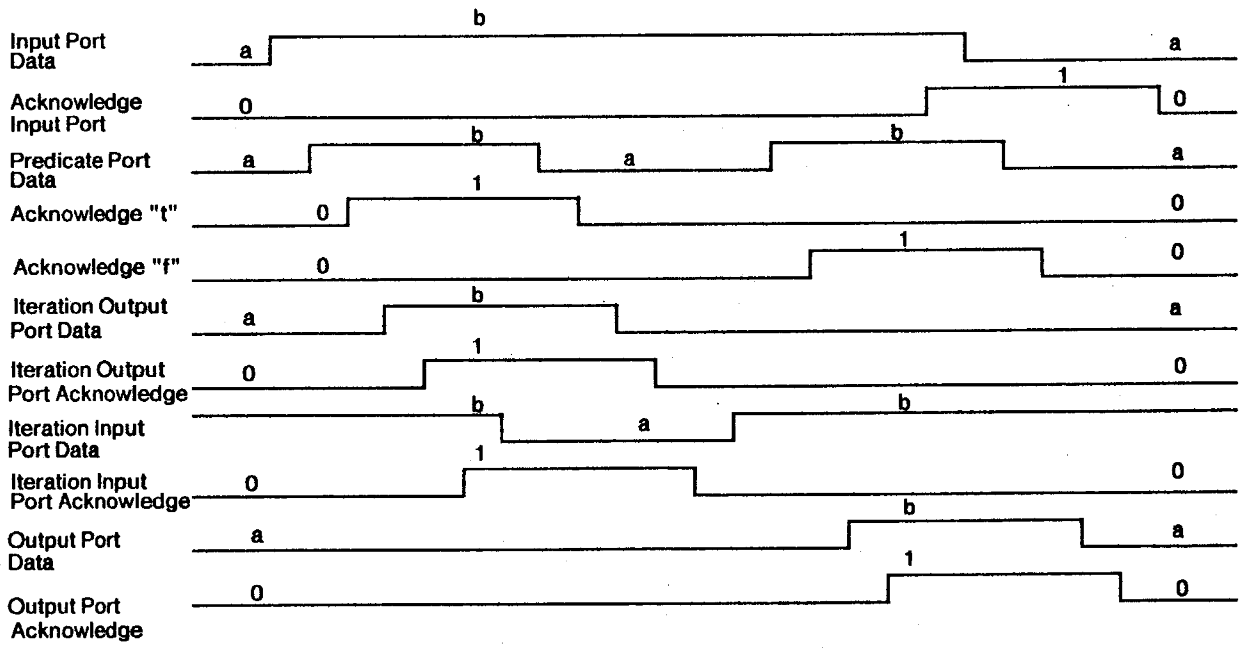
Figure 2.5

the acknowledge to the input port is made true. The input port then resets, followed by the resetting of the predicate port and the resetting of the "true" or "false" output port that was selected by the predicate.

The 'iteration' module is used to control the repeated execution of a subsystem in a self-timed system. The module has two input ports and three output ports. Figure 2.6 shows the terminal structure of the iteration module along with the sequence constraints between the input and output ports. Every data signal of both input ports must be connectable to at least one of the output ports. In the figure the body of the iteration is executed once. Once the input port becomes defined, the predicate port data signals are defined with a subset of the data of the input port. One of the two acknowledge signals of the predicate port must then change to "1." If the "f" acknowledge signal becomes true, then a subset of the data of the input port is connected to the output port. A single channel transaction now follows at the input and output port of the module. This is the case when the predicate is false and the iteration is terminated. If instead the acknowledge "t" signal becomes true, a subset of the data signals of the input port are switched to the iteration output port. A single transaction now occurs at the iteration output port of the module. This transaction corresponds to a single execution of the body of the iteration. The self-timed subsystem connected between the iteration output port and the iteration input port is the body of the iteration. When the acknowledge signal of the iteration output port becomes true, the predicate port data signals are reset. Note that the input port of the 'iteration' module is still defined. When the predicate port data signals are reset, the acknowledge "t" signal must reset. This results in the resetting of the data signals of the iteration output port. Once the acknowledge signal of the iteration output port becomes false, the data signals of the predicate port are defined with a subset of the data signals of the predicate input port. This results in the re-evaluation of the predicate. Note that the first time the data signals of the predicate port are defined with a subset of the data signals of the input port. If the predicate evaluates true, all subsequent re-evaluations of the predicate are performed with a subset of the data signals of the iteration input port. Similarly, if the predicate evaluates true the first time, the data of the iteration output port is defined with a subset of the data of the input port. However, on all subsequent true evaluations of the predicate the data at the iteration output port is defined with a subset of the data of the iteration input port. If the predicate evaluates false the first time, the data of the output port is defined with a subset of the data of the input port. If however the predicate evaluates false on the second or later re-evaluation, the data signals of the output port are defined with a subset of he



(a) 'Iteration' module logic symbol



1) "a" - All Data Signals Undefined

2) "b" - All Data Signals Defined

3) "0" - Signal False

4) "1" - Signal True

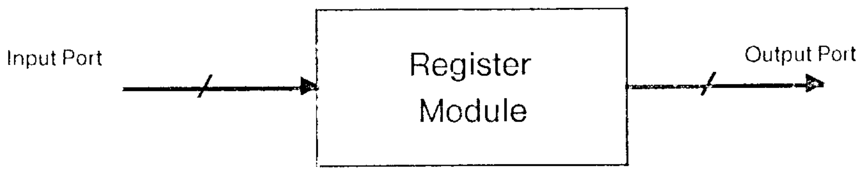
(b) Input/output bahavior of the 'Iteration' module

Figure 2.6

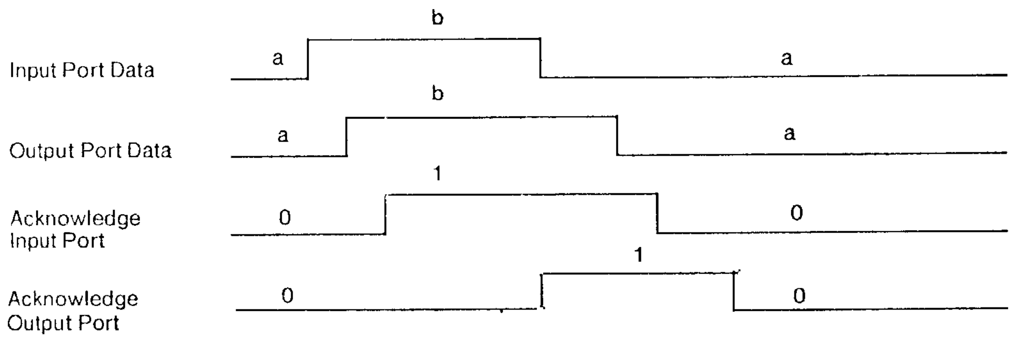
data signals of the iteration input port. Whenever the predicate evaluates false, a single transaction on the output port is followed by the resetting of the module.

As an example of the use of the 'iteration' module, consider a system that adds two non-negative integers. The input channel of the system receives both numbers to be added simultaneously. The input channel of the system can be connected to the input port of the 'iteration' module. The 'predicate' module connected to the 'iteration' module tests if the second integer is zero. The self-timed subsystem connected between the iteration output port and the iteration input port increments the first number and decrements the second. The output port of the 'iteration' module then delivers the correct sum of the two input integers.

The 'register' modules can be used to pipeline the operation of a self-timed system, or to build a sequential machine when used in conjunction with 'user' modules. These modules correspond to the registers of synchronous systems. The 'register' module has one input port and one output port. The number of data signals at the input port are the same as the number of data signals at the output port. Figure 2.7 shows the terminal structure of the 'register' module along with its timing constraints. When the input port of the 'register' module gets defined, the data signals of the port are stored in the buffer internal to the module. The data at the output port of the 'register' module is identical to the data at the input port of the module. The acknowledge line of the input port is set to true after the data has been stored in the 'register' module. The input port data signals can now change to the undefined state. The acknowledge of the input port is set to false only after the current data has been accepted by the module connected to the output port of the 'register' module. Once the data at the output of the 'register' module has been accepted, the buffers of the register module are all set to the undefined '-' state. This corresponds to a spacer state for the 'register' module. After storing the spacer in the buffers, the acknowledge of the input port is set to false. It is important to note that if a sequence of 'register' modules are connected, then no two adjacent modules can both store data. However, it is possible to have all the 'register' modules in the sequence store spacers. This corresponds to the sequence of 'register' modules being empty. If the sequence of 'register' modules is full, the information stored in these modules will alternate between data and spacers.



(a) 'Register' module logic symbol



1) "a" - All Data Signals Undefined

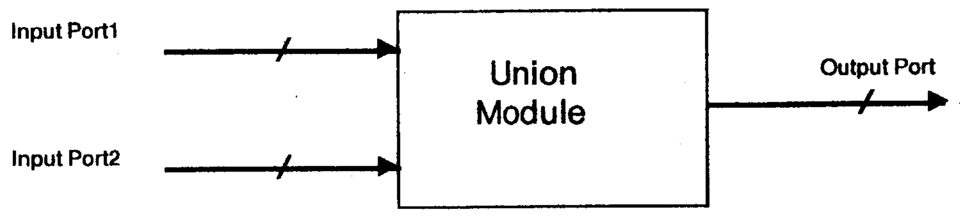
2) "b" - All Data Signals Defined

3) "0" - Signal False

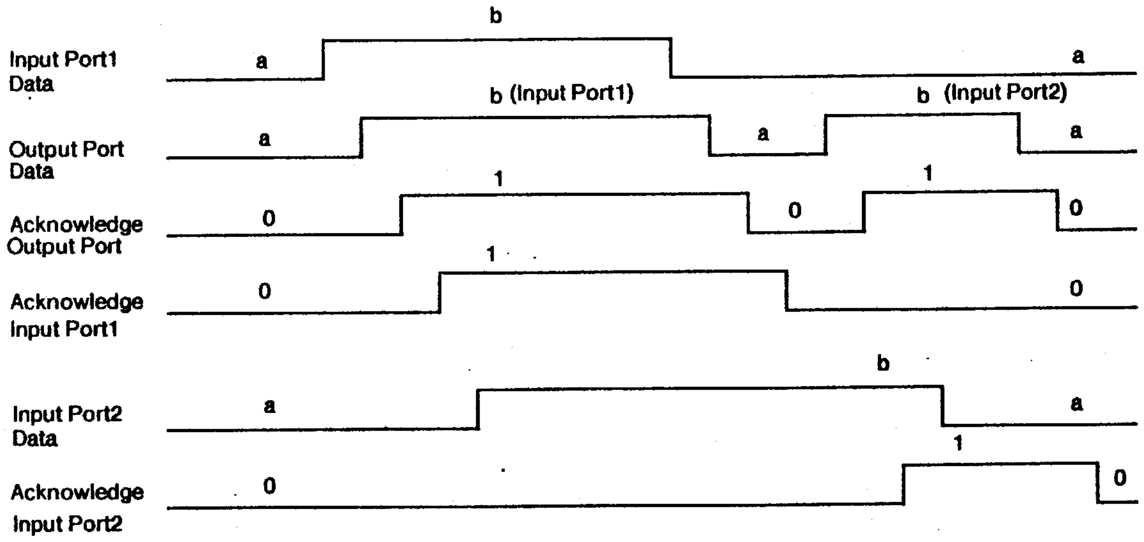
4) "1" - Signal True

(b) Input/output behavior of the 'Register' module

Figure 2.7



(a) 'Union' module logic symbol



(b) Input/output behavior of the 'Union' module

1) "a" - All Data Signals Undefined

2) "b" - All Data Signals Defined

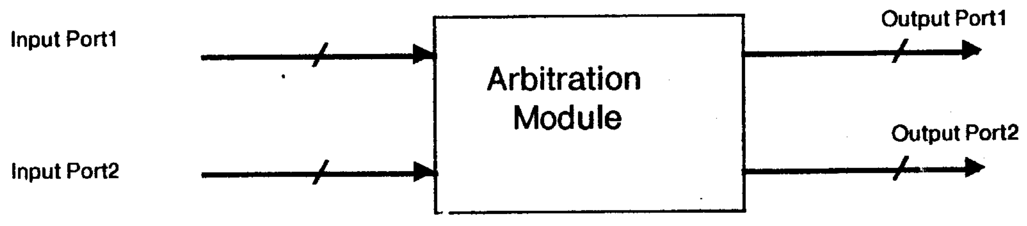
3) "0" - Signal False

4) "1" - Signal True

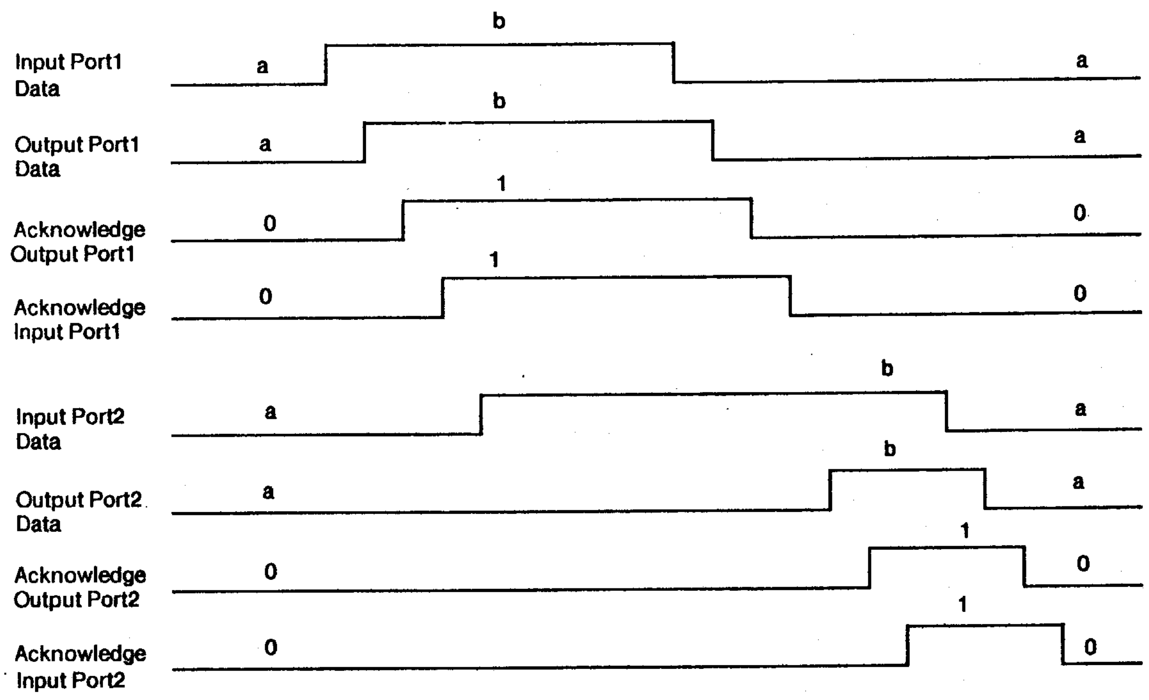
Figure 2.8

The 'union' module is used to share a hardware resource in a self-timed system. The 'union' module has two input ports and one output port. The number of data signals of all the ports of the module must be the same. Figure 2.8 gives the terminal structure of the 'union' module and its input/output timing constraints. In the figure, input port2 is initially blocked by a transaction of input port1. The inputs to the module are not assumed to be ordered, that is, a request to use the resource can come from both ports at arbitrarily close time intervals. If one input port gets defined while the other is reset, the output port of the 'union' module is switched to the input port that is defined. At this point the output port is considered granted to the defined input port. At this point a single module transaction, similar to the 'user' module, follows with the selected input and output ports. At the completion of this cycle the output port is considered free. If the second input port gets defined while the output port is not free, the transaction on the second input port is blocked. There is an arbiter internal to the 'union' module that guarantees the mutual exclusion of the output port to only one of the two input ports. If both input ports get defined at about the same time, the arbiter non-deterministically chooses one and grants the output port to it.

The 'arbitration' module is used to arbitrate between a pair of processes in a self-timed system. The two processes may or may not share the same hardware resources. The module is useful in preventing both the processes from being active at the same time. The processes are the subsystems connected to the output ports of the arbitration module. The 'arbitration' module has two input ports and two output ports. Each input port is associated with its own output port. The number of data signals on each input/output port pair must be the same, but the number of data signals of the two input or two output ports need not be the same. Figure 2.9 gives the terminal structure of the 'arbitration' module and its input/output timing constraints. In the figure, input port2 is initially blocked by a transaction of input port1. The input ports to the 'arbitration' module need not be ordered, the two input ports can request arbitration at about the same time. An arbiter internal to the module guarantees the mutual exclusion of the output ports to only one of the two input ports. When the request is granted to an input port, the output port associated with it is switched to the input port. From this point on, a single module transaction follows similar to the 'union' module where the input/output ports are the port pair selected by the arbiter. The input port not selected must wait for the entire module transaction to complete with the selected port before it can proceed.



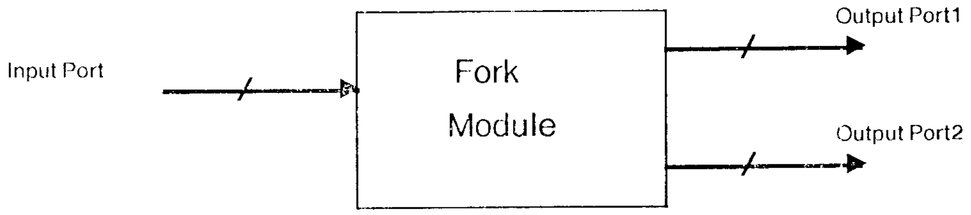
(a) 'Arbitration' module logic symbol



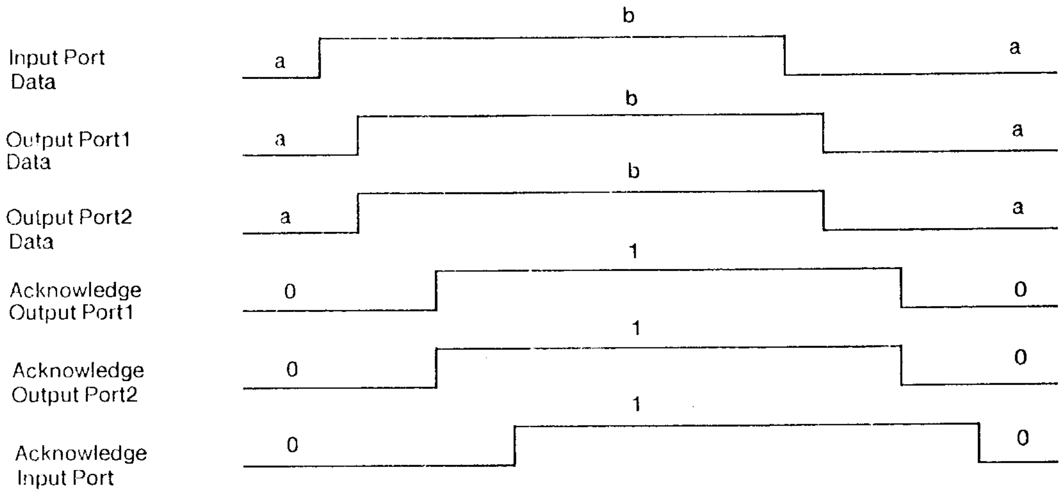
(b) Input/output behavior of the 'Arbitration' module

1) "a" - All Data Signals Undefined
2) "b" - All Data Signals Defined
3) "0" - Signal False
4) "1" - Signal True

Figure 2.9



(a) 'Fork' module logic symbol



1) "a" - All Data Signals Undefined
2) "b" - All Data Signals Defined
3) "0" - Signal False
4) "1" - Signal True

(b) Input/output behavior of the 'Fork' module

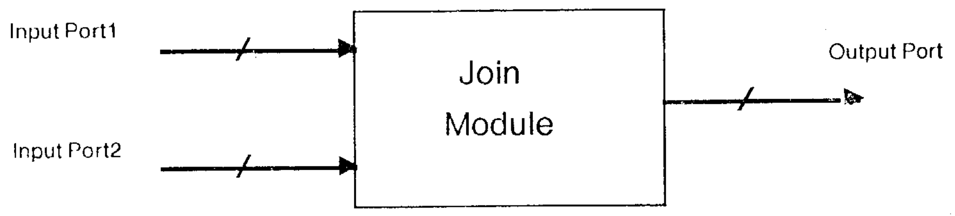
Figure 2.10

The 'fork' module is used to enable parallel operation in a self-timed system. The module has one input port and two output ports. Every data signal of the input port must be routed to at least one of the two output ports. Figure 2.10 shows the terminal structure of the 'fork' module and its input/output timing constraints. The 'fork' module starts off concurrent activities at its output ports whenever its input port becomes defined and waits for both output activities to terminate before allowing another module transaction.

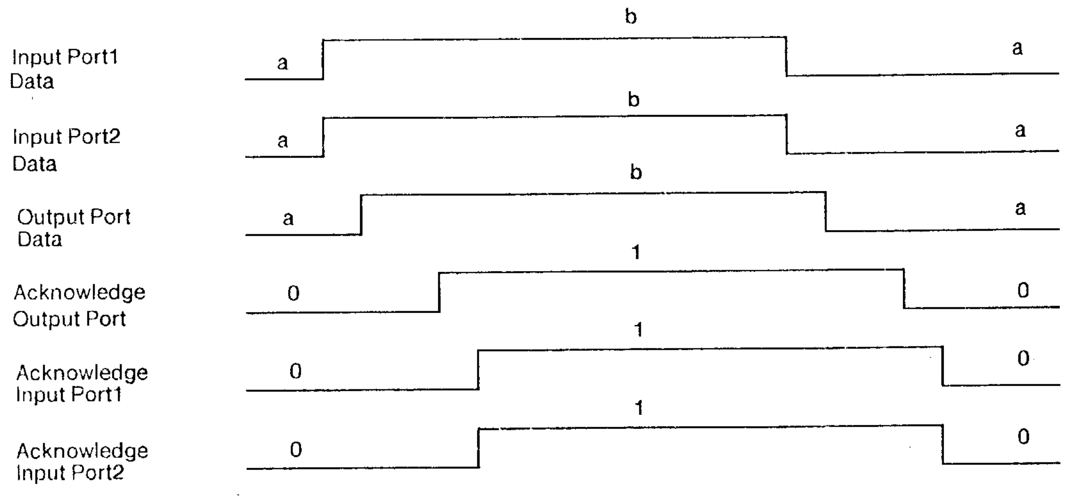
The 'join' module is used to synchronize the operation of modules in a self-timed system. The module has two input ports and one output port. The number of data signals on the input ports need not be the same. The number of data signals of the output port is the sum of the number of data signals of the two input ports. Figure 2.11 shows the terminal structure of the 'join' module and its input/output timing constraints. The 'join' module starts the operation of its output port only after both its input ports become defined.

The 'predicate' module is used to control conditional execution of sections of a self-timed system. The module has one input port and no output ports. Figure 2.12 shows the terminal structure of the predicate module and its input/output timing constraints. The predicate module is the only control module whose functionality can be specified by the designer. The predicate module consists of combinatorial logic whose functionality defines the predicate. When the predicate port becomes defined, the combinatorial logic of the 'predicate' module sets exactly one of the two acknowledge "t" or "f" signals to true. The "t" acknowledge signal indicates that the predicate evaluated true, and similarly, the "f" acknowledge signal indicates that the predicate evaluated false. The data signals of the predicate port can now change to the undefined state. The combinatorial logic of the 'predicate' module must then reset the acknowledge "t" or "f" signal that was true. It is important to note that the value of the predicate is only dependent on the data signals of the predicate port.

A 'source' module is used to stimulate repeated operation of a section of a self-timed system indefinitely. It has exactly one output port which has the same data value from one transaction to the next. Figure 2.13 shows the terminal structure of the 'source' module and its input/output timing constraints. As an example of their use, a 'source' module can be used to supply a constant to a subsection of a self-timed system. Similarly, a 'sink' module has exactly one input port and is used to mark the end of a computation at an output port of a module. Figure 2.14 shows the terminal structure of the 'sink' module and its input/output timing constraints. As an



(a) 'Join' module logic symbol



1) "a" - All Data Signals Undefined

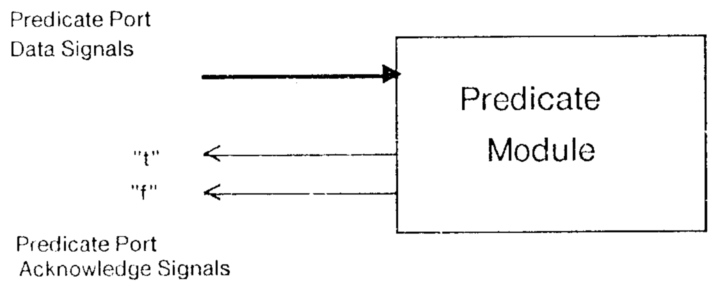
2) "b" - All Data Signals Defined

3) "0" - Signal False

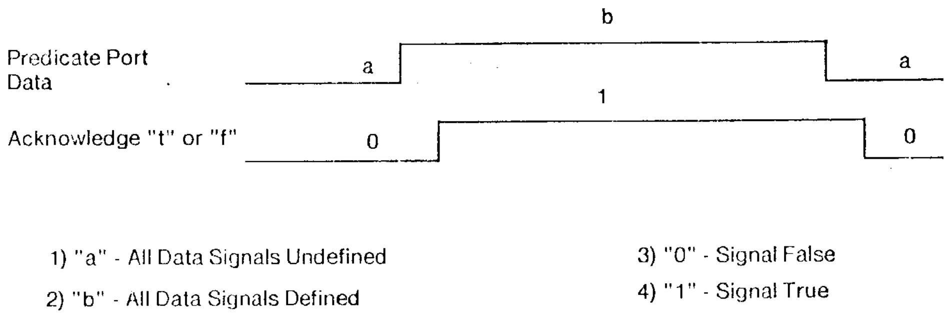
4) "1" - Signal True

(b) Input/output behavior of the 'Join' module

Figure 2.11

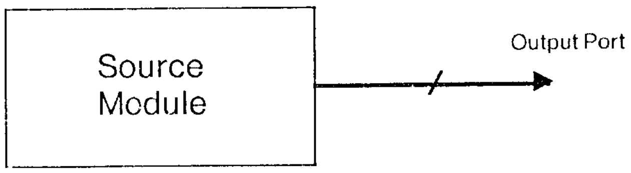


(a) 'Predicate' module logic symbol

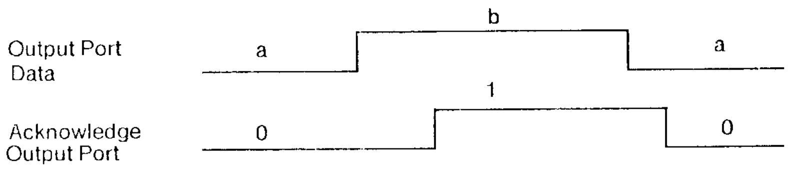


(b) Input/output behavior of the 'Predicate' module

Figure 2.12

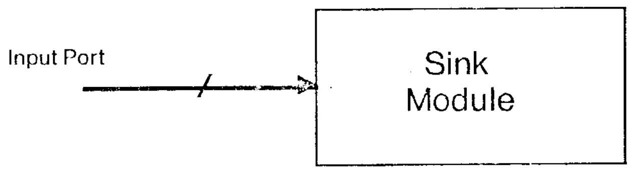


(a) 'Source' module logic symbol

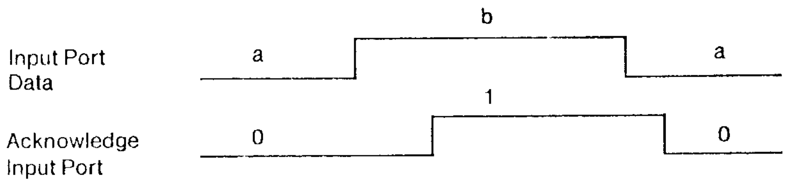


(b) Input/output behavior of the 'Source' module

Figure 2.13



(a) 'Sink' module logic symbol



(b) Input/output behavior of the 'Sink' module

1) "a" - All Data Signals Undefined
2) "b" - All Data Signals Defined
3) "0" - Signal False
4) "1" - Signal True

Figure 2.14

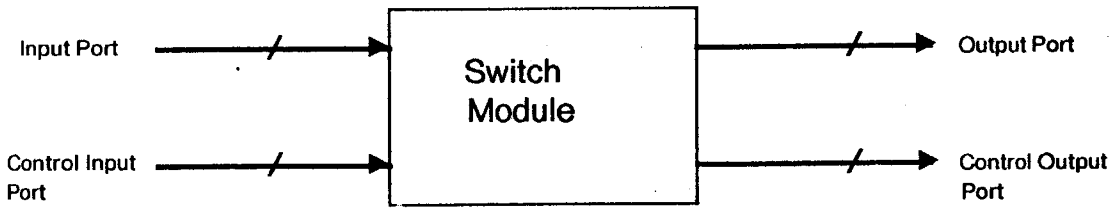
example of the use of a 'sink' module, consider a self-timed subsystem that has a 'sink' module connected to one of the output ports of a 'conditional' module. In such an arrangement the computation is to be terminated if a certain condition exists. The condition is defined by the 'predicate' module connected to the 'conditional' module and can be an error condition, the end of computation, etc.

A 'switch' module is used to conditionally block the operation of a component of a self-timed system. This is similar in concept to semaphores in programming languages which are used to synchronize sequential processes. Figure 2.15 shows the terminal structure of the 'switch' module and its input/output timing constraints. In the figure, the switch is initially in the off state. The input and output ports must have the same number of signals (greater than one), and the control input and control output ports must have exactly one data signal. The module has a switch internal to it that conditionally connects the input port to the output port. The state of this switch can be altered by the control input port. The switch is turned on as soon as the control input port gets defined with a '1' value. However, the control input port cannot turn off the switch while a transaction is in progress at the input and output ports. If the switch is in the on state and the input port gets defined, a single channel transaction takes place at the input and output ports. Once the switch has been turned on or off, the control input port is connected to the control output port and a single channel transaction takes place.

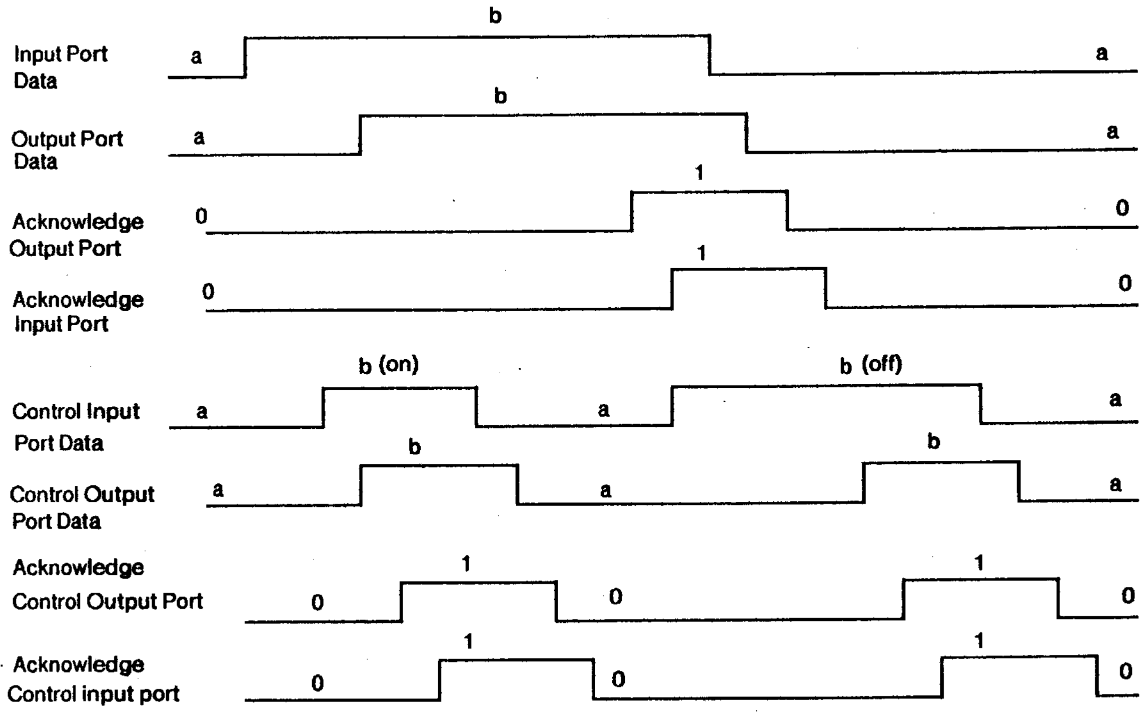
2.3 Legal Interconnection of Self-Timed Systems

This section will present the interconnection rules for the modules of a self-timed system. However, these rules will not guarantee the absence of deadlocks in a self-timed system that satisfies these interconnection rules. The task of the detection and prevention of deadlocks is left entirely to the designer of the system. The goal in presenting the following interconnection rules is to prevent the obvious cases where deadlocks are guaranteed to occur, and to prevent interconnections that cannot possibly have any meaning.

1) the channels of a self-timed system can only connect the output port of a module to the input port of some other module.
2) the number of data signals of the source and destination module ports of a channel must be the same. Also, the corresponding data and acknowledge signals must be connected correctly.
3) the input port of a 'predicate' module can only be connected to the predicate output port of a 'conditional' or 'iteration' module, and vice-versa.
4) every input and output port of a module must either be connected to another module or be an



(a) 'Switch' module logic symbol



(b) Input/output behavior of the 'Switch' module

1) "a" - All Data Signals Undefined
2) "b" - All Data Signals Defined
3) "0" - Signal False
4) "1" - Signal True

Figure 2.15

input or output port of the system.
5) every cycle in a system must have exactly '2*n + 1' 'register' modules in it, for some 'n' greater than zero. For the simple case with 'n' equal to one we have a single feedback register. A feedback register must store both the present state and next state variables, and each variable must be separated from the other by the spacer state.
6) every pipeline register must have '2*n' 'register' modules in it, for some 'n' greater than zero. Because of the reset signalling protocol, we must also store the spacer state for every pipelined signal.
7) external connection into or out of the body of the iteration (the self-timed subsystem connected between the iteration output port and the iteration input port of an 'iteration' module) is disallowed.

2.4 Underlying Circuit Model for Hardware Realization

The next two chapters will present the design of control and 'user' modules. The circuit model used in specifying the designs of the modules assumes that both the logic elements and interconnect have delays associated with them. The signals at the outputs of the logic elements are assumed to be at one of two possible levels "0" or "1," and the logic elements are assumed to switch instantaneously between these two levels.

It is assumed that the module designs presented in the next two chapters will be realized in nMOS VLSI where it is possible to meet the timing requirements within a module by adjusting the interconnect and/or logic element delays. In nMOS, the delays associated with the interconnect wires can be varied by changing their width and also by changing their material. The delay of the interconnect wires is inversely proportional to the width of the wires, and metal wires have the least delay followed by diffusion and polysilicon wires. The delay of the logic devices can be varied by changing their size. For a given load, the delay is inversely proportional to the size of the drivers. Also the load itself can be increased by adding parasitic capacitance.

3. Design of Control Modules

This Chapter will present designs for the 11 control module types that were specified in Chapter2. The design for each module type is a template which can be used to realize an arbitrary module of a given type. The number of data signals at the input and output port of each module type is variable, however, the designer must maintain the correct relationship between the number of signals at each input and output port as defined in section 2.2.

Every module in a self-timed system must function correctly independent of the delays in the input and output channels of the module. The correct behavior of a module can be guaranteed independent of the delays in the channels by ensuring that the outputs of a module change only after all internal signals are stable. This can be accomplished by satisfying timing constraints that are entirely local to the modules. The designs of the various module types are given in the last section of this chapter. These designs are given in terms of the control circuits whose behavior and designs are presented in the next section.

3.1 Design of the Control Circuits

There are seven different control circuits used in the design of the control modules. These are: 1) 'C' circuit, 2) 'D' circuit, 3) 'Switch' circuit, 4) 'Register' circuit, 5) 'Multiplexor' circuit, 6) 'Demultiplexor' circuit, and 7) 'Arbitration' circuit.

The 'C' circuit has two inputs and one output and is used as a synchronizing element (Figure 3.1a). When both inputs are the same the output follows the inputs, and when the inputs are different the output retains its previous state. This behavior is illustrated in the flow table of Figure 3.1b. The design of the 'C' circuit in terms of single rail boolean gates is given in Figure 3.1c, but correct behavior of this circuit requires that the internal feedback delay be less than any external feedback delay. A better realization is given in Figure 3.1d where the output changes state only after the internal signals have stabilized. This design is given in terms of a hysteresis gate whose transfer characteristics are given in Figure 3.1e.

The 'D' circuit is used to generate the request signal for an input port to a module as illustrated in Figure 2.3b. The logic symbol for the circuit is given in Figure 3.2a and its realization in terms of boolean gates and a 'C' circuit in Figure 3.2b. If the channel data signals are reset the output of the 'D' circuit is "0" and does not change to "1" until every data signal becomes defined. Once the output of the 'D' circuit changes to "1" it remains a "1" until every data signal of the input port resets.

The 'switch' circuit is used to control the data signals at an output port of a module. The circuit has dual rail data inputs, dual rail data outputs and a single rail control input (Figure 3.3a). When the control signal is "0" the data signals of the output are all in the undefined state, and when the control signal is "1" every data signal of the output is connected to its corresponding data signal at the input. Figure 3.3b gives the design of the 'switch' circuit for a single dual rail data signal using single rail boolean gates.

The 'register' circuit is used to store dual rail data signals. The circuit has dual rail data inputs, dual rail data outputs and a single rail load input (Figure 3.4a). When the load signal is "1" the output data signals follow their corresponding input data signals, and when the load signal is "0" the output data signals maintain their previous state. The flow table for a single line of a dual rail register is given in Figure 3.4b, and a realization of this flow table using boolean gates is given in Figure 3.4c. The correct operation of this circuit depends on the requirement that the internal feedback delays should be less than any external feedback delays. A design not having these requirements is given in Figure 3.4d in terms of boolean gates, a tri-level 'OR' gate (Figure3.4e) and a hysteresis gate (Figure 3.4f).

The 'multiplexor' circuit is used to multiplex dual rail data signals. The circuit has two sets of dual rail data inputs, one set of dual rail data outputs and two single rail control inputs (Figure 3.5a). The number of dual rail signals at the inputs and output must be the same, and every dual rail data signal at the output corresponds to a pair of dual rail data signals, one from each input. Both control signals should never be "1" simultaneously, and whenever a particular control signal is "1" the dual rail signals at the output are connected to their corresponding signals from the selected input. When both control signals are "0" the dual rail signals of the output are in the undefined state. Figure 3.5b gives the design of the circuit for a single dual rail data signal at the inputs and output using single rail boolean gates.

The 'demultiplexor' circuit is used to demultiplex dual rail data signals. The circuit has two sets of dual rail data outputs, a set of dual rail data inputs and two single rail control inputs (Figure 3.6a). The number of dual rail data signals at the outputs and input must be the same and every signal at the input corresponds to a pair of signals, one at each output. Both control signals should never be "1" simultaneously and whenever a particular control signal is "1" the corresponding output signal set is defined with data of the input. Whenever a control signal is "0" the data signals of the corresponding output set are in the undefined state. Figure 3.6b gives the design of the demultiplexor circuit for a single dual rail data signal using boolean gates.

The 'arbitration' circuit is used to arbitrate two input requests and grant a request to only one of its two outputs. The 'arbitration' circuit has two input and output 'request-acknowledge' signal pairs (Figure 3.7a) where each input signal pair corresponds to a unique pair at the output. Each 'request-acknowledge' signal pair is assumed to follow the reset signaling protocol defined in Figure 2.3a. The request signals of input1 and input2 can arrive concurrently to the 'arbitration' circuit, but the output request will be granted mutually exclusively to only one of output1 or output2 [3]. When an output request is granted to a particular input, the request signal of the selected input is connected to the request signal of the corresponding output. A complete cycle of the reset signaling protocol must take place with the selected input and output signal pair before the request for the waiting input can be granted. Figure 2.7b gives the realization of the 'arbitration' circuit using the 'C' circuit, boolean gates, d.c. voltage sources and comparators.

To guarantee the correct behavior of the 'arbitration' circuit, the designer must ensure that the delay from the "a1'" input to the "a1" output must be greater than the delay from "a1'" input to the output of the top "nand" gate in figure 3.7. A similar timing constraint must be satisfied for "a2'" and "a2."

3.2 Design of the Control Modules

This section will present the designs for the control modules using the control circuits of the previous section. The design of some of the modules involves the use of the 'C' circuit and the 'register' circuit that were defined in the last section. The specification of the initial state of a self-timed system must involve the specification of the initial state of every 'C' circuit, 'register' circuit and the inputs to the system.

The design of the 'register' module is given in Figure 3.8 where the dual-rail data signals are stored in the 'register' circuit. The data stored in the module alternates between a defined state and the spacer state. If the module is storing spacer data and the input port becomes defined, this data must be copied into the register only after the current spacer data has been accepted at the output. This synchronization is provided by the 'C' circuit. When the acknowledge signal of the output port changes to "0" the "load" signal of the 'register' circuit is set to "1." The 'D' circuit at the output port sets the acknowledge signal of the input port to "1" and resets the "load" signal via the exclusive-or gate, once the data has been stored in the module. The input port data signals can now reset, but this spacer data cannot be stored in the module until the current register data has been accepted at the output port. This synchronization is also provided by the 'C' circuit. When the acknowledge signal of the output port changes to "1" the "load" signal of the 'register' circuit is set to "1." The 'D' circuit at the output port resets the acknowledge signal of the input port and resets the "load" signal once the spacer data has been stored in the module.

The design of the 'union' module is given in Figure 3.9. The module is a port multiplexor, where at any time at most one input port can be connected to the output port. The 'multiplexor' circuit performs the multiplexing of the data signals of the input ports, and the arbitration circuit 'AC' arbitrates the requests from the two input ports and guarantees that only one input port can be connected to the output. When an input port becomes defined, the 'D' circuit of that input port sets the request input to the arbitration circuit to "1." If the resource is free and the arbitration is successful, the corresponding request signal at the output of the arbitration circuit is set to "1" and the multiplexor connects the data signals of this input port to the output. The acknowledge signal of the output port can now change to "1" which will cause the the selected acknowledge input of the arbitration circuit to change to "1" and the acknowledge signal of the selected input port to change to "1." The data signals of the selected input port can now reset which will cause the request input to the arbitration circuit to reset. The arbitration circuit, however, will not reset the request signal at the output until both the acknowledge and request signals of the selected input are "0." Therefore, the spacer data at the selected input port is transmitted to the output port. The acknowledge signal of the output port can now reset which will cause the selected acknowledge input of the arbitration circuit to reset and the acknowledge signal of the selected input port to reset. When both the acknowledge and request signals of the selected input port become "0" the request signal at the output of the arbitration circuit is reset and the output port is free.

The design of the 'arbitration' module is given in Figure 3.10. The module is similar to the 'union' module except that each input port has its associated output port. Therefore, two 'switch' circuits are required to conditionally connect an input port to its associated output port, instead of the 'multiplexor' circuit.

The design of the 'conditional' module is given in Figure 3.11. The 'demultiplexor' circuit is used to conditionally connect a subset of the data signals of the input port to one of the two output ports. The data signals of the predicate port are connected to a subset of the data signals of the input port. When the input port data signals become defined the data signals of the predicate port also become defined, and the predicate module must set one of the two acknowledge signals "t" or "f" to "1." If the "t" acknowledge signal is set to "1," the demultiplexor connects a subset of the data signals of the input port to the "true" output port. The acknowledge signal of the "true" output port can now change to "1" which will set the acknowledge signal of the input port to "1." The input port data signals can now reset which results in the resetting of the data signals of the predicate port. The acknowledge "t" signal must next be reset by the predicate module. The "2" control input to the demultiplexor, however, is not reset since the acknowledge signal of the "true" output port is "1." Therefore, the spacer data of the input port is transmitted to the "true" output port. The acknowledge signal of the "true" output port can now reset which results in the resetting of the acknowledge signal of the input port and the resetting of the "2" control input to the demultiplexor. A similar set of events occur when the predicate module sets the "f" acknowledge signal to "1" and the "false" output port is selected.

The design of the 'iteration' module is given in Figure 3.12. The 'multiplexor' circuit is used to select either the input port or the iteration input port for the next cycle of operation of the module. The input port is selected by the multiplexor when the iteration is first started, and for all subsequent cycles of the iteration the iteration input port is selected. The two cross-coupled "nor" gates at the input ports control the selection of the appropriate input port. The 'demultiplexor' circuit is used to transmit the selected input port to either the output port or the iteration output port. If the predicate evaluates false the selected input port is connected to the output port, and otherwise connected to the iteration output port. When the module is reset the input port, the predicate port, the output port and the iteration output ports are reset. The self-timed subsystem connected between the iteration output port and the iteration input port comprises the body of the iteration. Since the body of the iteration forms a cyclic structure it must have at least one feedback register (3 register modules connected in series) in the cycle. Because the body of the iteration must have a feedback register the data signals of the iteration input port must be defined when the module is reset.

When the iteration module is reset the control signal "1" of the multiplexor is set to "1." Therefore, when the input port becomes defined it is connected to the output of the multiplexor. The predicate port data signals are connected to a subset of the data signals at the output of the multiplexor, and the predicate module must next set one of the two acknowledge signals to "1." If the "f" acknowledge signal is set to "1" it signals that the iteration has terminated and a subset of the data signals at the output of the multiplexor are connected to the output port. The acknowledge signal of the output port will next change to "1" which will set the acknowledge signal of the input port to "1." When the input port data signals reset the acknowledge "f" signal will change to "0" but the "1" input to the demultiplexor will not be reset since the acknowledge signal of the output port is set. The spacer data of the input port will therefore be transmitted to the output port which will cause the acknowledge signal of the output port to reset followed by the resetting of the acknowledge signal of the input port.

If the predicate module sets the acknowledge "t" signal to "1" it indicates that another cycle of the iteration must be performed. A subset of the data signals at the output of the multiplexor are connected to the iteration output port. After the data signals at the iteration output port become defined the acknowledge signal of the iteration output port should change to "1" followed by the resetting of the data signals of the iteration input port. The resetting of the data signals at the iteration input port will cause the "1" input of the multiplexor to reset followed by the setting of the "2" input of the multiplexor. The predicate port data signals will therefore reset causing the acknowledge "t" signal to reset. However, the "2" input of the demultiplexor will not be reset since the acknowledge signal of the iteration output port is set. The spacer data at the output of the multiplexor will therefore be transmitted to the iteration output port. This will cause the acknowledge signal of the iteration output port to reset followed by the defining of the data signals at the iteration input port. The set of events from this point on are similar to those when the iteration was first started except that the data signals of the iteration input port are selected by the multiplexor instead of the data signals of the input port. When the iteration terminates the "2" input to the multiplexor resets followed by the setting of the "1" input. The iteration module is thus in the same initial state at the end of the iteration.

The design of the 'source' module is given in Figure 3.13 where the box labeled "constant" defines a constant dual rail datum. If the acknowledge signal of the output port is reset the data signals of the port are defined with the constant. The acknowledge signal of the output port should next change to "1" which will cause the data signals of the output port to reset. The cycle of operation is repeated again once the acknowledge signal of the output port resets.

The design of the 'sink' module is given in Figure 3.14. When the data signals of the input port are defined the 'D' circuit sets the acknowledge signal to "1." This should be followed by the resetting of the data signals of the input port which will cause the resetting of the acknowledge signal of the input port.

The design of the 'fork' module is given in Figure 3.15. When the module is reset all the input and output ports are reset. Starting from the module reset state, when the data signals of the input port become defined the data signals of both output ports are defined with a subset of the data signals of the input port. The 'C' circuit guarantees that the acknowledge signal of the input port is not set to "1" until the acknowledge signals of both output ports are set to "1." The data signals of the input port should reset after the acknowledge signal of the input port becomes "1" causing the data signals of both output ports to reset. The 'C' circuit also prevents the acknowledge signal of the input port from resetting until the acknowledge signals of both output ports reset.

The design of the 'join' module is given in Figure 3.16. The data signals of the output port consist of the data signals of both input ports. When the module is reset both input ports and the output port are reset. Starting from the module reset state, the output port data signals will not be defined until the data signals of both input ports are defined. The acknowledge signal of the output port should change to "1" after the output port becomes defined. This will cause the acknowledge signals of both input ports to be set to "1." The data signals of the output port will not reset until the data signals of both input ports reset. Once the data signals of the output port reset, the acknowledge signal of the output port should reset which will cause the resetting of the acknowledge signals of both input ports.

The design of the 'predicate' module is given in Figure 3.17. The combinatorial logic defines a single dual rail signal whose value defines the value of the predicate. Chapter 4 will present the general design methodology for realizing dual rail combinatorial circuits. The module is in the reset state when the predicate port data signals are reset and the two acknowledge signals are reset. Starting from the module reset state, when the data signals of the predicate port become defined, the combinatorial logic will define the value of the dual rail predicate signal to either "1" or "0." The dual rail to single rail decoder connects "line1" of the of the predicate signal to the acknowledge "t" signal and "line2" to the acknowledge "f" signal. After the selected acknowledge signal is set to "1" the predicate port data signals can reset which will cause the dual rail predicate signal to change to the undefined state. The acknowledge "t" or "f" signal that was set to "1" should now reset, thereby returning the module to the reset state. Since the two lines of a dual rail signal are never both true simultaneously we are guaranteed that only one of the two acknowledge signals of the predicate port can be true at any time.

The design of the 'switch' module is given in figure 3.18a. The design is given in terms of the 'arbitration' module "A" and the 'SW' circuit. The 'SW' circuit is similar to the 'switch' circuit except that it has a state variable that controls the state of the switch (Figure 3.18b). When the control input to the 'SW' circuit gets defined with a value of "1" the switch is turned on, and when it gets defined with a value of "0" the switch is turned off. The 'arbitration' circuit "A" prevents the state of the switch from being set in the middle of a transaction between the input and output ports, and also prevents a transaction between the input and output ports when the state of the switch is being set.

If the input port gets defined when the switch is off the data signals are blocked at the input of the 'SW' circuit. However, if the switch is on, the data signals of the input port are transmitted to the output of the 'SW' circuit. If the data signals at the output of the 'SW' successfully arbitrate through the 'arbitration' module a single transaction follows between the input and output ports of the 'switch' module. If arbitration is not successful, the control input port must have won the arbitration. The input port data signals will be output to the output port (after the transaction on the control input and control output ports has terminated) if the switch was turned on. If the control input signal turned the switch off, the input port data signals will be blocked till the switch is turned on.

3.3 Timing Requirements for the Control Modules

This section will present the timing requirements for the control module designs given in the previous section. In defining the timing requirements each delay will be specified as the delay between two signals within a module. It should be assumed that this delay includes the delay of every logic element and interconnect in the path between the two signals. For the starting signal of the path the delay also includes the delay of the logic element driving the the signal, and for the terminating signal of the path the delay also includes the delay of the interconnect connected to the logic element driving the terminating signal of the path. If the terminating signal is connected to an input of a control circuit, then this delay must also include the time required for all signal paths to settle within the control circuit from this input.

The design of the 'register' module is given in figure 3.8. In the module, the delay from the output of the 'D' circuit (at the output port) to the "load" input of the 'register' circuit must be less than the delay from the output of the 'D' circuit (at the output port) to the acknowledge signal of the input port.

The design of the 'union' module is given in figure 3.9. In the module, the delay from the "r1'" output of the 'arbitration' circuit to the "1" input of the 'multiplexor' circuit must be greater than the delay from the "r1'" output of the 'arbitration' circuit to the "a1'" input of the 'arbitration' circuit. A similar requirement holds for the "r2'" output of the 'arbitration' circuit, the "2" input of the 'multiplexor' circuit and the "a2'" input of the 'arbitration' circuit.

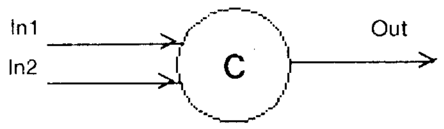
The design of the 'arbitration' module is given in figure 3.10. There are no additional timing requirements for the correct operation of this module.

The design of the 'conditional' module is given in figure 3.11. In the module, the delay from the acknowledge signal of the "false" output port to the acknowledge signal of the input port must be greater than the delay from the acknowledge signal of the "false" output port to the "1" input of the 'demultiplexor' circuit. A similar requirement holds for the acknowledge signal of the "true" output port, the acknowledge of the input port and the "2" input of the 'demultiplexor' circuit.

The design of the 'iteration' module is given in figure 3.12. In the module the delay from the acknowledge signal of the output port to the acknowledge signal of the input port must be greater than the delay from the acknowledge signal of the output port to the "1" input of the 'demultiplexor' circuit. A similar requirement holds for the acknowledge signal of the iteration output port, the acknowledge of the iteration input port and the "2" input of the 'demultiplexor' circuit.

The design of the 'switch' module is given in Figure 3.18a. The delay from the data outputs of the 'SW' circuit to the input of the 'arbitration' module should be less than the delay from the control output of the 'arbitration' module to the control input of the 'SW' circuit. This ensures that there are no gliches at the output port if the switch is turned off after the input port data signals are defined at the output of the 'SW' circuit. For the 'SW' circuit to function correctly, the delay from the control input signal to the control output signal should be greater than the delay from the control input signal to the "control" input to the 'switch' circuit.

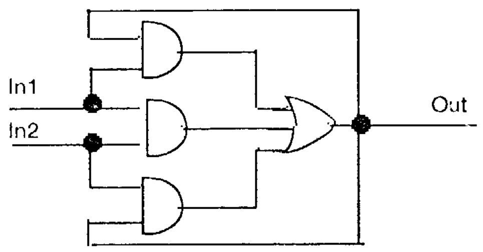
The following control modules do not have any additional timing requirements to guarantee their correct behavior: 1) the 'source' module (figure 3.13), 2) the 'sink' module (figure 3.14), 3) the 'fork' module (figure 3.15), 4) the 'join' module (figure 3.16) and 5) the 'predicate' module (figure 3.17).



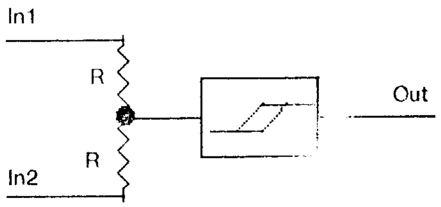
(a) 'C' circuit logic symbol

	In1,In2			
Out	00	01	11	10
0	0	0	1	0
1	0	1	1	1

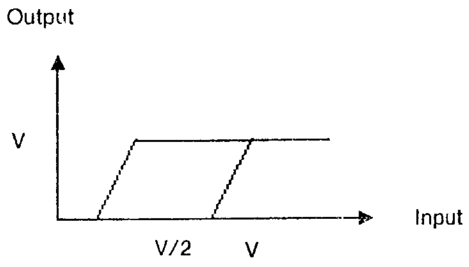
(b) Flow table for the 'C' circuit



(c) Realization of the 'C' circuit using logic gates

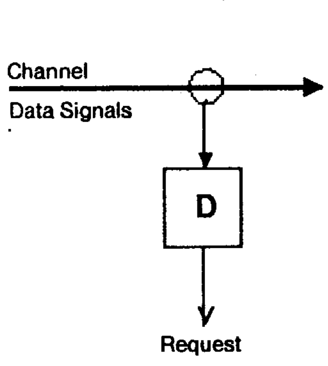


(d) Realization of the 'C' circuit using a hysteresis gate

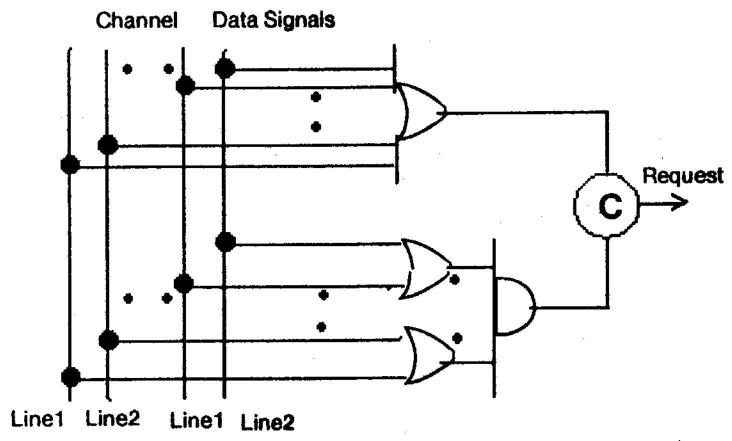


(e) Input/output behavior of the hysteresis gate

Figure 3.1 Design of the 'C' circuit

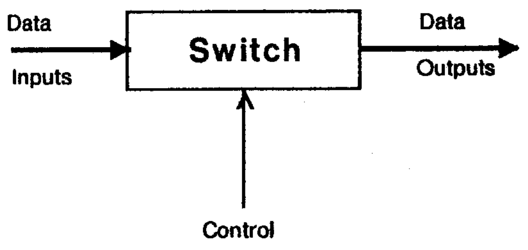


(a) 'D' circuit logic symbol

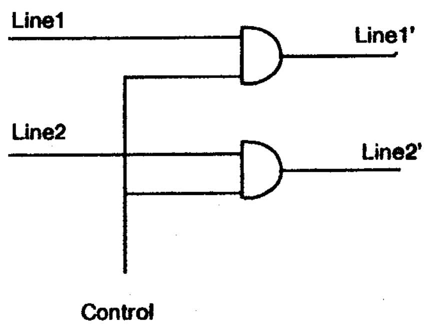


(b) Realization of the 'D' circuit

Figure 3.2 Design of the 'D' Circuit

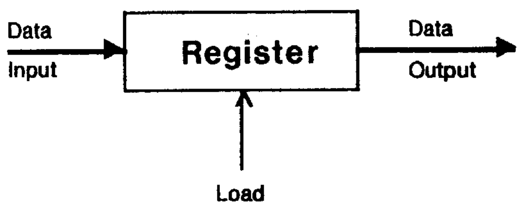


(a) 'Switch' circuit logic symbol



(b) Realization of the 'Switch' circuit for a single data signal

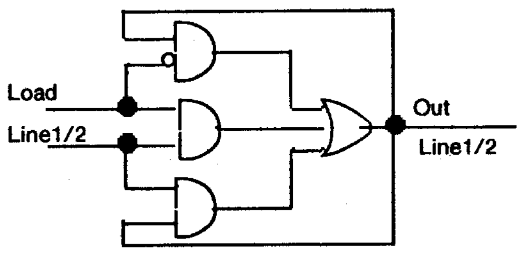
Figure 3.3 Design of the 'Switch' Circuit



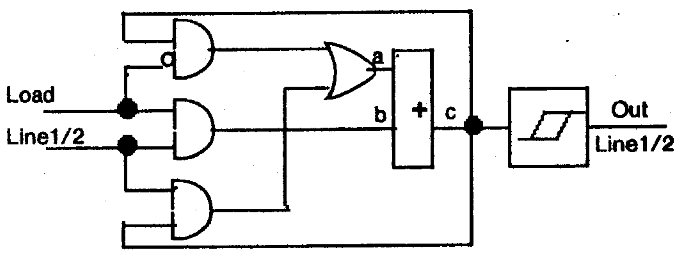
(a) 'Register' circuit logic symbol

Out \ Load, Input	00	01	11	10
0	0	0	1	0
1	1	1	1	0

(b) Flow table for the 'Register' circuit



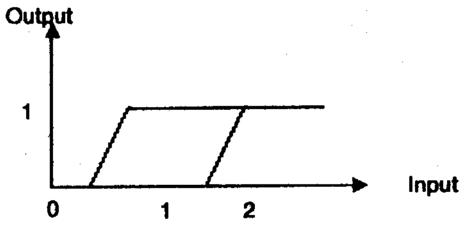
(c) Realization of the 'Register' circuit using logic gates for a single data line



(d) Realization of the 'Register' circuit with a tri-level 'OR' gate and a hysteresis gate for a single data line

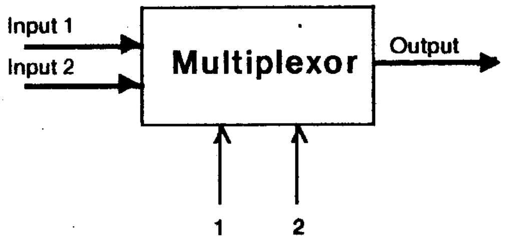
a	b	c
0	0	0
0	1	1
1	0	2
1	1	2

(e) Input/output behavior of the tri-level 'OR' gate

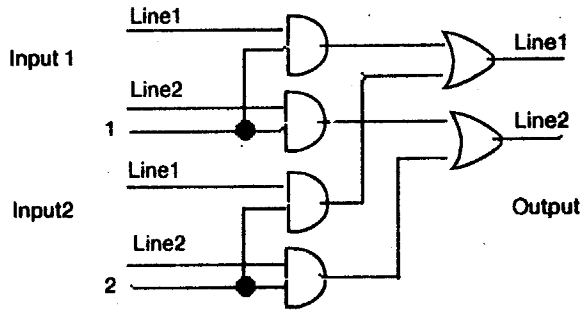


(f) Input/output behavior of the hysteresis gate

Figure 3.4 Design of the 'Register' Circuit

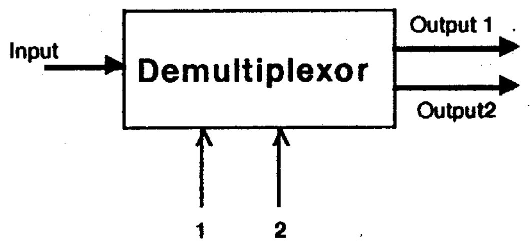


(a) 'Multiplexor' circuit logic symbol

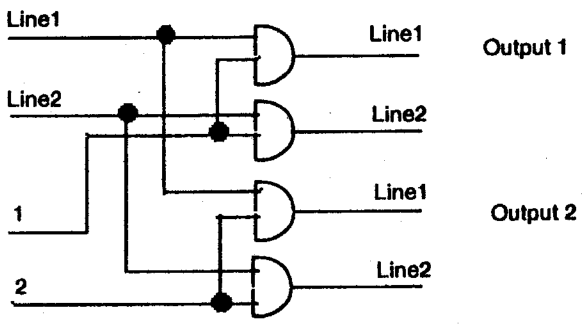


(b) Realization of the 'Multiplexor' circuit for a single data signal

Figure 3.5 Design of the 'Multiplexor' Circuit

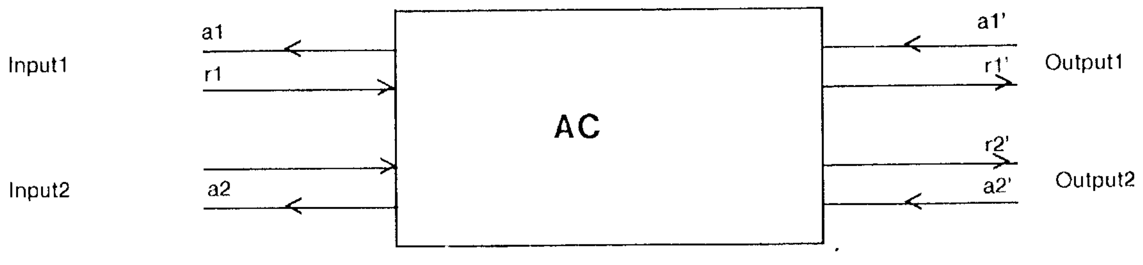


(a) 'Demultiplexor' circuit logic symbol

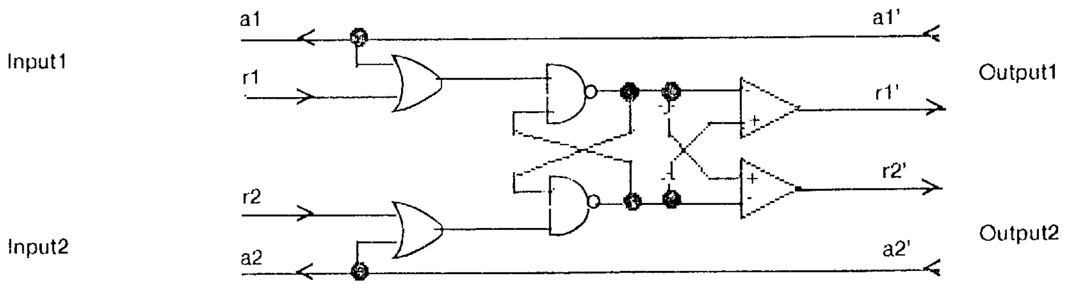


(b) Realization of the 'Demultiplexor' circuit for a single data signal

Figure 3.6 Design of the 'Demultiplexor' Circuit



(a) 'Arbitration' circuit logic symbol



(b) Realization of the 'Arbitration' circuit

Figure 3.7 Design of the 'Arbitration' Circuit

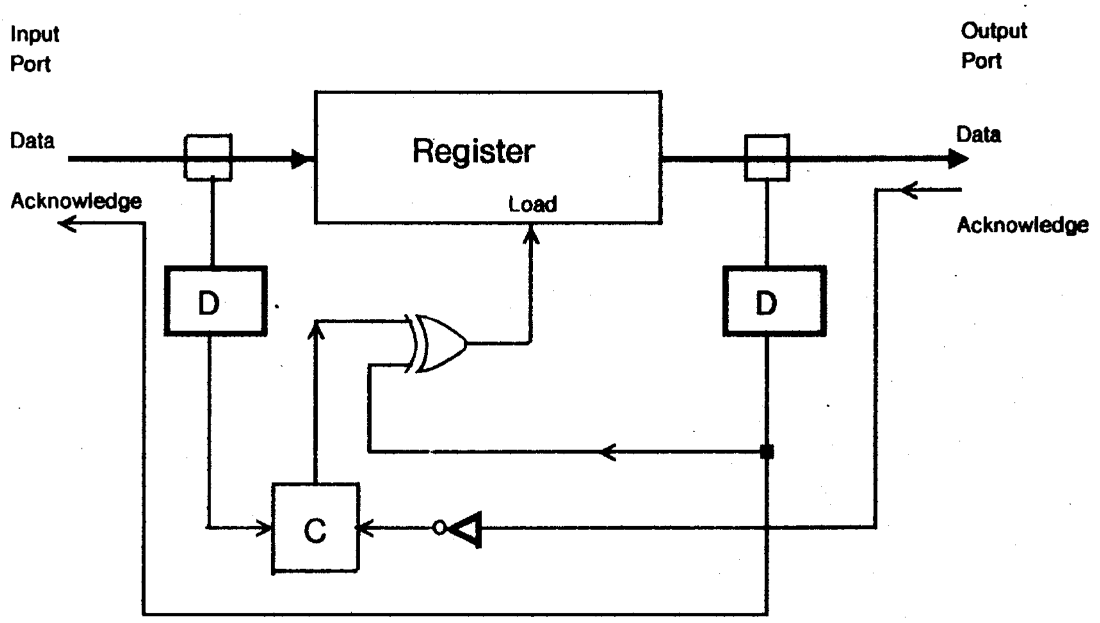


Figure 3.8 Design of the 'Register' module

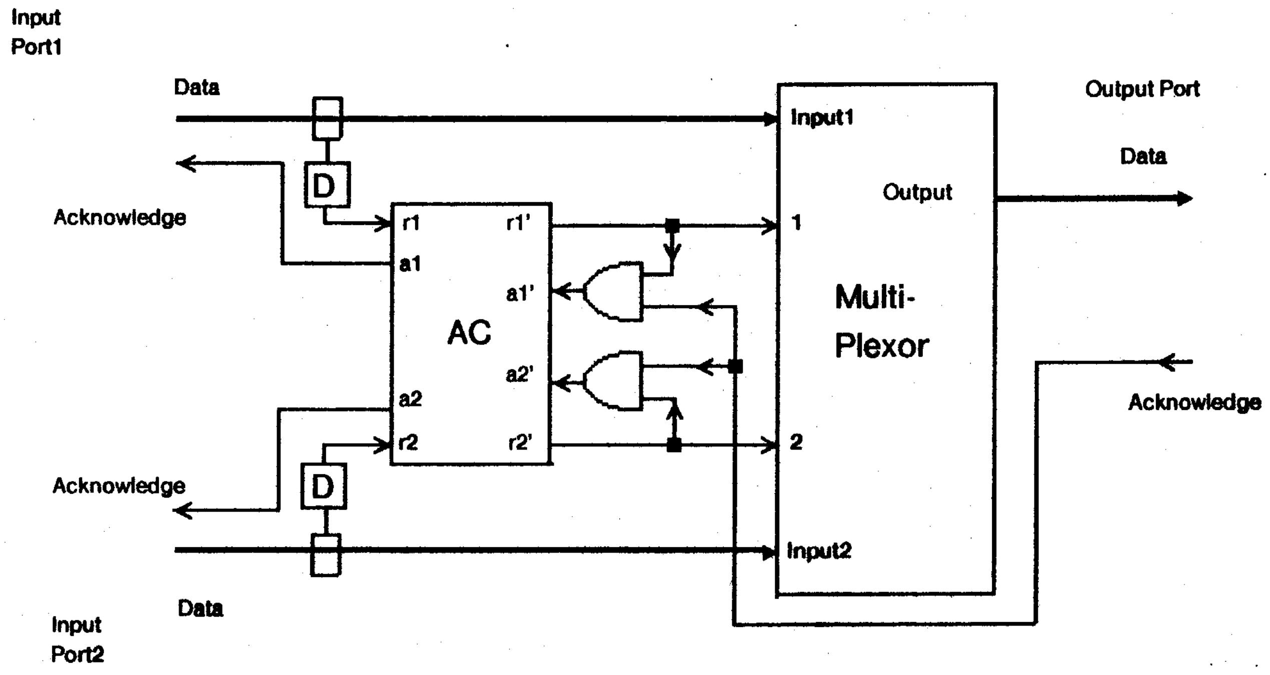


Figure 3.9 Design of the 'Union' module

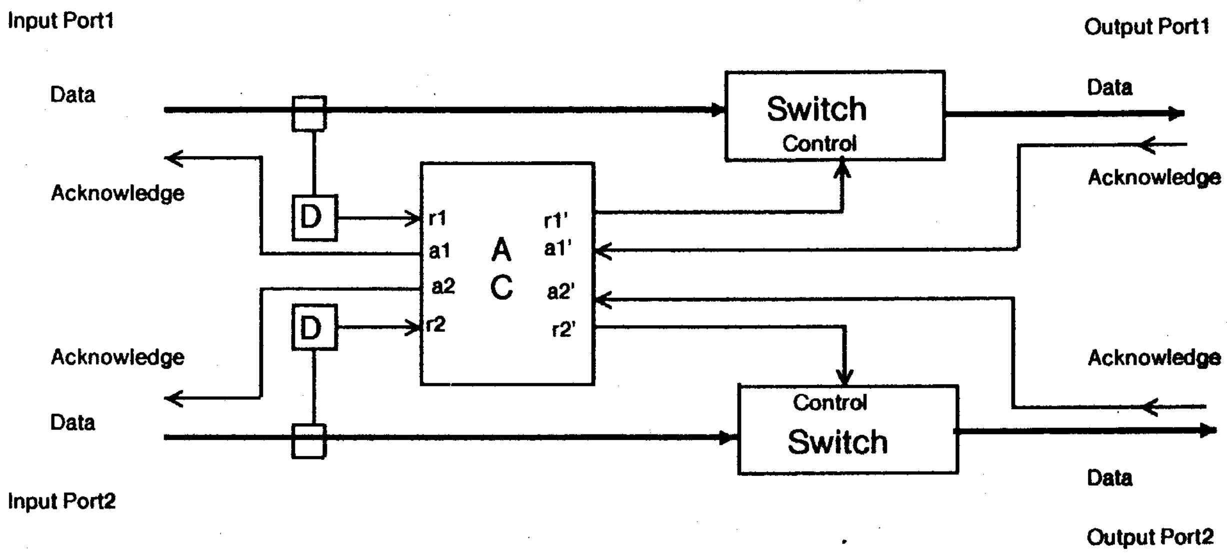


Figure 3.10 Design of the 'Arbitration' module

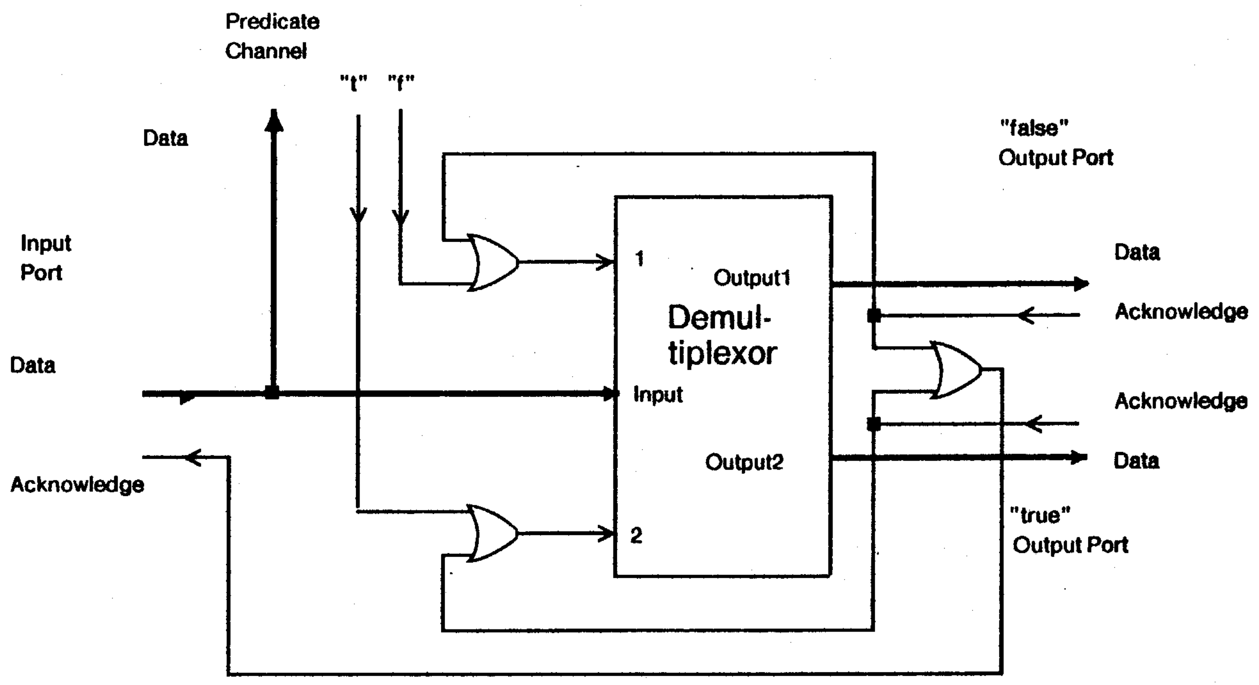


Figure 3.11 Design of the 'Conditional' module

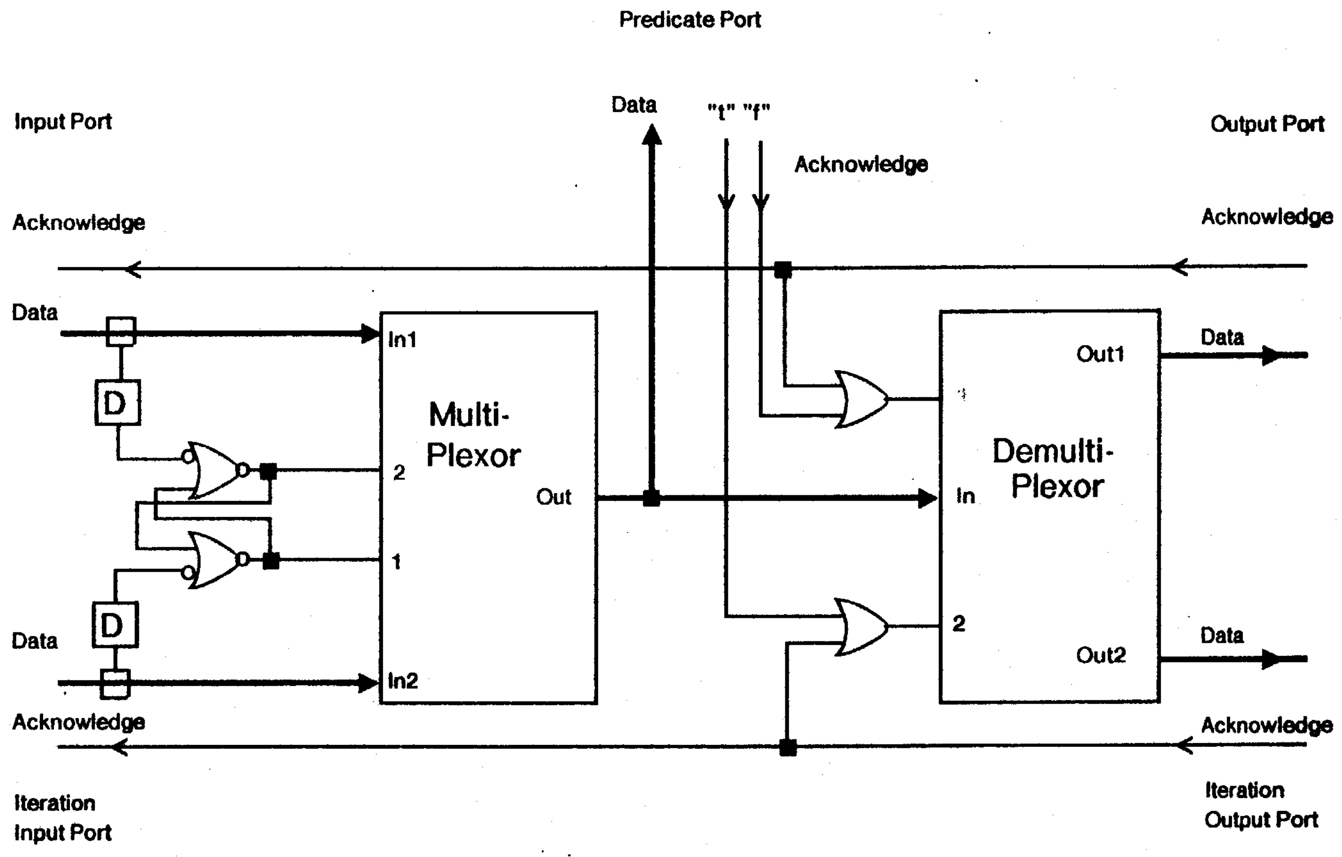


Figure 3.12 Design of the 'Iteration' module

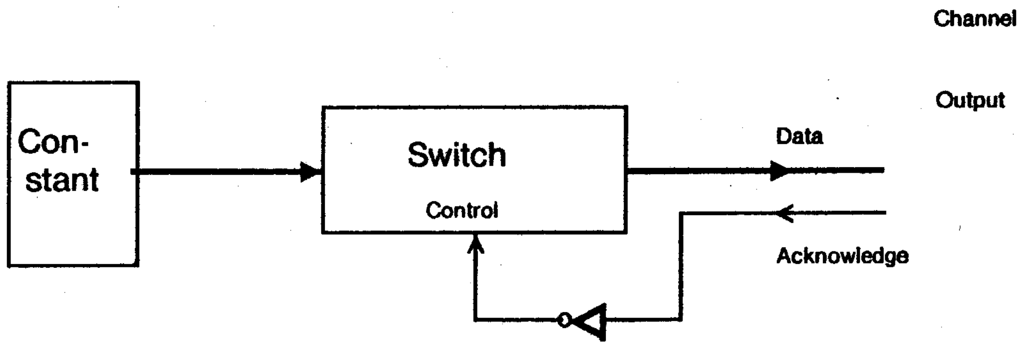


Figure 3.13 Design of the 'Source' module

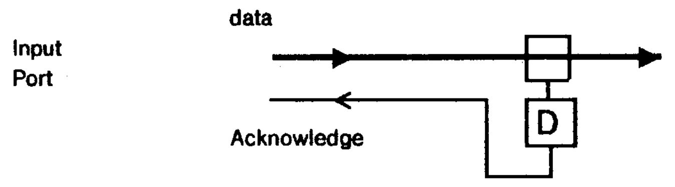


Figure 3.14 Design of the 'Sink' module

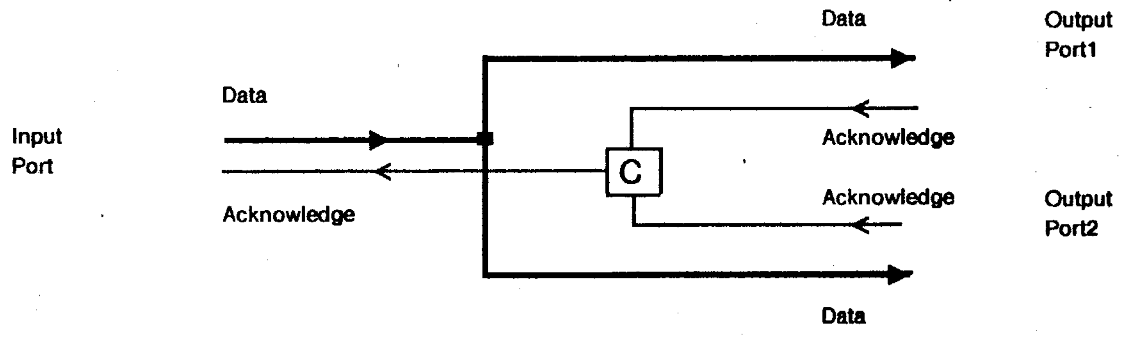


Figure 3.15 Design of the 'Fork' module

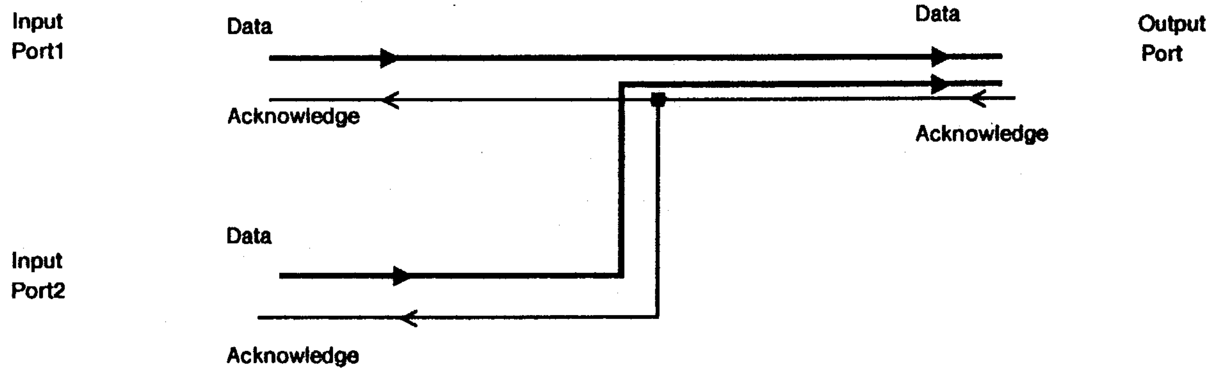


Figure 3.16 Design of the 'Join' module

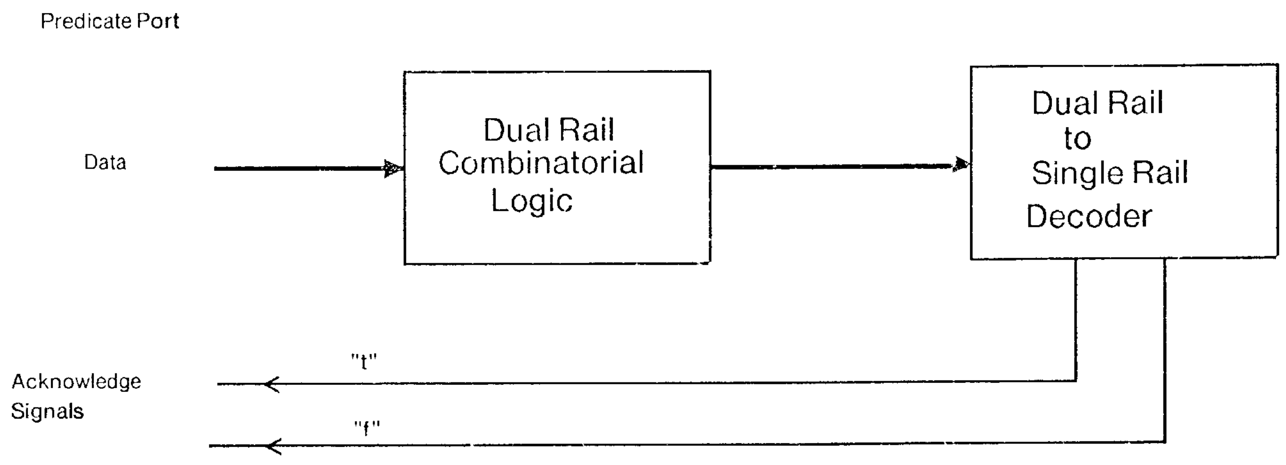

Predicate Port
Data
Dual Rail Combinatorial Logic
Dual Rail to Single Rail Decoder
"t"
Acknowledge Signals
"f"

Figure 3.17 Design of the 'Predicate' module

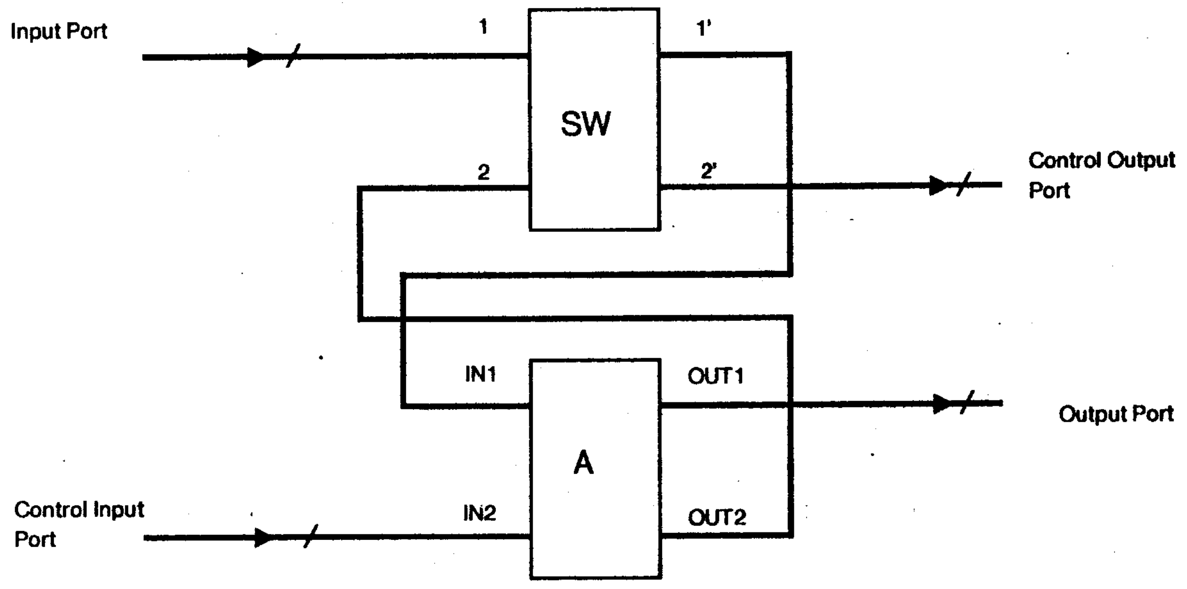


Figure 3.18a Design of the 'Switch' module

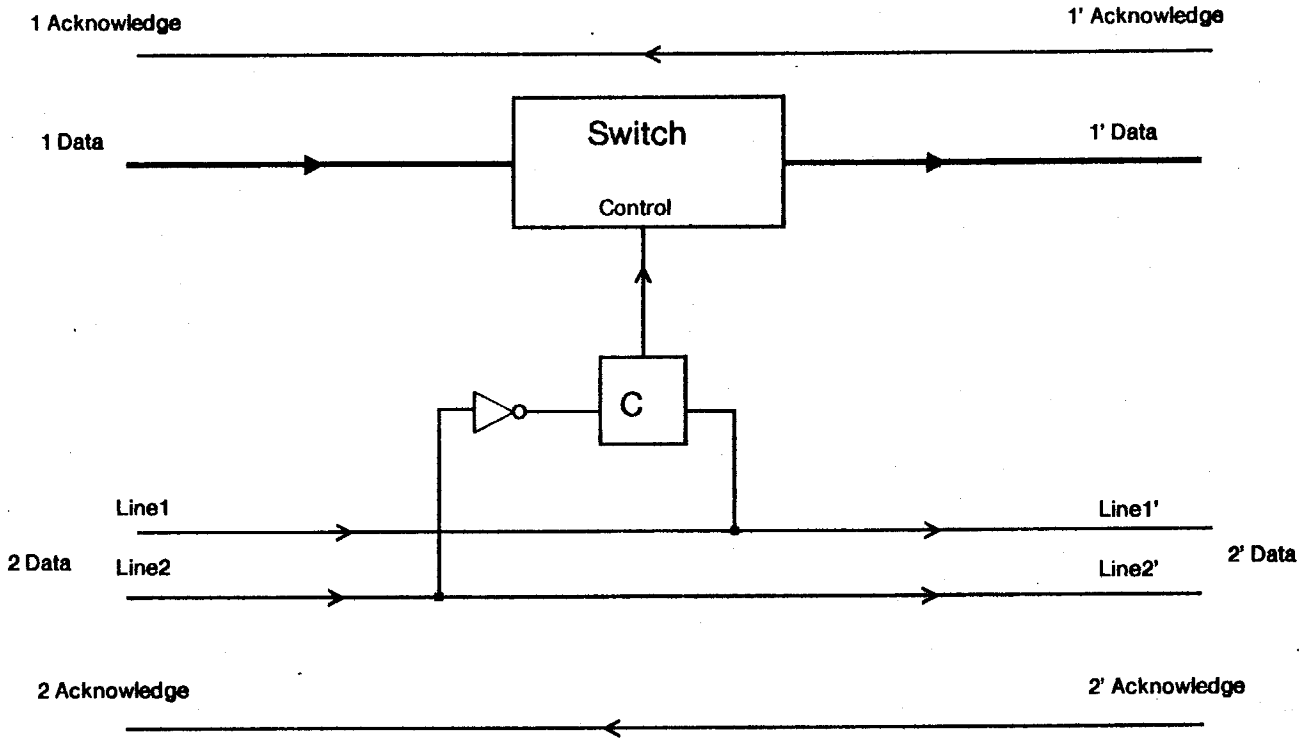


Figure 3.18b Design of the 'SW' circuit

4. Design of the 'User' Modules

The 'user' modules perform all the data transformation functions in a self-timed system. The modules have exactly one input port and one output port. Figure 4.1 gives the design of the 'user' modules in terms of dual rail combinatorial logic. The functionality of the 'user' modules is specified by defining the dual rail combinatorial logic of the module.

Dual rail combinatorial circuits consists of an acyclic interconnection of dual rail boolean logic elements. The dual rail logic elements are the dual rail 'and,' gates 'or,' gates and 'inverters.' Chapter2 defined the input/output timing constraints for the 'user' modules. The 'user' module is reset when both the input and output ports are reset. Starting from the module reset state, the signals of the output port should not change to the defined state until the signals of the input port are defined. Similarly, the signals of the output port should not change to the undefined state until the signals of the input port change to the undefined state. To guarantee the correct terminal behavior of the 'user' modules, each dual rail logic element must change to a defined state only after all its inputs become defined. Similarly, the output of a dual rail logic element must change to the undefined state only after all its inputs become undefined.

Figure 4.2 gives the logic symbol and design of the dual rail 'inverter.' The inverter has one dual rail input and output signal. The realization of the dual rail 'inverter' is extremely simple and requires only the renaming of the input signal lines at the output.

Figure 4.3 gives the logic symbol and design of the dual rail 'and' gate. The 'and' gate has two dual rail input signals and one dual rail output signal. The design of the 'and' gate is given in terms of 'C' circuits and a single rail 'or' gate. When both input signals of the 'and' gate are in the undefined state the inputs to all the 'C' circuits are "0" and the output of the 'and' gate is undefined. If only one input signal becomes defined, there will be no 'C' circuit with both inputs "1" and the output of the 'and' gate will remain undefined. When both input signals become defined, exactly one 'C' circuit will have both its input signals "1." The output of this 'C' circuit will change to "1" and the output of the 'and' gate is defined with the appropriate value. The output of this 'C' circuit will remain a "1" until both input signals become undefined. Therefore, the output of the 'and' gate will remain defined until both input signals become undefined. The four 'C' circuits each decode one of the four possible combinations of the input signals. The output of the 'and' gate should be defined with a value of "1" only when both input signals are defined with a value of "1." The other three combinations of the input signals set the value of the 'and' gate to "0."

The logic symbol and design of the dual rail 'or' gate is given in figure 4.4. The 'or' gate has two dual rail input signals and one dual rail output signal. The design is similar to the design of the dual rail 'and' gate, except that the input and output signals are inverted.

From these dual rail logic elements it is possible to realize an arbitrary dual rail combinatorial function, and hence an arbitrary 'user' module. However, it is not implied that this is the only way of implementing dual rail combinatorial functions, or even that this is necessarily the best realization. There are no additional timing requirements for the correct behavior of these logic elements, besides those mentioned in chapter3 for the 'C' circuit.

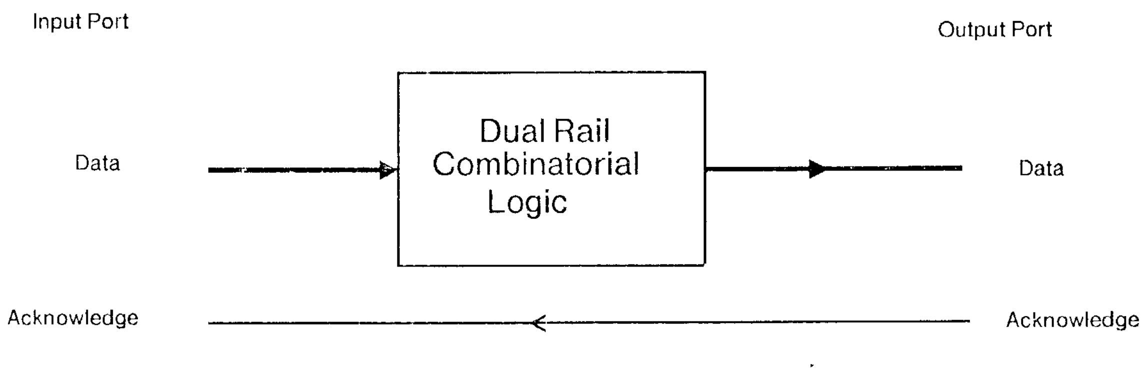


Figure 4.1 Design of the 'User' module

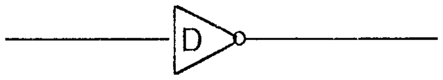


Figure 4.2a Logic symbol of the dual rail inverter

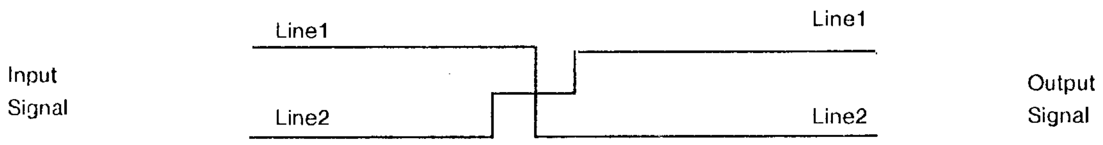


Figure 4.2b Design of the dual rail inverter

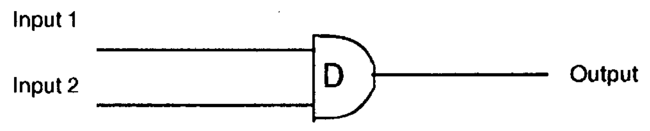


Figure 4.3a Logic symbol of the dual rail and gate

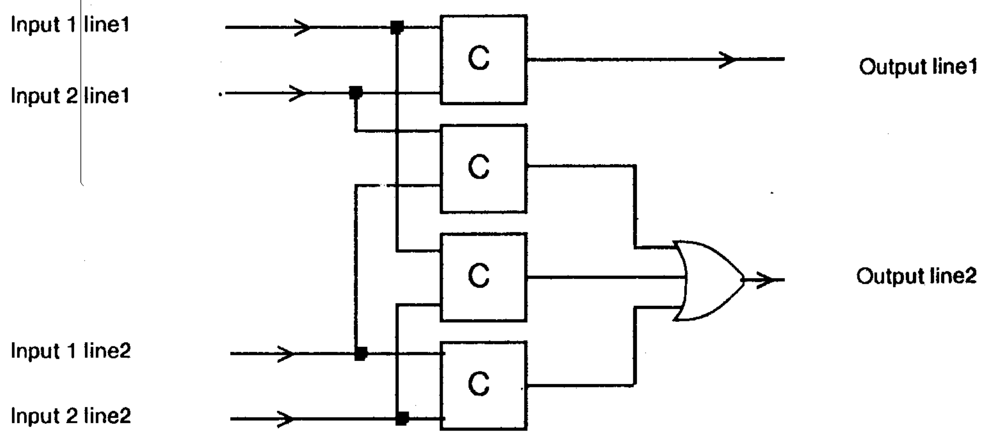


Figure 4.3b Design of the dual rail and gate

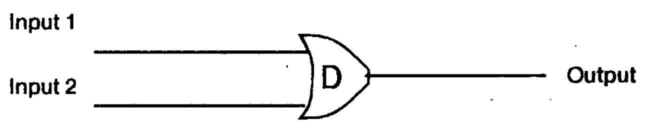


Figure 4.4a Logic symbol of the dual rail or gate

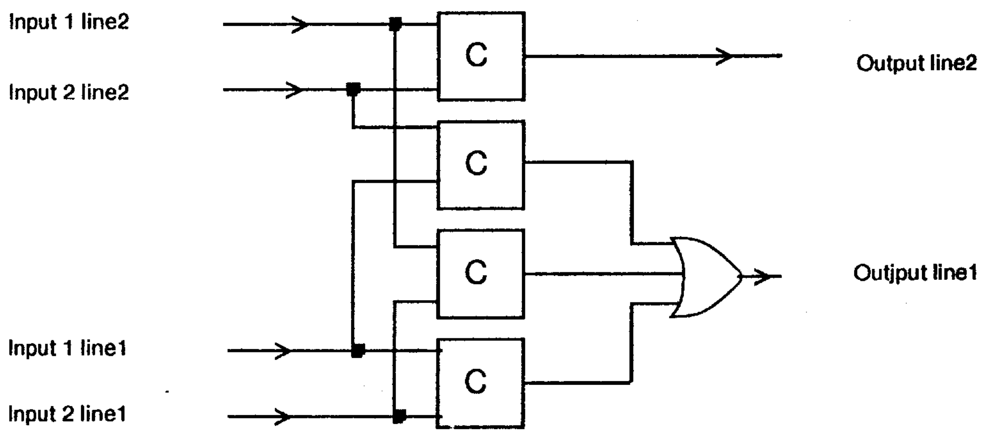


Figure 4.4b Design of the dual rail or gate

5. Design Example of a 2×2 Router

This chapter will present the design of a 2X2 router using the design methodology proposed in this thesis, and contrast it with a synchronous design. A 2X2 router is a *packet switch* that has two input and two output ports. A packet arriving at each input port requests to be output to one of the two output ports. The router can thus be used in implementing arbitrary NXN packet routing networks [13]. Figure 5.1 gives the terminal structure of a 2X2 router.

A packet is a sequence of one or more bytes, where each byte consists of eight data signals and the "last_byte_of_packet" signal. Whenever the "last_byte_of_packet" signal is "1" it indicates that the current byte is the last byte of the packet. The first byte of the packet is called the address byte. The data signals of the address byte define the direction of the packet for the various levels of routers in the routing network. The least significant data signal of the address byte defines the direction signal of a 2X2 router. If the direction signal is false, the packet is transmitted to the output port "Output1," and otherwise output to the output port "output2." The wires that connect the data signals between routers should be rotated so that the routers at each level in the routing network look at a different data signal to define their direction signal.

A router is reset when there are no packets being output from either input port. Starting from the reset state, when an input port becomes defined with the address byte of a packet, the router must look at the direction signal and forward all bytes of the packet to the selected output port. If a second packet from the other input port is to be output to the same output port it must be blocked until all bytes of the first packet are output. However, if the second packet is to be output to the free output port, its transmission can proceed concurrently with that of the first packet. It can be seen that the time required to transmit a byte of a packet through a router is not fixed. The communication protocol for transmitting the bytes of a packet must therefore be bi-directional.

There are two additional communication control signals associated with every byte of a packet. These are the 'request' and 'acknowledge' signals. These signals follow the reset signaling protocol defined in Figure 2.3a. It is important to note that direction of the acknowledge signal is opposite to that of the packet, while the direction of the request signal is the same as that of the packet. When the request signal is "1" at an input port of a router it indicates that a byte of the packet is defined and should be transmitted to the appropriate output port. The acknowledge signal of the input port should change to "1" once the byte has been transmitted to the desired output port and the acknowledge signal of the output port is "1." The request signal should next reset and the router should reset the request signal at the selected output port. When the acknowledge signal of the output port resets the router should reset the acknowledge signal of the input port. This marks the end of the transmission of the packet byte, and the next byte of the packet, or the first byte of another packet, can be transmitted in a similar fashion.

The next section will give the detailed design of a synchronous router. To give a meaningful comparison of the logical complexity of the synchronous and asynchronous designs, all component counts for the 'and' and 'or' gates are given for dual input gates.

5.1 Synchronous Design of a 2X2 Router

Before presenting the detailed design of the synchronous router it is important to mention the assumptions made regarding the synchronous system that the router is a part of.

It is assumed that the entire system operates on the same clock, and therefore the inputs to the router need not be synchronized. All signals of a channel will change in the same clock cycle and can be encoded using the single rail protocol by transmitting an explicit *request* signal with every byte of a packet. In presenting the details of the design, the clock and reset logic will be left out for simplicity.

The block diagram of the synchronous router is given in Figure 5.2. 'M1' is a packet demultiplexor with one input port and two output ports. When the address byte of a packet becomes defined at the input port it is output to output port 0 if the direction signal is false, and otherwise output to output port 1. 'M1' also remembers the value of the direction signal so that it can forward the remaining bytes of the packet (if there are any) to the appropriate output port. The "last_byte_of_packet" signal is used to indicate that the next byte at the input port is the address byte of a new packet. 'M2' is a packet multiplexor with two input ports and one output port. 'M2' is in the idle state when no packet is being transmitted from either input. If an input port becomes defined when 'M2' is idle, all bytes of this packet are transmitted to the output port. If the other input port becomes defined while a packet is being transmitted, it is blocked until all bytes of the packet from the selected input are transmitted. 'M2' has an arbiter internal to it that guarantees that only one input is selected if both input ports become defined when it is in the idle state.

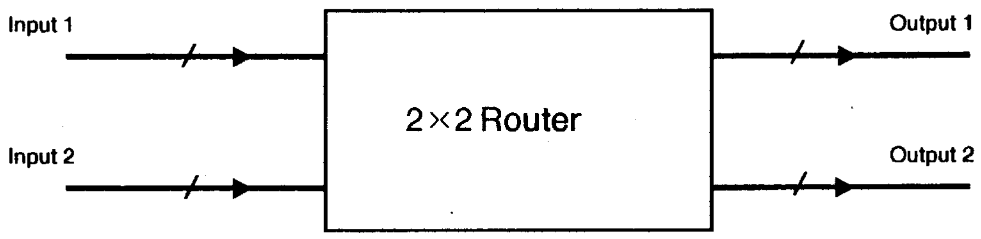


Figure 5.1 Terminal Structure of the 2×2 Router

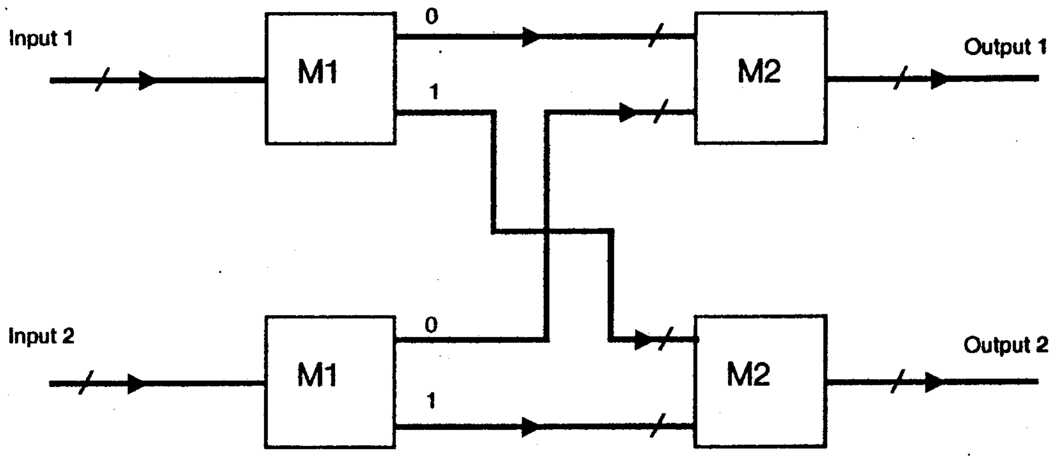


Figure 5.2 Block Diagram of the Synchronous Router

5.1.1 Detailed Design of M1

The design of 'M1' is given in Figures 5.3, 5.4 and 5.5. The design is separated into a data flow section and a control section. The data flow section consists of a demultiplexor (Figure 5.3b) which is controlled by the control section. The following signals of the input port are the inputs to the control section of 'M1':

1) DIR - this is the least significant data signal of a packet byte. This signal defines the direction of the packet during the address byte.

2) LAST - this is the 'last_byte_of_packet' signal.

3) REQ - this is the request signal of the input port.

4) ACK - this is the acknowledge signal of the input port.

The demultiplexor has two inputs from the control section, these are:

1) D1 - when this signal is true, the input port is connected to the "OUT1" port of the demultiplexor.

2) D2- when this signal is true, the input port is connected to the "OUT2" port of the demultiplexor.

Figure 5.3c gives the design of the demultiplexor for all signals except the acknowledge signal, while Figure 5.3d gives the design for the acknowledge signal. The designs are different since the direction of the acknowledge signal is opposite to the direction of all other signals in a channel. The component count for the data flow section of 'M1' is given below:

1) 22 'and' gates.

2) 1 'or' gate.

The state transition diagram for the control section of 'M1' is given in Figure 5.4, and its realization in Figure 5.5. In all the state transition diagrams the arcs only show those transitions that change the state of the machine. For the input combinations for which no arcs are shown it should be assumed that the machine remains in the same state.

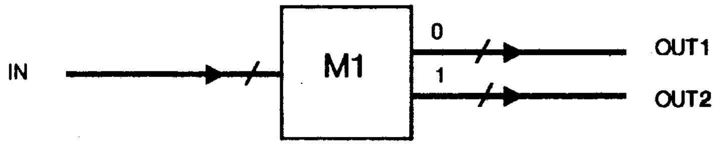


Figure 5.3a Terminal structure of M1

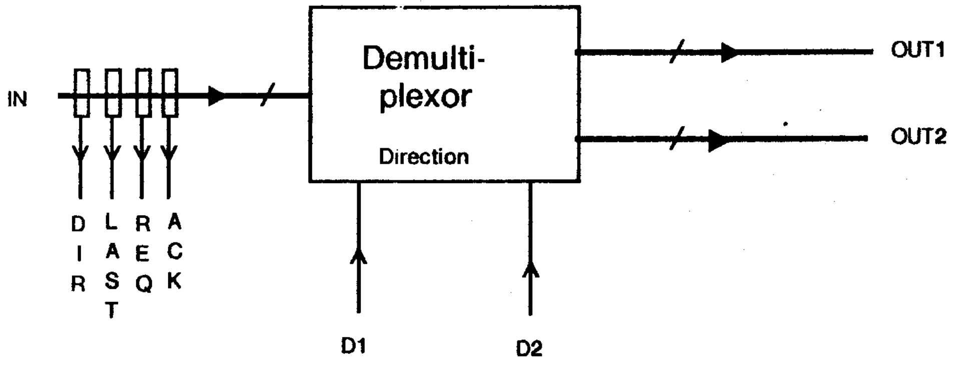


Figure 5.3b Data flow section of M1

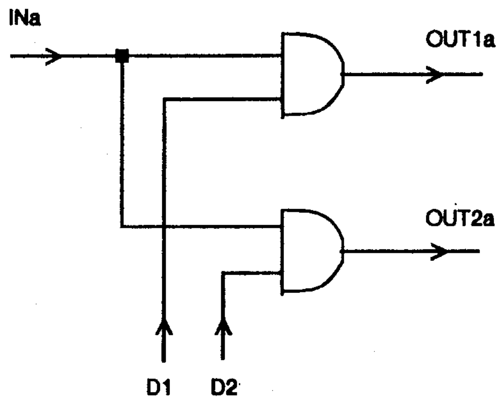


Figure 5.3c Design of the demultiplexor for all signals except the acknowledge signal

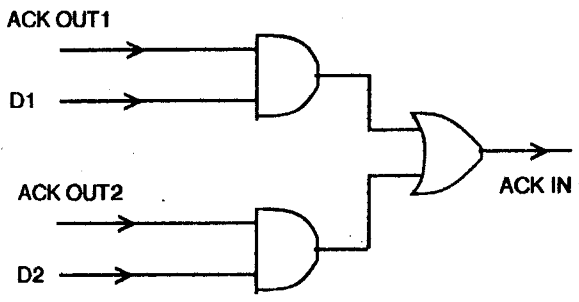
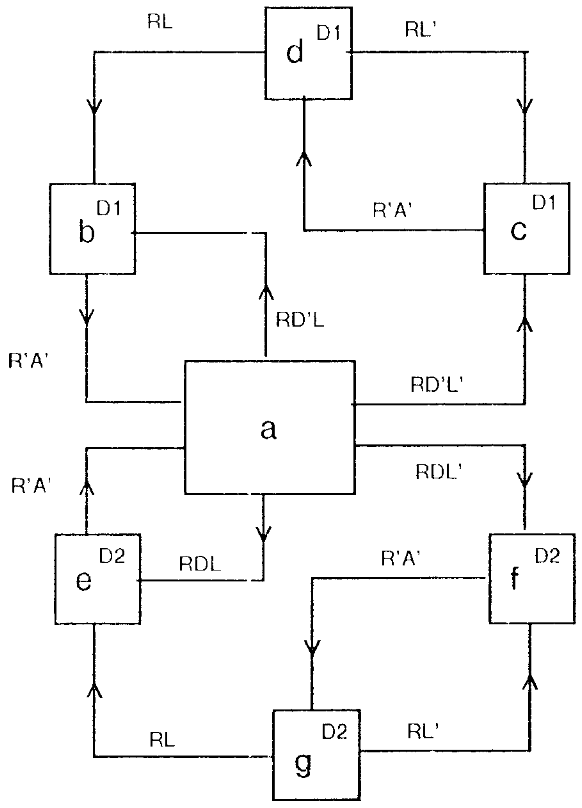


Figure 5.3d Design of the demultiplexor for the acknowledge signal



Abbreviations -	R - is the REQ (request) signal.
	L - is the LAST (last byte of packet) signal.
	A - is the ACK (acknowledge) signal.
	D - is the DIR (direction) signal.
Signal notation -	R' - stands for signal "R" inverted.
Diagram notation -	States are in the center of the boxes in lower case.
	Outputs that are "1" are in the top right hand corner of the states in upper case.
State Definition	"a" - Idle, no input packet is being transmitted.
	"b" - Last byte of a packet is being output to OUT1.
	"c" - A byte of a packet is being output to OUT1. The current byte is not the last byte of the packet.
	"d" - A byte has been transmitted to OUT1. More bytes remain to be transmitted.
	"e" - Last byte of a packet is being output to OUT2.
	"f" - A byte of a packet is being output to OUT2. The current byte is not the last byte of the packet.
	"g" - A byte has been transmitted to OUT2. More bytes remain to be transmitted.

Figure 5.4 State transition diagram for the control section of M1

Present State	Inputs	Next State	Outputs
1 2 3	D L R A	1 2 3	D1 D2
0 0 0	- - 0 -	0 0 0	0 0
0 0 0	0 0 1 -	0 1 0	0 0
0 0 0	0 1 1 -	0 0 1	0 0
0 0 0	1 0 1 -	1 0 1	0 0
0 0 0	1 1 1 -	1 0 0	0 0
0 0 1	- - 1 -	0 0 1	1 0
0 0 1	- - - 1	0 0 1	1 0
0 0 1	- - 0 0	0 0 0	1 0
0 1 0	- - 1 -	0 1 0	1 0
0 1 0	- - - 1	0 1 0	1 0
0 1 0	- - 0 0	0 1 1	1 0
0 1 1	- - 0 -	0 1 1	1 0
0 1 1	- 0 1 -	0 1 0	1 0
0 1 1	- 1 1 -	0 0 1	1 0
1 0 0	- - 1 -	1 0 0	0 1
1 0 0	- - - 1	1 0 0	0 1
1 0 0	- - 0 0	0 0 0	0 1
1 0 1	- - 1 -	1 0 1	0 1
1 0 1	- - - 1	1 0 1	0 1
1 0 1	- - 0 0	1 1 0	0 1
1 1 0	- - 0 -	1 1 0	0 1
1 1 0	- 0 1 -	1 0 1	0 1
1 1 0	- 1 1 -	1 0 0	0 1

Encoding	State
0 0 0	a
0 0 1	b
0 1 0	c
0 1 1	d
1 0 0	e
1 0 1	f
1 1 0	g

$$1^{n+1} = 1(2+3+R+A) + 1'2'3'DR$$

$$2^{n+1} = 2(1'3LR+13'R)' + 1'2'3'D'L'R + 12'3R'A'$$

$$3^{n+1} = 3(2'R'A'+2L'R)' + 1'2'3'R(D'L+DL') + 1'23'R'A' + 123'L'R$$

$$D1 = 1'(2+3)$$

$$D2 = 1$$

Abbreviations -	R - is the REQ (request) signal.
	L - is the LAST (last byte of packet) signal.
	A - is the ACK (acknowledge) signal.
	D - is the DIR (direction) signal.
Signal notation -	R' - stands for signal "R" inverted, etc..

Figure 5.5 State transition table for the control section of M1

When no packet is being transmitted from the input port 'M1' is in the idle state "a." It remains in this state until the address byte of a packet becomes defined at the input port. Figure 5.4 shows that all arcs leaving the idle state have the request signal of the input port set. When an address byte becomes defined at the input port there are four important cases, depending on the values of the direction signal and the "last_byte_of_packet" signal. For example, if the direction signal is false and the "last_byte_of_packet" signal is true 'M1' enters the state "b." In this case the input byte is the first and last byte of the packet and should be output to the output port "OUT1." The "D1" input to the demultiplexor is set in state "b" so that the input byte can be transmitted to the output port "OUT1." When the packet byte has been accepted at the output port "OUT1" the acknowledge signal of the "OUT1" port will change to "1." The demultiplexor will transmit this signal to the acknowledge signal of the input port. The request signal of the input port should next change to "0" which will cause the request signal of the output "OUT1" port to change to "0." The acknowledge signal of the "OUT1" port should now change to "0" followed by the resetting of the acknowledge signal of the input port. When both the acknowledge and request signals of the input port are reset it indicates that the transmission of the packet byte through 'M1' is complete. Since this was the first and last byte of the packet 'M1' should return to the idle state and reset the "D1" input to the demultiplexor.

The behavior of 'M1' for the other cases is similar except that it should not return to the idle state after transmitting the first byte of the packet, if the first byte is not also the last byte of the packet. For example, if a byte of a packet (that is not the last byte) has been transmitted to output port "OUT1" 'M1' enters state "d." The important difference between the states "a" and "d" is that the "D1" input to the demultiplexor is "1" in state "d." The input port is thus connected to the selected output port of the demultiplexor for the transmission of the entire packet.

The component count for the control section of 'M1' is given below:

1) 9 inverters.
2) 40 'and' gates.
3) 12 'or' gates.
4) 3 flip-flops.

5.1.2 Detailed Design of M2

The design of 'M2' is given in Figures 5.6, 5.7, 5.8 and 5.9. The design is separated into the data flow section and the control section. The data flow section consists of the multiplexor (Figure 5.6b) which is controlled by the control section. The following signals from each input port are the inputs to the control section of 'M2':

1) LAST - this is the 'last_byte_of_packet' signal.

2) REQ - this is the request signal of the input port.

3) ACK - this is the acknowledge signal of the input port.

The multiplexor has two inputs from the control section, these are:

1) D1 - when this signal is true, input port "IN1" is connected to the output port of the multiplexor.

2) D2- when this signal is true, input port "IN2" is connected to the output port of the multiplexor.

Figure 5.6c gives the design of the multiplexor for all signals except the acknowledge signal, while Figure 5.6d gives the design for the acknowledge signal. The designs are different since the direction of the acknowledge signal is opposite to the direction of all other signals in a channel. The component count for the data flow section of 'M2' is given below:

1) 22 'and' gates.

2) 10 'or' gates.

The block diagram of the control section of 'M2' is given in Figure 5.7a. The state transition diagram for the control section of 'M2' is given in Figure 5.8, and its realization in Figure 5.9. When no packet is being transmitted from either input port 'M2' is in the idle state "a." It must remain in this state until the address byte of a packet gets defined at one of the two input ports. This is signaled by the request signal changing to "1" at an input port. The arbiter 'A' arbitrates the requests from the two input ports and guarantees that the two outputs "R1a" and "R2a" are never true simultaneously. Figure 5.8 shows that 'M2' remains in the idle state until one of the two arbitrated request signals "R1a" or "R2a" changes to "1." When a byte becomes defined at an input port there are four important cases depending on which of the two signals "R1a" or "R2a" changed to "1" and the value of the "last_byte_of_packet" signal of the selected input. For example, if "R1a" changed to "1" and the "last_byte_of_packet" signal was true 'M2' would enter state "b." In this case the byte from the "IN1" port is the first and last byte of the packet. The "D1" input to the multiplexor is set in state "b" so that the packet byte at "IN1" is output to the

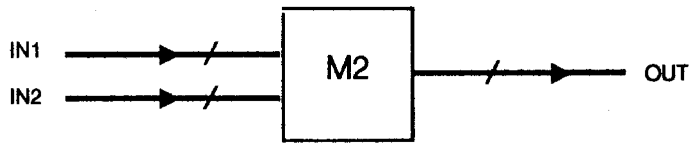


Figure 5.6a Terminal structure of M2

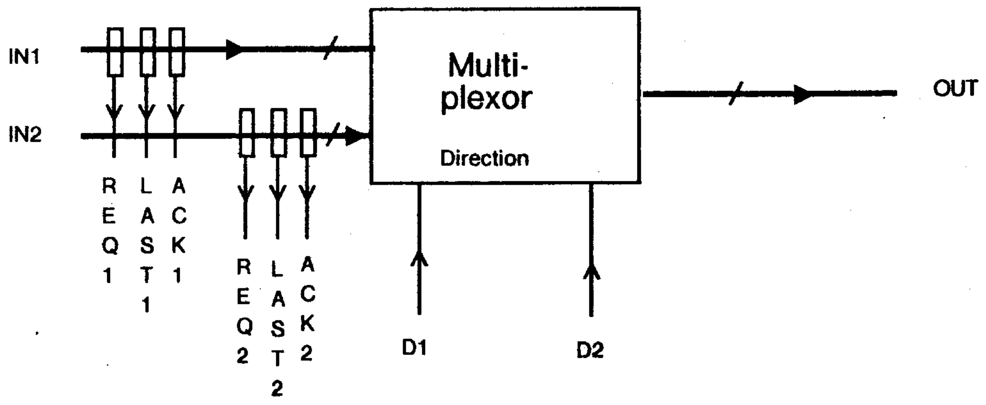


Figure 5.6b Data flow section of M2

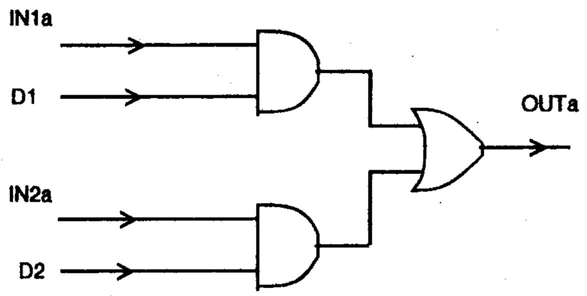


Figure 5.6c Design of the multiplexor for all signals except the acknowledge signal

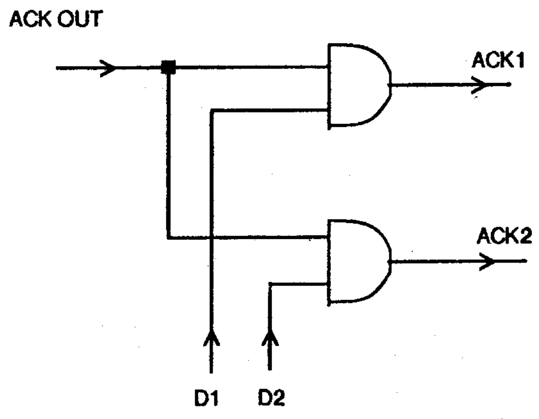


Figure 5.6d Design of the multiplexor for the acknowledge signal

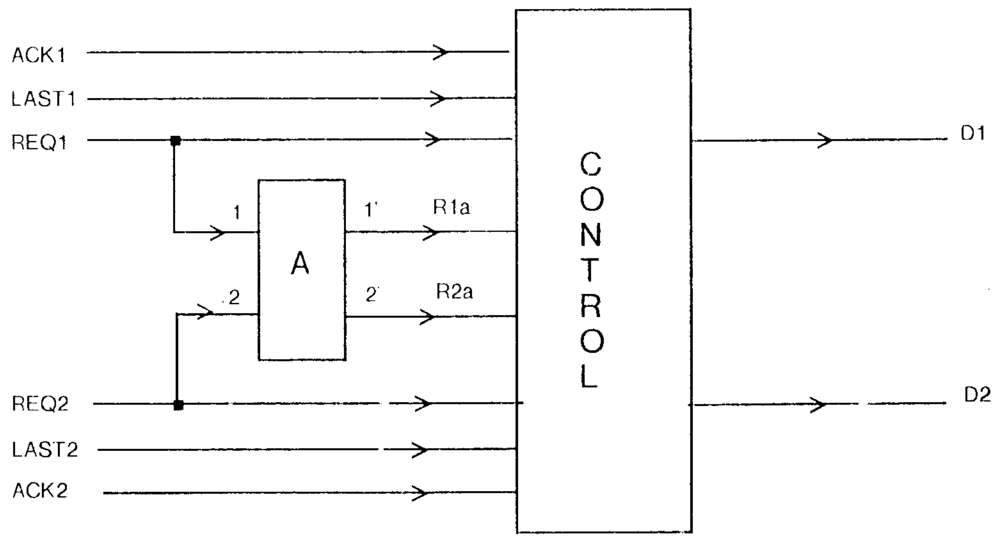


Figure 5.7a Block diagram of the control section of M2

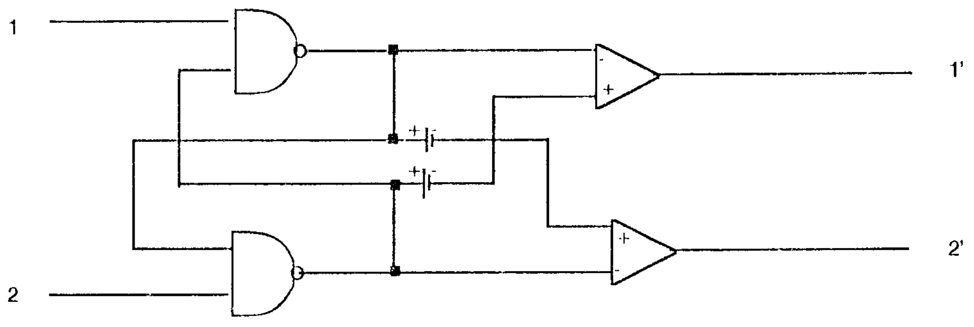
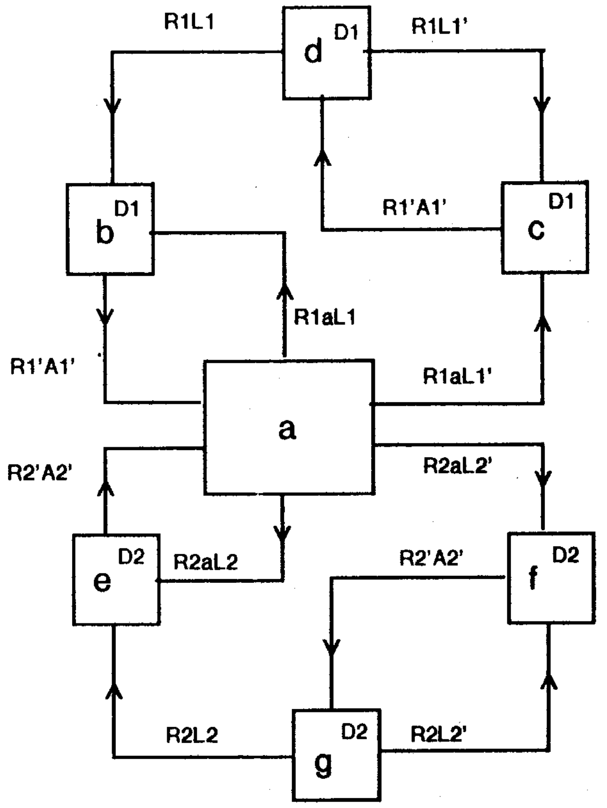


Figure 5.7b Design of the arbiter 'A'



Abbreviations -	R1 - is the REQ1 (request1) signal. R2 - is the REQ2 (request2) signal. A1 - is the ACK1 (acknowledge1) signal. A2 - is the ACK2 (acknowledge2) signal. L1 - is the LAST1 (last byte of packet 1) signal. L2 - is the LAST2 (last byte of packet 2) signal.
Signal notation -	R1' - stands for signal "R1" inverted, etc.. R1a, R2a - are the signals at the output of the arbiter.
Diagram notation -	States are in the center of the boxes in lower case. Outputs that are "1" are in the top right corner of the states in upper case.
State Definition -	"a" - Idle, no packet is being transmitted from either input. "b" - The last byte of a packet is being output from IN1. "c" - A byte of a packet is being output from IN1. The current byte is not the last byte of the packet. "d" - A byte has been transmitted from IN1. More bytes remain to be transmitted. "e" - The last byte of a packet is being output from IN2. "f" - A byte of a packer is being output from IN2. The current byte is not the last byte of the packet. "g" - A byte has been transmitted from IN2. More bytes remain to be transmitted.

Figure 5.8 State transition diagram for the control section of M2

Present State	Inputs		Next State	Outputs
	R	R		
	A L R 1	A L R 2		
1 2 3	1 1 1 a	2 2 2 a	1 2 3	D1 D2
0 0 0	- 0 - 1	- - - -	0 1 0	0 0
0 0 0	- 1 - 1	- - - -	0 0 1	0 0
0 0 0	- - - -	- 0 - 1	1 0 1	0 0
0 0 0	- - - -	- 1 - 1	1 0 0	0 0
0 0 1	- - 1 -	- - - -	0 0 1	1 0
0 0 1	1 - - -	- - - -	0 0 1	1 0
0 0 1	0 - 0 -	- - - -	0 0 0	1 0
0 1 0	- - 1 -	- - - -	0 1 0	1 0
0 1 0	1 - - -	- - - -	0 1 0	1 0
0 1 0	0 - 0 -	- - - -	0 1 1	1 0
0 1 1	- - 0 -	- - - -	0 1 1	1 0
0 1 1	- 0 1 -	- - - -	0 1 0	1 0
0 1 1	- 1 1 -	- - - -	0 0 1	1 0
1 0 0	- - - -	- - 1 -	1 0 0	0 1
1 0 0	- - - -	1 - - -	1 0 0	0 1
1 0 0	- - - -	0 - 0 -	0 0 0	0 1
1 0 1	- - - -	- - 1 -	1 0 1	0 1
1 0 1	- - - -	1 - - -	1 0 1	0 1
1 0 1	- - - -	0 - 0 -	1 1 0	0 1
1 1 0	- - - -	- - 0 -	1 1 0	0 1
1 1 0	- - - -	- 0 1 -	1 0 1	0 1
1 1 0	- - - -	- 1 1 -	1 0 0	0 1

Encoding	State
0 0 0	a
0 0 1	b
0 1 0	c
0 1 1	d
1 0 0	e
1 0 1	f
1 1 0	g

$$1^{n+1} = 1(2+3+A2+R2) + 1'2'3'R2a$$

$$2^{n+1} = 2(1'23L1R1+123'R2)' + 12'3A2'R2' + 1'2'3'L1'R1a$$

$$3^{n+1} = 3(1'2'3A1'R1'+1'23L1'R1+12'3A2'R2')' + 1'2'3'(L1R1a+L2'R2a) + 123'L2'R2 + 1'23'A1'R1'$$

$$D1 = 1'(2+3)$$

$$D2 = 1$$

Abbreviations -	
	R1 - is the REQ1 (request1) signal.
	R2 - is the REQ2 (request2) signal.
	A1 - is the ACK1 (acknowledge1) signal.
	A2 - is the ACK2 (acknowledge2) signal.
	L1 - is the LAST1 (last byte of packet1) signal.
	L2 - is the LAST2 (last byte of packet2) signal.

Signal notation -	
	R1' - stands for signal "R" inverted, etc..
	R1a, R2a - are the signals at the output of the arbiter.

Figure 5.9 State transition table for the control section of M2

output port. When the packet byte has been accepted at the output port the acknowledge signal of the output port should change to "1." The multiplexor will transmit this signal to the acknowledge signal of the input port "IN1." The request signal of the input port "IN1" should next reset which will cause the request signal of the output port to change to "0." The acknowledge signal of the output port should now reset followed by the resetting of the acknowledge signal of the input port "IN1." When both the acknowledge and request signals of the input port "IN1" have reset it indicates that the transmission of the packet byte through 'M2' is complete. Since this was the first and last byte of the packet 'M2' should return to the idle state and reset the "D1" input to the multiplexor.

The behavior of 'M2' for the other cases is similar except that it must not return to the idle state after transmitting the first byte of a packet if the first byte is not also the last byte of the packet. For example, if a byte of a packet (that is not the last byte) has been transmitted from the input port "IN1" 'M2' enters state "d." The important difference between the states "a" and "d" is that the "D1" input to the multiplexor is "1" in state "d." The input port "IN1" is thus connected to the output port of the multiplexor for the transmission of the entire packet.

It is important to note that in the idle state 'M2' checks the "R1a" and "R2a" signals to see if the address byte of a packet is defined at an input port. However, for all subsequent bytes of a packet (if there are any) 'M2' checks the request signal of the input port. If the second, or later, byte of a packet is being transmitted from an input port the output port must already be granted to this input port and arbitration is not required. Also, the request signal of the blocked input can pass through to the output of the arbiter. If we used the request signals at the output of the arbiter to define the availability of a packet byte, for the second or later bytes of a packet, a deadlock could occur.

The component count for the control section of 'M2' is given below:

1) 11 inverters.

2) 47 'and' gates.

3) 14 'or' gates.

4) 3 flip-flops.

5) 2 'nand' gates.

6) 2 comparators.

5.1.3 Component Count for the Synchronous Design

The component count for the synchronous design is twice the component count of M1 and M2. This is given below:

1) 40 inverters.

2) 262 'and' gates.

3) 4 'nand' gates.

4) 74 'or' gates

5) 12 flip-flops.

6) 4 comparators.

5.2 Self-Timed Design of the 2X2 Router

This section will present a design of the 2X2 router using the methodology proposed in this thesis. The input and output ports of the router include the signals mentioned earlier in this chapter. The input/output port signals of the self-timed design differ from those of the synchronous design since the data signals are encoded using the dual rail signalling protocol and the explicit single rail request signal is not transmitted.

The block diagram of the router is given in Figure 5.10. The design of the router is specified in terms of three machines 'M1,' 'C1,' and 'M2.' 'M1' outputs every byte of a packet at its input port to its output port with two additional signals "F" and "D." "F" is the first byte of packet signal and "D" is the direction of packet signal. 'M1' is in the reset state when no packet is being transmitted to the output port. When the address byte of a packet becomes defined at an input port 'M1' must transmit it to the output port with the signal "F" set to "1" and the signal "D" set the value of the least significant data signal of the packet byte. The value of the direction signal must be saved so that the signal "D" is set to the correct value for the transmission of the remaining bytes of the packet. The value of the signal "F" is set to "0" for the transmission of all subsequent bytes of the packet. When the "last_byte_of_packet" signal of the input port becomes "1" it indicates that 'M1' should return to the reset state after transmitting the current byte of the packet.

The machine 'C1' outputs a byte from the input port to the output port 0 if the direction signal "D" is false, and otherwise outputs the input byte to the output port 1. The direction signal "D" is only used to route each byte of a packet to the appropriate output and is not transmitted at either output port. The signals at the output ports of 'C1' are therefore the data signals of the packet byte at the input port of the router and the first byte of packet signal "F."

Machine 'M2' is a packet multiplexor, where the request to output a packet can come from both inputs concurrently. 'M2' is in the idle state when no packet is being transmitted from either input port. When an input port becomes defined 'M2' checks to see if it is the first byte of a packet by examining the signal "F." If the current byte is the first byte of the packet it must arbitrate for the resource. If arbitration is successful all bytes of the packet from this input port are transmitted to the output port. If arbitration is unsuccessful the input port is blocked until all bytes of the packet from the other input are transmitted. The signal "F" is generated within the router to aid 'M2' in performing its task, but this signal is not transmitted to the output port. The signals at the output port of 'M2' (and hence the router) are therefore identical to those at the input ports of the router.

5.2.1 Detailed Design of M1

'M1' is a self-timed finite state machine whose terminal structure and block diagram are given in Figures 5.11a,b. 'User1' is a 'user' module that defines the next state and output as a combinatorial function of the inputs and the present state of the machine. The state variables of the machine are stored in the feedback registers 'Reg' which consist of three 'register' modules connected in series. 'J' is a 'join' module that is used to synchronize the inputs and the present state of the machine, and 'F' is a 'fork' module that forwards the next state variables to the feedback registers and the outputs to the output port.

The state transition diagram of 'M1' is given in Figure 5.12, and its realization in Figure 5.13. When no packet is being transmitted 'M1' is in the idle state "a." It must remain in this state until the address byte of a packet is defined at the input port. Since 'M1' is a self-timed finite state machine it will remain in the idle state until the data signals of the input port become defined. When the input port becomes defined 'M1' sets the first byte of packet signal "F" to "1" and the direction signal "D" to the value of the least significant data signal of the packet byte. After transmitting the address byte of the packet with the "F" and "D" signals 'M1' must enter a new

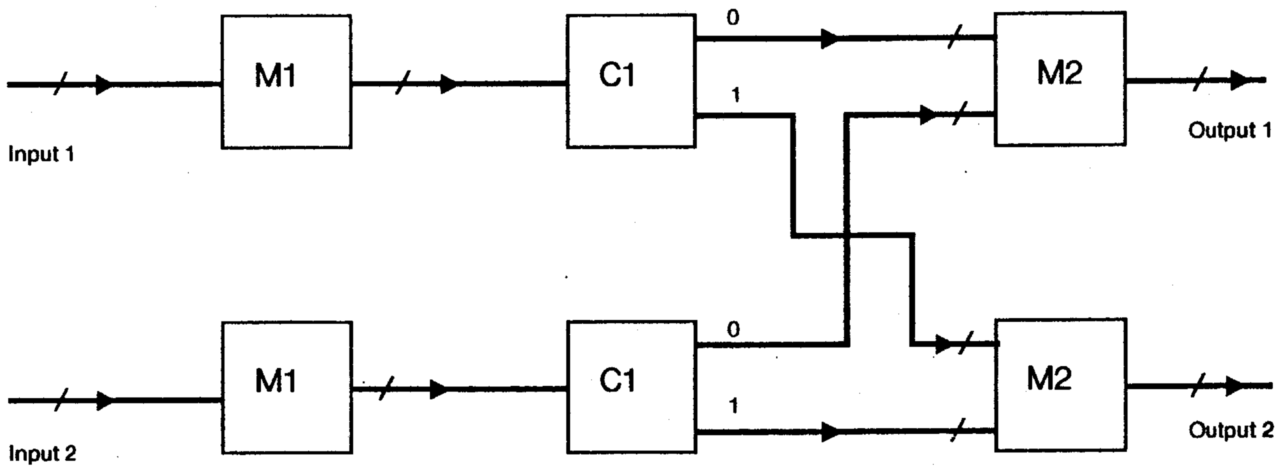


Figure 5.10 Block Diagram of the Self-Timed Router

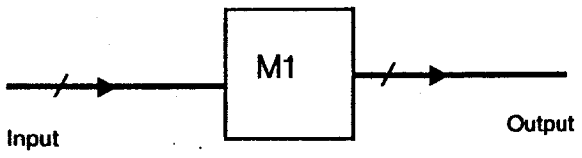


Figure 5.11a Terminal Structure of M1

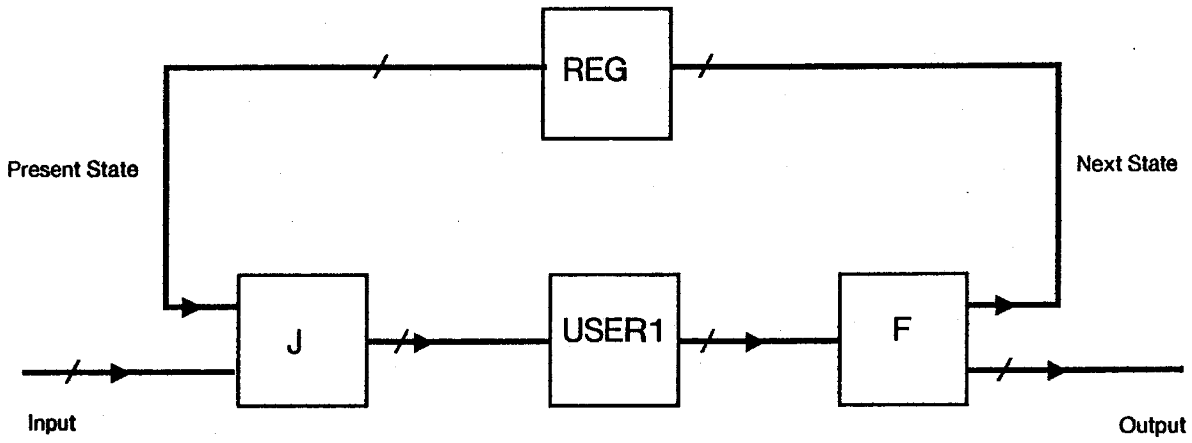
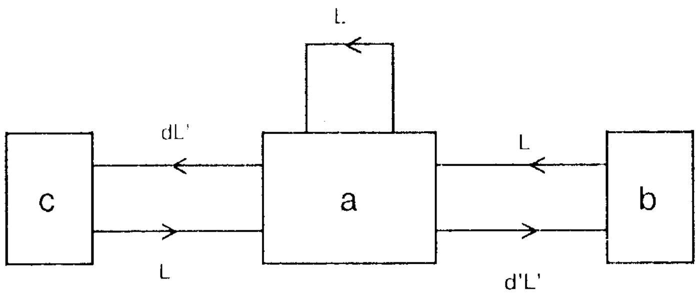


Figure 5.11b Block Diagram of M1



Abbreviations -	"d"	- This is the direction signal of the packet. It is defined by the least significant data signal of the address byte.
	"L"	- This is the last byte of packer signal.
Signal notation	L' stands for signal "L" inverted.	
Diagram notation	States are in the center of the boxes in lower case.	
State definition	"a"	- Idle, no packet is being transmitted.
	"b"	A byte of the packet is being output to output port1 of the router.
	"c"	- A byte of the packet is being output to output port2 of the router.

Figure 5.12 State transition diagram of M1

Present State 1 2	Inputs d L	Next State 1 2	Outputs F D
0 0	0 0	0 1	1 0
0 0	0 1	0 0	1 0
0 0	1 0	1 0	1 1
0 0	1 1	0 0	1 1
0 1	- 0	0 1	0 0
0 1	- 1	0 0	0 0
1 0	- 0	1 0	0 1
1 0	- 1	0 0	0 1

Encoding	State
0 0	a
0 1	b
1 0	c

$$1^{n+1} = 1L' + 2'L'd$$

$$2^{n+1} = 2L' + 1'L'd'$$

$$F = 1'2'$$

$$D = 1 + 2'd$$

Figure 5.13 State transition table of M1

state that depends on the value of the "last_byte_of_packet" signal and the direction of packet signal. For example, if the "last_byte_of_packet" signal is true, 'M1' should return to the idle state since the next byte will be the address byte of a new packet. However, if the first byte is not the last byte of the packet 'M1' must remember the value of the direction of packet signal. For example, if the direction of packet signal is true 'M1' will enter state "c." For the transmission of the remaining bytes of the packet 'M1' will set the signal "F" to "0" and the signal "D" to "1." If in transmitting the remaining bytes of the packet if the "last_byte_of_packet" signal is "0" 'M1' will remain in the state "c" since more bytes of the packet remain to be output. However, if the "last_byte_of_packet" is "1," 'M1' must return to the idle state. The behavior of 'M1' is similar when the first byte is not the last byte of the packet and the direction signal is false.

The component count for 'M1' is given below. All component counts for the self-timed design are obtained from the design of the modules given in Chapters 3 and 4.

1) 42 'and' gates.
2) 64 'or' gates.
3) 15 inverters.
4) 3 'exclusive-or' gate.
5) 54 'C' gates.

5.2.2 Detailed Design of C1

'C1' is a conditional machine whose terminal structure and block diagram are given in Figures 5.14a,b. 'Ca' is a 'conditional' module that routes all the signals at the input port, except the direction of packet signal "D," to one of the two output ports. 'P1' is the identity 'predicate' module whose input port data signals are connected to the direction of packet signal "D." If the input port becomes defined and the direction of packet signal is "D" is false, all signals of the input port, except "D," are output to the output port "Output0." Similarly, if the direction of packet signal "D" is true, all signals of the input port, except "D," are output to the output port "Output1." The component count for 'C1' is given below:

1) 3 'or' gates.
2) 40 'and' gates.

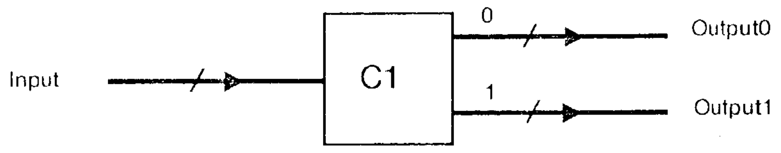


Figure 5.14a Terminal Structure of C1

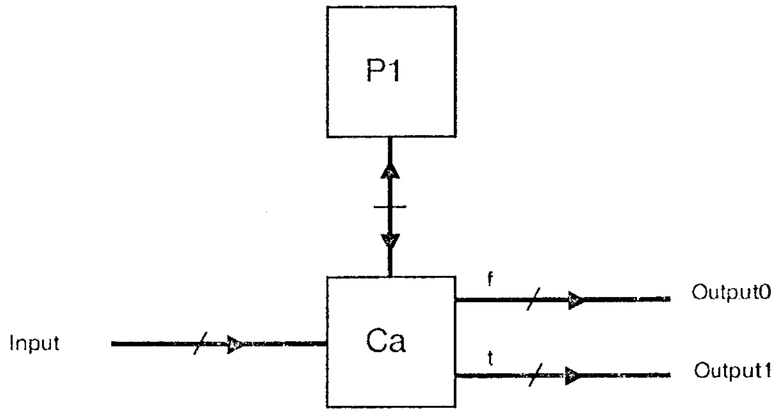


Figure 5.14b Block Diagram of C1

5.2.3 Detailed Design of M2

'M2' is a packet multiplexor whose terminal structure and block diagram are given in Figures 5.15a,b. Every byte of a packet at an input port is separated into a control section and a data section. Each byte of a packet can only be output after its associated control signal has successfully arbitrated for the resource. If the control signal of one input port successfully arbitrates for the resource then the control signal of the other input port is blocked until all bytes of the selected port are transmitted. For every byte of a packet from the selected input port the control signal of the other input port is blocked if the current byte is not the last byte of the packet, and unblocked if the current byte is the last byte of the packet.

'F' is a 'fork' module that separates the signals of an input port into the control and data sections. The first byte of packet signal "F" and the "last_byte_of_packet" signal of the input port form the control section and are output at port "1," while the rest signals of the input port form the data section and are output at port "2." 'J' is a 'join' module that allows a byte of a packet to be output only after its associated control signal has successfully arbitrated for the resource. The "2" input port of the module receives the "last_byte_of_packet" signal of the packet byte after it has successfully arbitrated for the resource. Although the first byte of packet signal "F" is a control signal it is not output since it is not part of the packet byte. 'U2' is a 'union' module that outputs the bytes of the selected packet to the output port of 'M2.'

'C2' is a conditional machine whose terminal structure and block diagram are given in Figures 5.16a,b. The signals at the input port of 'C2' are the "last_byte_of_packet" signal and the first byte of packet signal "F." 'Cb' is a conditional module that conditionally transmits the "last_byte_of_packet" signal of the input port to one of the two output ports. 'P2' is the identity predicate module whose input port data signals are connected to the first byte of packet signal "F." The "last_byte_of_packet" signal is thus transmitted to the output port "Output0" if "F" is false, and otherwise transmitted to the output port "Output1."

If the current byte is the first byte of the packet then the associated control signal must first arbitrate for the resource. This is accomplished by transmitting the "last_byte_of_packet" signal through the 'arbitration' module 'A' and the 'switch' module 'S.' The 'arbitration' module guarantees that if the control signals of the first byte of both input packets are arbitrating for the resource only one will pass through the module. The 'switch' module 'S' is used to block the

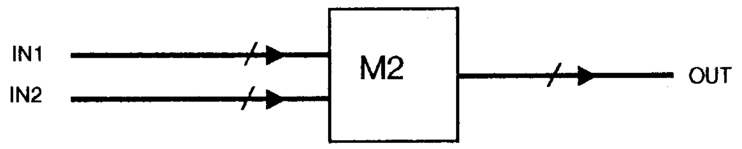


Figure 5.15a Terminal Structure of M2

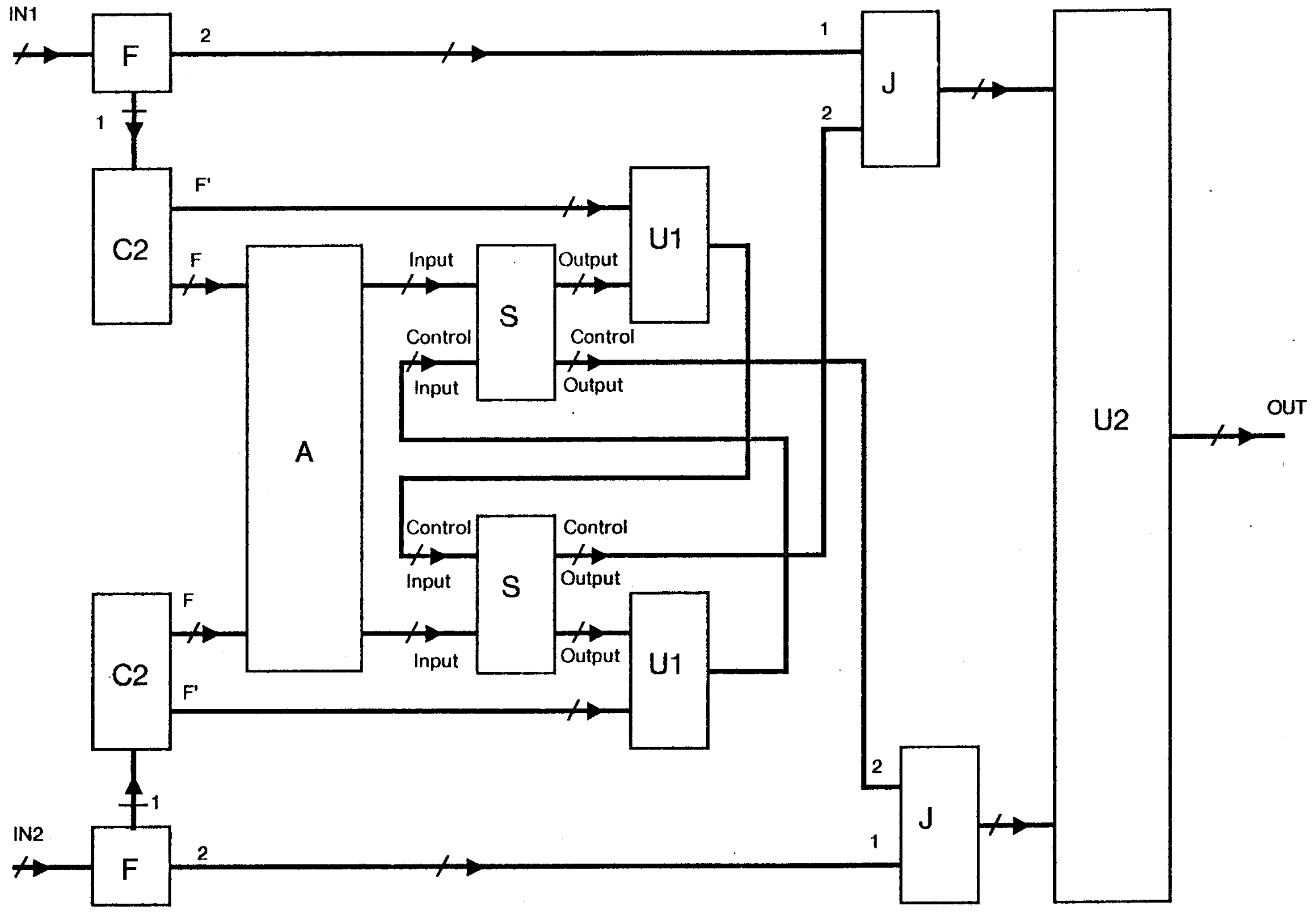


Figure 5.15b Block Diagram of M2

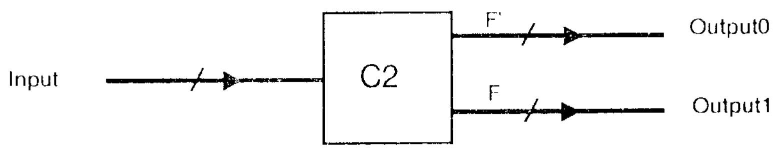


Figure 5.16a Terminal Structure of C2

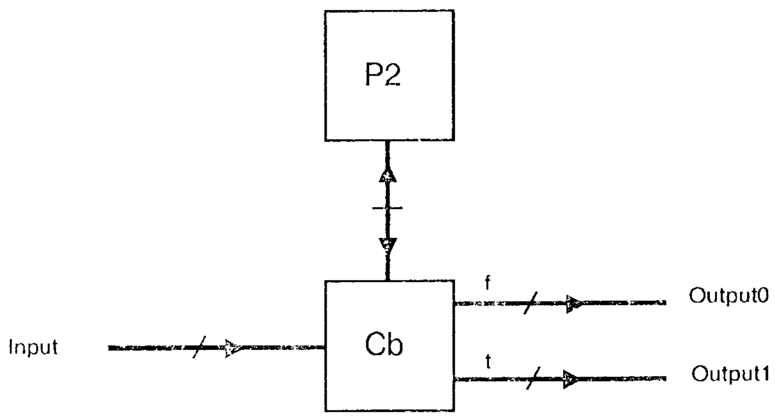


Figure 5.16b Block Diagram of C2

control signal of an input if the resource is already in use by the other input. The data to be switched is input/output at the input and output ports respectively. Initially the state of both 'switch' modules is "on" so that the "last_byte_of_packet" signal at the output of the 'arbitration' module will also pass through its 'switch' module. Next, the switch for the other input must be turned off if this is not the last byte of the packet, and otherwise the switch for the other input must be turned on if this is the last byte of the packet. This is accomplished by transmitting the "last_byte_of_packet" signal through the 'union' module 'U1' to the control input port of the 'switch' module for the other input. The control signal passes through to the control output port after setting the state of the 'switch' module, and joins the corresponding data signals at its 'join' module "J."

If the current byte is not the first byte of the packet then the resource must already be granted to this input. The behavior is similar to the case for the first byte of the packet except that it is not necessary to arbitrate for the resource. In this case the "last_byte_of_packet" signal is transmitted from the the "F'" output port of 'C2' to the input of the 'union' module "U1" directly, thus bypassing the 'arbitration' module and the 'switch' module.

The component count for 'M2' is twice the component count of 'F,' 'C2,' 'S,' 'U1,' 'J' and the component count of 'A' and 'U2.' This is given below:

1) 4 inverters.
2) 90 'or' gates.
3) 102 'and' gates.
4) 16 'nand' gates.
5) 16 comparators.
6) 8 'C' gates.

The component count for the entire router is twice the component count of 'M1,' 'C1' and 'M2.' This is given below:

1) 38 inverters.
2) 314 'or' gates.
3) 368 'and' gates.
4) 32 'nand' gates.
5) 32 comparators.
6) 124 'C' gates.

7) 6 'exclusive-or' gates.

5.3 Comparison of the Design Methodologies

This section will compare the design of the synchronous and self-timed routers. This is not meant to be an absolute comparison of the two design methodologies, but merely an indication of the benefits and drawbacks of each. The issues considered in this comparison are: logical complexity, performance, wiring complexity, ease of design and verification.

From the previous sections it is obvious that the logical complexity of the self-timed design is greater than that of the synchronous design. This is a first order comparison of the two designs. All component counts for the 'and' and 'or' gates were given in terms of dual input gates. Also, no attempt had been made to optimize the design for a particular integrated circuit technology. For example, in nMOS VLSI, one can take advantage of *pass* transistors to reduce the complexity of a design.

One measure of the performance of a system can be based on the number of logic gates between the input and output of the system. This model assumes that the interconnect delays are negligible in comparison to the delays of the logic elements. For the synchronous system this also assumes that all delays between clocked storage elements are equal to the clock period. The clock period is thus optimized so that that it does not add to the delay of the system in any way. For the self-timed router, a packet at an input port must propagate through 78 levels of logic before it can appear at an output. The synchronous router, however, only requires 29 levels of logic from an input port to an output port. There may be additional delays from an input port to an output port of the synchronous or self-timed routers if there are any arbitration delays. Synchronous systems can be expected to have a higher performance than self-timed systems when the size of the system is so small that the interconnect delays are negligible in comparison to the delays of the logic elements and it is possible to optimize the clock period so that it does not add to the delay of the system. As mentioned in Chapter1, this will not be a valid assumption for VLSI circuits in the future. The communication delays will be more troublesome for the synchronous router since it lacks locality of communication. The self-timed router, on the other hand, has locality of communication as information transfer is only between adjacent modules. If the router is to be a part of a larger system incorporated on an entire VLSI chip, the self-timed router can be expected to have a higher performance than the synchronous router.

The self-timed router requires approximately twice the number of wires compared to the synchronous router. This is due to the dual-rail signaling convention used in the self-timed design methodology, and the single-rail signaling convention used in the synchronous design methodology. The wires lengths of the two routers are comparable, except for the clock signal of the synchronous router which is global to the system.

The design of the synchronous router involves the design of the components of the router, their connection, and the verification of timing constraints global to the system. The design of the self-timed router is however independent of any timing considerations (given that the set of self-timed modules exist). This is analogous to designing a conventional computer program; one is concerned with the textual content of the program and not the speed with which the various instructions are executed. The design and verification of the self-timed router is much simpler than that of the synchronous router. Though the logical complexity of the self-timed router is greater, its conceptual complexity is not. For example, the machines 'M1,' and 'M2' of the synchronous router have many more states than the design of the machine 'M1' of the self-timed router. The task of the designer of the synchronous router includes managing the communication protocol at the inputs and outputs of every machine of the system. In a self-timed system, however, this is all automated since the channel communication protocol is incorporated into the design of the modules.

The self-timed router is definitely easier to design and debug. It will require more logic elements than the synchronous router, but can be expected to have a higher performance when incorporated in large VLSI systems.

6. Summary and Conclusion

This thesis has presented a design methodology for self-timed systems which should be extremely attractive for implementing systems in VLSI. Self-timed systems are characterized by the absence of any timing reference to which all operations are synchronized. This is in contrast to synchronous systems where all operations are synchronized to the global system clock.

In the proposed methodology, a self-timed system is a legal interconnection of self-timed modules. Self-timed modules thus form the atomic building blocks of every self-timed system. A self-timed module ,in general, has a set of input and output ports through which it communicates with the other modules of the system using the asynchronous reset signaling protocol. The wires that connect the input port of one module to the output port of another module are called channels. The methodology does not separate the interconnection of modules into a data flow and control section; the signals of a channel include both control and data information.

There are 12 different module types in the proposed design methodology whose behavior has been given in Chapter2. The choice of modules has been influenced by familiar programming constructs such as iterations, conditionals and by constructs that allow for parallelism and synchronization, such as forking and joining. Non-determinacy can be introduced into a self-timed system by incorporating the non-deterministic 'arbitration' and 'union' modules, which in addition can be used to share hardware resources.

The thesis presents a template for the design of the modules in Chapters 3 and 4. Using these templates it is possible to realize an arbitrary module of a given type. The designs also specify the timing requirements that must be maintained within an arbitrary module of a given type to ensure that its behavior is independent of the channel delays at its input and output ports.

The behavior of a self-timed system is guaranteed to be independent of the delays of the channels connecting the modules by ensuring that the signals at the output of each module change only after all signals internal to the module have stabilized. The only exceptions to this are the cases where a self-timed system has the potential of deadlocking, or when the system is non-determinate. If the system has the potential of deadlocking, its behavior can be influenced to either cause or prevent deadlocks by choosing appropriate values for the channel delays. While if a system is non-determinate, its behavior can be influenced by the channel delays in a path leading to an input port of a non-determinate 'arbitration' or 'union' module.

Chapter 5 contrasted the design of a 2X2 router using the design methodology proposed in this thesis and a synchronous methodology. This design exercise illustrated that the logical complexity of the self-timed design was approximately two and a half times that of the synchronous design, although the self-timed design was conceptually simpler since it dealt with higher level constructs. This design example was for a relatively small system, and it is hard to appreciate the advantages of the self-timed methodology for such a small design.

For the design of complex systems in VLSI with 100,000 or more logic devices the self-timed methodology will be more attractive due to its relative ease of design and higher perfOrmance. Systems of this magnitude can be expected to have a higher performance when designed using the self-timed methodology compared to a synchronous methodology since their behavior is not limited by the worst case delays of the system and it is possible to take advantage of locality of communication. The main disadvantages of synchronous VLSI systems will be their communication costs and the prohibitive task of maintaining global timing constraints. However, there are certain specialized synchronous systems where the self-timed methodology would not be appropriate due to its logical complexity. These include memories, pipelined architectures and other special purpose architectures exhibiting locality of communication and the absence of global busses.

6.1 Areas for Further Research

The module interconnection rules of the proposed self-timed design methodology do not guarantee the absence of deadlocks, they merely prevent the obvious ones from occurring. Due to the extremely undesirable nature of this problem it would be appropriate to come up with a theory of deadlock avoidance using the proposed module types. Hopefully this theory will lead to improved module interconnection rules that will guarantee the absence of deadlocks in a self-timed system. If this is not possible, the theory should at least provide an algorithm that will identify all the cases where the possibility of deadlocks exists in a self-timed system.

The existing methodology does not aid in the hierarchical design of a system; this is essentially the task of the designer. It would be extremely attractive to incorporate this design methodology into a top down design environment, such as the ADL (architectural design language) proposed by Leung [7]. At the lowest level of the design process, a textual description of the various modules could be automatically translated into a layout using a simple macro type expansion.

Another important question of interest is to analyze the various module types and to see if they are too primitive or too complex from the standpoints of performance, logical complexity, deadlock avoidance, ease of design etc. At one extreme it is possible to make the self-timed modules as primitive as the logical 'and' and 'or' gates, but it is doubtful if this would be of practical value. It is easier to optimize a design if the constructs are more primitive, but the design effort must be greater. If the modules are too complex it will not be possible to optimize the design, and the methodology must also provide for many module types since each module will lack universal applicability.

One of the strongest criticisms from present designers will probably be regarding the absence of the familiar constructs such as PLAs and memories. The methodology presented for realization of dual-rail combinatorial circuits in Chapter 4, though functionally correct, is not satisfactory from a complexity viewpoint. It would be appropriate to come up with a more efficient design methodology for realizing dual rail combinatorial circuits based on the familiar concepts of PLAs and memories which are omnipresent in most synchronous systems.

Bibliography

[1] Armstrong, D.B., Arthur, D.F. and Menon, P.R. "Design of asynchronous circuits assuming unbounded gate delays," IEEE Transactions on Computers 18, 12 (December 1969), 1110-1120.

[2] Chaney, T.J. and Mulnar, C.E. "Anomalous behavior of synchronizer and arbiter circuits," IEEE Transactions on Computers 22, 4 (April 1973), 421-422.

[3] Chaney, T.J., Ornstein, S.M. and Littlefield, W.M. "Beware the synchronizer," COMPCON-72 IEEE Computer Society Conference, San Francisco, California, September 1972, 317-340.

[4] Clark, B. "A speed independent implementation of data flow schemas," Computation Structures Group Memo 82, Laboratory for Computer Science, MIT, Cambridge, Ma., July 1972.

[5] Conway, L. and Mead, C. Introduction to VLSI Systems, Reading, Ma., Addison-Wesley, 1980.

[6] Hon, R.W. "Implementing VLSI systems in a research environment," Proceedings of the Caltech Conference on VLSI, January 1979, 139-145.

[7] Leung, C.K. "ADL: An architecture description language for packet communicating systems," Computation Structures Group Memo 185, Laboratory for Computer Science, MIT, Cambridge, Ma., Oct. 1979.

[8] Misunas, D.P. "Petri nets and speed independent design," Communications of the ACM 16, 8 (August 1973), 474-481.

[9] Moore, G.E. "Are we really ready for VLSI," Proceedings of the Caltech Conference on VLSI, January 1979, 3-14.

[10] Mohsen, A. "Device and circuit design for VLSI," Proceedings of the Caltech Conference on VLSI, January 1979, 31-54.

[11] Patil, S.S. "An asynchronous logic array," Technical Memo 62, Laboratory for Computer Science, MIT, Cambridge, Ma., May 1975.

[12] Peterson, J.L. "Petri nets," ACM Computing Surveys 9, 3 (September 1977), 223-252.

[13] Redford, J.L. "A design for a routing module," Computation Structures Note 39, Laboratory for Computer Science, MIT, Cambridge, Ma., February 1979.

[14] Seitz, C.L. "Self timed VLSI systems," Proceedings of the Caltech Conference on VLSI, January 1979, 345-355.

[15] Shannon, C.E. "Symbolic analysis of relay and switching circuits," Trans. of AIEE 57 (1938), 713-723.

[16] Course Notes on 6.032 Computation Structures, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., 1979.

[17] Course Notes on Cs284 Self-timed Systems, Caltech, Department of Computer Science, Pasadena, California, 1978.